# Introduction to Scientific Programming in C++17/Fortran2008

The Art of HPC, volume 3

Victor Eijkhout

2017–2022, formatted March 14, 2023

Book and slides download: https://tinyurl.com/vle322course

Public repository: https://bitbucket.org/VictorEijkhout/intro-programming-public

# Contents

# PART I

# INTRODUCTION

# Chapter 1

# Introduction

## 1.1 Programming and computational thinking

In this chapter we take a look at the history of computers and computer programming, and think a little about what programming involves.

### 1.1.1 History

In the early days of computing, hardware design was seen as challenging, while programming was little more than data entry. The fact that one of the earliest programming languages was called 'Fortran', for

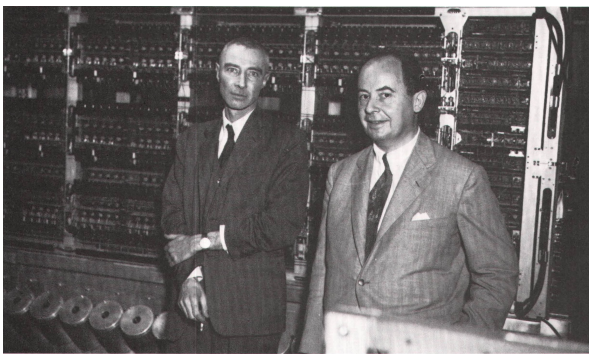


Figure 1.1: Robert Oppenheimer and John von Neumann

'formula translation', speaks to this: once you have the math, programming was thought to be nothing more than translating the math into code. The fact that programs could have subtle errors, or *bugs*, came as quite a surprise to the earliest computer designers.

The fact that programming was not as highly valued also had the side-effect that many of the early programmers were women. Before electronic computers, a 'computer' was a person executing computations, probably with a mechanical calculating device, and often these were women. From this, the earliest people programming electronic computers to perform these calculations were, usually mathematically educated, women. Two famous examples were Navy Rear-admiral Grace Hopper, inventor of the Cobol language, and Margaret Hamilton who led the development of the Apollo program software. This situation changed after the 1960s and certainly with the advent of PCs[1].

1. http://www.sysgen.com.ph/articles/why-women-stopped-coding/27216

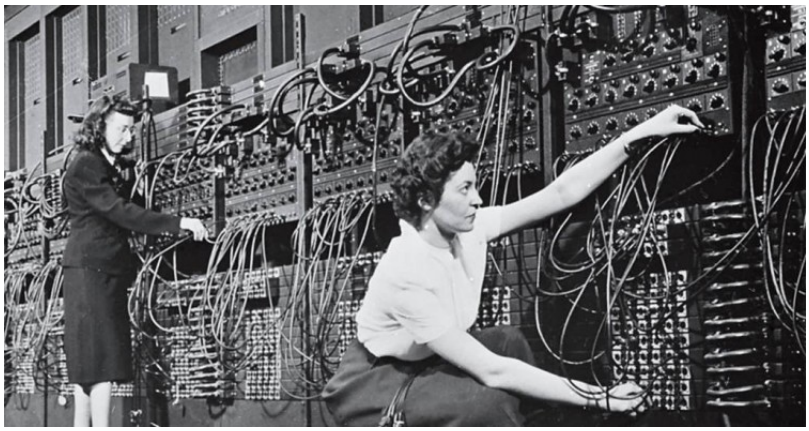


Figure 1.2: Programming the ENIAC

### 1.1.2 Is programming science, art, or craft?

As the previous section argued, programming is more than simple translation of math into instructions for hardware. Could it be a science? There are certainly scientific aspects to programming:

- Algorithms and complexity theory have a lot of math in them.
- Programming language design is another mathematically tinged subject.

However, programming itself is not a science.

The term 'software engineering' may lead you to suspect that designing and producing software is an engineering discipline, but this is also not quite the case. There is no certification for software engineers, and there is no body of accepted techniques the way there is for civil engineering and such disciplines.

For a large part programming is a discipline. What constitutes a good program is a matter of taste. That does not mean that there aren't recommended practices. In this course we will emphasize certain practices that we think lead to good code, as likewise will discourage you from certain idioms.

None of this is an exact science. There are multiple programs that give the right output. However, programs are rarely static. They often need to be amended or extended, or even fixed, if erroneous behavior comes to light, and in that case a badly written program can be a detriment to programmer productivity. An important consideration, therefore, is intelligibility of the program, to another programmer, to your professor in this course, or even to yourself two weeks from now.

### 1.1.3 Computational thinking

Mathematical thinking:

- Number of people per day, speed of elevator ⇒ yes, it is possible to get everyone to the right floor.
- Distribution of people arriving *etc.* ⇒ average wait time.

Sufficient condition $\neq$ existence proof.

Computational thinking: actual design of a solution

- Elevator scheduling: someone at ground level presses the button, there are cars on floors 5 and 10; which one do you send down?

Coming up with a strategy takes creativity!

**Exercise 1.1.** A straightforward calculation is the simplest example of an algorithm.

Calculate how many schools for hair dressers the US can sustain. Identify the relevant factors, estimate their sizes, and perform the calculation.

**Exercise 1.2.** Algorithms are usually not uniquely determined:
there is more than one way solve a problem.

Four self-driving cars arrive simultaneously at an all-way-stop intersection. Come up with an algorithm that a car can follow to safely cross the intersection. If you can come up with more than one algorithm, what happens when two cars using different algorithms meet each other?

Looking up a name in the phone book

- start on page 1, then try page 2, et cetera
- or start in the middle, continue with one of the halves.

What is the average search time in the two cases?

Having a correct solution is not enough!

> A powerful programming language serves as a framework within which we organize our ideas. Every programming language has three mechanisms for accomplishing this:
>
> - primitive expressions
> - means of combination
> - means of abstraction
>
> *Abelson and Sussman, The Structure and Interpretation of Computer Programs*

- The elevator programmer probably thinks: ‘if the button is pressed’, not ‘if the voltage on that wire is 5 Volt’.
- The Google car programmer probably writes: ‘if the car before me slows down’, not ‘if I see the image of the car growing’.
- … but probably another programmer had to write that translation.

A program has layers of abstractions.

Abstraction means your program talks about your application concepts, rather than about numbers and characters and such.

Your program should read like a story about your application; not about bits and bytes.

Good programming style makes code intelligible and maintainable.

(Bad programming style may lead to lower grade.)

> The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague — Edsger Dijkstra

What is the structure of the data in your program?

Stack: you can only get at the top item

Queue: items get added in the back, processed at the front

A program contains structures that support the algorithm. You may have to design them yourself.

### 1.1.4 Hardware

Yes, it's there, but we don't think too much about it in this course.

Advanced programmers know that hardware influences the speed of execution; see HPC book [9], section 1.7.

### 1.1.5 Algorithms

> An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time
> [A. Levitin, Introduction to The Design and Analysis of Algorithms, Addison-Wesley, 2003]

The instructions are written in some language:

- We will teach you C++ and Fortran;
- the compiler translates those languages to machine language

- Simple instructions: arithmetic.
- Complicated instructions: control structures
  - conditionals

  - loops

- Input and output data: to/from file, user input, screen output, graphics.
- Data during the program run:
  - Simple variables: character, integer, floating point
  - Arrays: indexed set of characters and such
  - Data structures: trees, queues
    * Defined by the user, specific for the application
    * Found in a library (big difference between C/C++!)

## 1.2 About the choice of language

There are many programming languages, and not every language is equally suited for every purpose. In this book you will learn C++ and Fortran, because they are particularly good for scientific computing. And by 'good', we mean

- They can express the sorts of problems you want to tackle in scientific computing, and
- they execute your program efficiently.

There are other languages that may not be as convenient or efficient in expressing scientific problems. For instance, *python* is a popular language, but typically not the first choice if you're writing a scientific program. As an illustration, here is simple sorting algorithm, coded in both C++ and python.

Python vs C++ on bubblesort:

```
for i in range(n-1):
  for j in range(n-i-1):
    if numbers[j+1]<numbers[j]:
      swaptmp = numbers[j+1]
      numbers[j+1] = numbers[j]
      numbers[j] = swaptmp
```

```
for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (numbers[j+1]<numbers[j]) {
      int swaptmp = numbers[j+1];
      numbers[j+1] = numbers[j];
      numbers[j] = swaptmp;
    }
```

```
$ python bubblesort.py 5000
Elapsed time: 12.1030311584
$ ./bubblesort 5000
Elapsed time: 0.24121
```

But this ignores one thing: the sorting algorithm we just implemented is not actually a terribly good one, and in fact python has a better one built-in.

Python with quicksort algorithm:

```
numpy.sort(numbers,kind='quicksort')
```

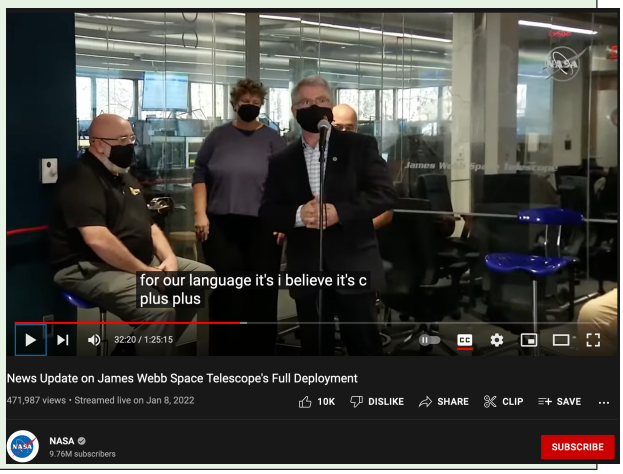
```
[] python arraysort.py 5000
```

```
Elapsed time: 0.00210881233215
```

So that is another consideration when choosing a language: is there a language that already comes with the tools you need. This means that your application may dictate the choice of language. If you're stuck with one language, don't reinvent the wheel! If someone has already coded it or it's part of the language, don't redo it yourself.

Application domains where C++ rules:

- Scientific computing; interoperability with C/Python code.
- Embedded processors
- Game engines



# Chapter 2

# Logistics

## 2.1 Programming environment

Programming can be done in any number of ways. It is possible to use an Integrated Development Environment (IDE) such as *Xcode* or *Visual Studio*, but for if you're going to be doing some computational science you should really learn a *Unix* variant.

- If you have a *Linux* computer, you are all set.
- If you have an *Apple* computer, it is easy to get you going. You can use the standard *Terminal* program, or you can use a full *X windows* installation, such as `XQuartz`, which makes Unix graphics possible. This, and other Unix programs can be obtained through a *package manager* such as *homebrew* or *macports*.
- *Microsoft Windows* users can use *putty* but it is probably a better solution to install a virtual environment such as *VMware* (`http://www.vmware.com/`) or *Virtualbox* (`https://www.virtualbox.org/`).

Next, you should know a text editor. The two most common ones are *vi* and *emacs*.

### 2.1.1 Language support in your editor

The two most popular editors are *emacs* and *vi* or *vim*. Both have support for programming languages, doing syntax coloring, and helping you with the correct indentation.. Most of the time, your editor will detect what language a file is written in, based on the file extension:

- `cxx,cpp,cc` for C++, and
- `f90,F90` for Fortran.

If your editor somehow doesn't detect the language, you can add a line at the top of the file:

```
// -*- c++ -*-
```

for C++ mode, and

```
! -*- f90 -*-
```

for Fortran mode.

Main advantages are automatic indentation (C++ and Fortran) and supplying block end statements (Fortran). The editor will also apply 'syntax coloring' to indicate the difference between keywords and variables.

## 2.2 Compiling

The word 'program' is ambiguous. Part of the time it means the *source code*: the text that you type, using a text editor. And part of the time it means the *executable*, a totally unreadable version of your source code, that can be understood and executed by the computer. The process of turning your source code into an executable is called *compiling*, and it requires something called a *compiler*. (So who wrote the source code for the compiler? Good question.)

Here is the workflow for program development

1. You think about how to solve your program
2. You write code using an editor. That gives you a source file.
3. You compile your code. That gives you an executable.
   Oh, make that: you try to compile, because there will probably be compiler errors: places where you sin against the language syntax.
4. You run your code. Chances are it will not do exactly what you intended, so you go back to the editing step.

## 2.3 Your environment

The following exercise is for the situation where there is a central class computer. To prove that you have your connectivity sorted out, do the following.

**Exercise 2.1.** Do an online search into the history of computer programming. Write a page, if possible with illustration, and turn this into a `pdf` file. Submit this to your teacher.

# Chapter 3

# Teachers guide

This book was written for a one-semester introductory programming course at The University of Texas at Austin, aimed primarily at students in the physical and engineering sciences. Thus, examples and exercises are as much as possible scientifically motivated. This target audience also explains the inclusion of Fortran.

This book is not encyclopedic. Rather than teaching each topic in its full glory, the author has taken a 'recommended practices' approach, where students learn enough of each topic to become a competent programmer. (Recommended by who you might ask? The author freely admits being guided by his own taste. However, he lets himself be informed by plenty of other current literature.) This serves to keep this book at a manageable length, and to minimize class lecture time, emphasizing lab exercises instead.

Even then, there is more material here than can be covered and practiced in one semester. If only C++ is taught, it is probably possible to cover the whole of Part II; for the case where both C++ and Fortran are taught, we have a suggested time line below.

## 3.1 Justification

The chapters of Part II and Part III are presented in suggested teaching order. Here we briefly justify our (non-standard) sequencing of topics and outline a timetable for material to be covered in one semester. Most notably, Object-Oriented programming is covered before arrays, and pointers come very late, if at all.

There are several thoughts behind this. For one, dynamic arrays in C++ are most easily realized through the `std::vector` mechanism, which requires an understanding of classes. The same goes for `std::string`.

Secondly, in the traditional approach, object-oriented techniques are taught late in the course, after all the basic mechanisms, including arrays. We consider OOP to be an important notion in program design, and central to C++, rather than an embellishment on the traditional C mechanisms, so we introduce it as early as possible.

Even more elementary, we emphasize range-based loops as much as possible over indexed loops, since ranges are increasing in importance in recent language versions.

### 3.1.1 Algorithms

Some of the programming exercises in this course ask the student to reproduce algorithms that exist in the `std::algorithm` header. Thus this course could be open to the criticism that students should learn their Standard Template Library (STL) algorithms, rather than recreating them themselves. (See section 14.2.)

My defense would be that a programmer should know more than what algorithms to pick from the standard library. Students should understand the mechanisms behind these algorithms, and be able to reproduce them, so that, when this is needed, they can code variations of these algorithms.

## 3.2 Time line for a C++/F03 course

As remarked above, this book is based on a course that teaches both C++ and Fortran2003. Here we give the time line used, including some of the assigned exercises.

For a one semester course of slightly over three months, two months would be spent on C++ (see table 3.1), after which a month is enough to explain Fortran; see table 3.3. Remaining time will go to exams and elective topics.

### 3.2.1 Advanced topics

We also outline a 'C++ 101.5' course: somewhere between beginning and truly advanced. Here we assume that the student has had about 8 lectures worth of C++, covering

1. basic control structures,
2. simple functions including parameter passing by reference,
3. arrays through `std::vector`.

Based on this, the topics in table 3.2 can be taught in that order.

### 3.2.2 Project-based teaching

To an extent it is inevitable that students will do a number of exercises that are not connected to any earlier or later ones. However, to give some continuity, this book contains some programming projects that students gradually build towards.

The following are simple projects that have a sequence of exercises building on each other:

**Prime** Prime number testing, culminating in prime number sequence objects, and testing a corollary of the Goldbach conjecture. Chapter 46.

**Geom** Geometry related concepts; this is mostly an exercise in object-oriented programming. Chapter 47.

**Root** Numerical zero-finding methods. Chapter 48.

The following project are halfway serious research projects:

**Infect** The spreading of infectious diseases; these are exercises in object-oriented design. Students can explore various real-life scenarios. Chapter 50.

**Pagerank** The Google Pagerank algorithm. Students program a simulated internet, and explore pageranking, including 'search engine optimization'. This exercise uses lots of pointers. Chapter 51.

**Gerrymandering** Redistricting through dynamic programming. Chapter 52.

| | | Exercises | | | | |
|---|---|---|---|---|---|---|
| lesson | Topic | in-class | homework | prime | geom | infect |
| 1 | Statements and expressions | 4.4 | 4.9 (T) | 46.1 | | |
| 2 | Conditionals | 5.2 (S), 5.3 | 5.4 (T) | 46.2 | | |
| 3 | Looping | 6.5 (S), 6.4 | 6.6 (T) | 46.3, 46.4 | | |
| 4 | continue | | | | | |
| 5 | Functions | 7.1 (S), 7.2 | 7.6 | 46.6, 46.7 (T) | | |
| 6 | continue | 7.11 | | | 47.1 | |
| 7 | I/O | | 12.1 | | | |
| 8 | Objects | | | 9.8 (S), 46.8 (T), 46.10 | 47.3 | 50.1 |
| 9 | continue | | | | | |
| 10 | has-a relation | | | | 47.9 (T), 47.10, 47.1, 47.12 | 50.2 |
| 11 | inheritance | | | | 47.14, 47.15 | |
| 12 | Vectors | 10.1 (S), 10.2 | 10.21 | 46.15 | | 50.2 and further |
| 13 | continue | | | | | |
| 14 | Strings | | | | | |

Table 3.1: Two-month lesson plan for C++; the annotation '(S)' indicates that a skeleton code is available; '(T)' indicates that a tester script is available.

| | | | | Exercises | |
|---|---|---|---|---|---|
| lesson | Topic | Book | Prerequisite | In-class | Homework |
| 1 | Welcome, accounts | | | | Essay, `coe_history` |
| 2,3 | Unix, compilation, check prior knowledge | | | | Collatz: 6.13; swap: 7.5; vectors: 10.19, `coe_catchup` |
| 4, 5 | Test-driven development | 68 | Separate compilation 19 | 68.1 (S), 48.1–48.5 | 48.7 `coe_bisection` |
| 6, 7 | Objects | 9 | | 9.3 (S), 9.8 (S) | 46.8 `coe_primes` |
| 8 | Class inclusion | 9.2 | | 47.9, 47.12 | |
| 9 | Inheritance | 9.3 | | 47.14 | 46.9, 46.10 `coe_goldbach` |
| 10 | Vectors | 10 | | 10.1 (S), 10.8, 10.9 | |
| 11 | Vectors in classes | 10.6 | | 10.14 (S) | 10.21 `coe_pascal` |
| 12, 13 | Lambda functions | 13 | | 48.9 (S), 48.10 | 48.11, 48.12 `coe_newton` |
| 14,15 | STL, variant, optional | 24.5.2 | | 46.14 | Eight queens: 49 |
| 16 | Pointers | 16, 66.1.2 | | | 66.3–66.6 |
| 17 | C pointers | 17 | | | |
| 18 | libraries and cmake | | | | |
| | `cxxopts` | 63.1 | | 63.2 | |
| | `fmt` | | | | |
| | `random` | 24.6 | | | |
| | Exceptions formal approaches | 23 | | | |

Table 3.2: Advanced lessons for C++; the annotation '(S)' indicates that a skeleton code is available; '(T)' indicates that a tester script is available.

| lesson | Topic | Book | Slides | in-class | homework |
|---|---|---|---|---|---|
| 1 | Statements and expressions | 31 | 4F | | |
| | Conditionals | 32 | 5F | | |
| 2 | Looping | 33 | 6F | | 33.1 |
| | Functions | 34 | 7F | | |
| 3 | I/O | 42 | 8F | | |
| | Arrays | 40 | 14F | 40.1, 40.3, 40.5 | |
| 4 | Objects | 39 | 10F | 39.2 | |
| | Modules | 38 | 9F (?) | | 38.2 |
| 5 | Pointers | 41 | 15F | | |

Table 3.3: Accelerated lesson plan for Fortran; the annotation '(S)' indicates that a skeleton code is available; '(T)' indicates that a tester script is available.

**Scheduling** Exploration of concepts related to the Multiple Traveling Salesman Problem (MTSP), modeling *Amazon Prime*. Chapter 53.

**Lapack** Exploration of high performance linear algebra. Chapter 54.

Rather than including the project exercises in the didactic sections, each section of these projects list the prerequisite basic sections.

The project assignments give a fairly detailed step-by-step suggested approach. This acknowledges the theory of Cognitive Load [10].

Our projects are very much computation-based. A more GUI-like approach to project-based teaching is described in [7].

### 3.2.3 Choice: Fortran or advanced topics

After two months of grounding in OOP programming in C++, the Fortran lectures and exercises reprise this sequence, letting the students do the same exercises in Fortran that they did in C++. However, array mechanisms in Fortran warrant a separate lecture.

## 3.3 Scaffolding

As much as possible, when introducing a new topic we try to present a working code, with the first exercise being a modification of this code. At the root level of the repository is a directory `skeletons`, containing the example programs. In the above tables, exercises for which a skeleton code is given are marked with '(S)'.

## 3.4 Grading guide

Here are some general guidelines on things that should count as negatives when grading. These are all style points: a code using them may very well give the right answer, but they go against what is

commonly considered good or clean code.

The general guiding principle should be:

> Code is primarily for humans to read, only secondarily for computers to execute. This means that in addition to being correct, the code has to convince the reader that the result is correct.

As a corollary:

> Code should only be optimized when it is correct. Clever tricks detract from readability, and should only be applied when demonstrably needed.

### 3.4.1 Code layout and naming

Code should use proper indentation. Incorrect indentation deceives the reader.

Non-obvious code segments should be commented, but proper naming of variables and functions goes a long way towards making this less urgent.

### 3.4.2 Basic elements

- Variables should have descriptive names. For instance, `count` is not descriptive: half of all integers are used for counting. Count of what?
- Do not use global variables:

```
int i;
int main() {
  cin >> i;
}
```

### 3.4.3 Looping

The loop variable should be declared locally, in the loop header, if there is no overwhelming reason for declaring it global.

Local declaration:

```
for ( int i=0; i<N; i++) {
  // something with i
}
```

global:

```
int i;
for ( i=0; i<N; i++) {
  // something with i
  if (something) break;
}
// look at i to see where the break
    was
```

Use range-based loops if the index is not strictly needed.

### 3.4.4 Functions

There is no preference whether to define a function entirely before the main, or after the main with only its declaration before.

Defined before:

```
bool f( double x ) {
  return x>0;
}
int main() {
  cout << f(5.1);
}
```

Only declaration given before:

```
bool f(double x);
int main() {
  cout << f(5.1);
}
bool f( double x ) {
  return x>0;
}
```

Only use C++-style parameter passing by value or reference: do not use C-style `int*` parameters and such.

### 3.4.5 Scope

Variables should be declared in the most local scope that uses them. Declaring all variables at the start of a subprogram is needed in Fortran, but is to be discouraged in C++

### 3.4.6 Classes

- Do not declare data members `public`.
- Only write accessor functions if they are really needed.
- Make sure method names are descriptive of what the method does.
- The keyword `this` is hardly ever needed. This usually means the student has been looking at stackoverflow too much.

### 3.4.7 Arrays vs vectors

- Do not use old-style C arrays.

  ```
  int indexes[5];
  ```

- Certainly never use *malloc* or `new`.
- Iterator (*begin*, *erase*) are seldom used in this course, and should only be used if strictly needed, for instance with 'algorithms'.

### 3.4.8 Strings

Do not use old-style C strings:

```
char *words = "and another thing";
```

### 3.4.9 Other remarks

#### *3.4.9.1 Uninitialized variables*

Uninitialized variables can lead to undefined or indeterminate behavior. Bugs, in other words.

```
int i_solution;
int j_solution;
bool found_solution;
for(int i = 0; i < 10; i++){
  for(int j = 0; j < 10; j++){
    if(i*j > n){
      i_solution = i;
      j_solution = j;
      found_solution = true;
      break;
    }

  }
  if(found_solution){break;}
}
cout << i_solution << "," << j_solution << endl;
```

```
%% icpc -o missinginit missinginit.cxx && echo 40 | ./missinginit
0, -917009232
6, 7
```

This whole issue can be side stepped if the compiler or runtime system can detect this issue. Code structure will often prevent detection, but runtime detection is always possible, in principle.

For example, the Intel compiler can install a run-time check:

```
%% icpc -check=uninit -o missinginit missinginit.cxx && echo 40 | ./missin

Run-Time Check Failure: The variable 'found_solution' is being used in mis

Aborted (core dumped)
```

#### 3.4.9.2 Clearing objects and vectors

The following idiom is found regularly:

```
vector<int> testvector;
for ( /* possibilities */ ) {
  testvector.push_back( something );
  if ( testvector.sometest() )
    // remember this as the best
  testvector.clear();
}
```

A similar idiom occurs with classes, which students endow with a `reset()` method.

This is more elegantly done by declaring the testvector inside the loop: the reset is then automatically taken care of.

The general principle here is that entities need to be declared as local as possible.

# PART II

# C++

# Chapter 4

# Basic elements of C++

## 4.1 From the ground up: Compiling C++

In this chapter and the next you are going to learn the C++ language. But first we need some externalia: where do you get a program and how do you handle it?

In programming you have two kinds of files:

- *source files*, which are understandable to you, and which you create with an editor such as `vi` or `emacs`; and
- *binary files*, which are understandable to the computer, and unreadable to you.

Your source files are translated to binary by a *compiler*, which 'compiles' your source file.

Let's look at an example:

```
icpc -o myprogram myprogram.cxx
```

This means:

- you have a source code file `myprogram.cxx`;
- and you want an executable file as output called `myprogram`,
- and your compiler is the Intel compiler *icpc*. (If you want to use the C++ compiler of the *GNU* project you specify *g++*; the compiler of the clang project is *clang++*.)

Let's do an example.

**Exercise 4.1.** Make a file `zero.cc` with the following lines:

```
#include <iostream>
using std::cout;

int main() {
  return 0;
}
```

and compile it. Intel compiler:

```
icpc -o zeroprogram zero.cc
```

Run this program (it gives no output):

```
./zeroprogram
```

In the above program:

1. The first two lines are magic, for now. Always include them. Ok, if you want to know: the `#include` line is a *pre-processor* (chapter 21) directive; it includes a *header* file into your program that makes library functionality available.
2. The `main` line indicates where the program starts; between its opening and closing brace will be the *program statements*.
3. The `return` statement indicates successful completion of your program. (If you wonder about the details of this, see section 4.6.1.)

If you followed the above instructions, and as you may have guessed, you saw that this program produces absolutely no output when you run it.

If you do `ls` in your current directory, you see that you now have two files: the source file `zero.cc` and the executable `zeroprogram`. There is some freedom in how you choose these names.

File names can have extensions: the part after the dot. (The part before the dot is completely up to you.)

- `program.cxx` or `program.cc` are typical extensions for C++ sources.
- `program.cpp` is also used, but there is a possible confusion with 'C PreProcessor'.
- Using `program` without extension usually indicates an *executable*. (What happens if you leave the `-o myprogram` part off the compile line?)

Let's make the program do something: display a 'hello world' message on your screen. For now, just copy this line; the details of what it all means will come later.

**Exercise 4.2.** Add this line:

```
cout << "Hello world!" << '\n';
```

(copying from the pdf file is dangerous! please type it yourself)

Compile and run again.

(Did you indent the 'hello world' line? Did your editor help you with the indentation?)

Test your knowledge of the file types involved in programming!

**Review** 4.1. True or false?

1. The programmer only writes source files, no binaries.
2. The computer only executes binary files, no human-readable files.

### 4.1.1 A quick word about unix commands

The compile line

```
g++ -o myprogram myprogram.cxx
```

can be thought of as consisting of three parts:

- The command `g++` that starts the line and determines what is going to happen;
- The argument `myprogram.cxx` that ends the line is the main thing that the command works on; and
- The option/value pair `-o myprogram`. Most Unix commands have various options that are, as the name indicates, optional. For instance you can tell the compiler to try very hard to make a fast program:

  ```
  g++ -O3 -o myprogram myprogram.cxx
  ```

  Options can appear in any order, so this last command is equivalent to

  ```
  g++ -o myprogram -O3 myprogram.cxx
  ```

Be careful not to mix up argument and option. If you type

```
g++ -o myprogram.cxx myprogram
```

then Unix will reason: 'myprogram.cxx is the output, so if that file already exists (which, yes, it does) let's just empty it before we do anything else'. And you have just lost your program. Good thing that editors like *emacs* keep a backup copy of your file.

### 4.1.2 C++ is a moving target

The C++ language has gone through a number of standards. (This is described in some detail in section 27.6.) In this course we focus on a fairly recent standard: C++17. Unfortunately, your compiler will assume an earlier standard by default, so constructs taught here may be marked as ungrammatical.

You can tell your compiler to use the modern standard:

```
icpc -std=c++17 [other options]
```

but to save yourself a lot of typing, you can define

```
alias icpc='icpc -std=c++17'
```

in your shell startup files. On the class `isp` machine this alias has been defined by default.

## 4.2 Statements

Each programming language has its own (very precise!) rules for what can go in a source file. Globally we can say that a program contains instructions for the computer to execute, and these instructions take the form of a bunch of 'statements'. Here are some of the rules on statements; you will learn them in more detail as you go through this book.

- A program contains statements, each terminated by a semicolon.
- 'Curly braces' can enclose multiple statements.
- A statement can correspond to some action when the program is executed.
- Some statements are definitions, of data or of possible actions.

- Comments are 'Note to self', short:

```
cout << "Hello world" << "\n"; // say hi!
```

  and arbitrary:

```
cout << /* we are now going
            to say hello
         */ "Hello!" << /* with newline: */ "\n";
```

In the examples so far you see output statements terminated as:

```
cout << something << "\n";
```

Sometimes you will also see:

```
// at the top of the program:
using std::endl;

// in the source:
cout << something << endl;
```

The distinction will not be important to you for now; see the discussion in section 12.4 if you're curious.

**Exercise 4.3.** Take the 'hello world' program you wrote above, and duplicate the hello-line. Compile and run.

Does it make a difference whether you have the two hellos on the same line or on different lines?

Experiment with other changes to the layout of your source. Find at least one change that leads to a compiler error. Can you relate the message to the error?

Your program source can have several types of errors. Distinguishing by when you notice them, we roughly distinguish them as follows. (Later there will be a whole chapter on errors: 23.)

1. *Syntax* or *compile-time* errors: these arise if what you write is not according to the language specification. The compiler catches these errors, and it refuses to produce a *binary file*.
2. *Run-time* errors: these arise if your code is syntactically correct, and the compiler has produced an executable, but the program does not behave the way you intended or foresaw. Examples are divide-by-zero or indexing outside the bounds of an array.

**Review** 4.2. True or false?

- If your program compiles correctly, it is correct.
- If you run your program and you get the right output, it is correct.

In the program you just wrote, the string your displayed was completely up to you. Other elements, such as the `cout` keyword, are fixed parts of the language. Most programs contains them.

You see that certain parts of your program are inviolable:

- There are *keywords* such as `return` or `cout`; you can not change their definition.
- Curly braces and parentheses need to be matched.

- There has to be a `main` keyword.
- The `iostream` and `std` are usually needed.

### 4.2.1 Language vs library and about: using

The examples above had a line

```
#include <iostream>
```

which allowed you to write `std::cout` in your program for output. The `iostream` is a *header*, and it add *standard library* functionality to the base language.

Functionality such as `cout` can be used in various ways:

You can spell it out as `std::cout`:

```
#include <iostream>
int main() {
  std::cout "hello\n";
  return 0;
  }
```

You can add a `using` statement:

```
#include <iostream>
using std::cout;
int main() {
  cout "hello\n";
  return 0;
  }
```

Instead of having separate `using` statements for each library function, you could also use a single line

```
using namespace std;
```

in your program. While it is common to find this in examples online, it is frowned upon; see section 20.3 for a discussion.

**Exercise 4.4.** Experiment with the `cout` statement. Replace the string by a number or a mathematical expression. Can you guess how to print more than one thing, for instance:

- the string `One third is`, and
- the result of the computation $1/3$,

with the same `cout` statement? Do you get anything unexpected?

## 4.3 Variables

A program could not do much without storing data: input data, temporary data for intermediate results, and final results. Data is stored in *variables*, which have

- a name, so that you can refer to them,
- a *datatype*, and
- a value.

Think of a variable as a labeled placed in memory.

- The variable is defined in a *variable declaration*,
- which can include an *variable initialization*.

- After a variable is defined, and given a value, it can be used,
- or given a (new) value in a *variable assignment*.

```
int i,j; // declaration
i = 5;   // set a value
i = 6;   // set a new value
j = i+1; // use the value of i
i = 8; // change the value of i
       // but this doesn't affect j:
       // it is still 7.
```

### 4.3.1 Variable declarations

A variable is defined, in a *variable declaration*. This associates its name and its type, and possibly an initial value; see section 4.3.2.

Let's first talk about what a variable name can be.

- A variable name has to start with a letter;
- a name can contains letters and digits, but not most special characters, except for the underscore.
- For letters it matters whether you use upper or lowercase: the language is *case sensitive*.
- Words such as `main` or `return` are *reserved words*.
- Usually `i` and `j` are not the best variable names: use `row` and `column`, or other meaningful names, instead.
- (Strictly speaking, a variable name can start with an underscore, but it's better not to do that.)

Next, a *variable declaration* states the type and the name of a variable. If you have multiple variables of the same type, you can combine the declarations.

A variable declaration establishes the name and the type of a variable:

```
int n;
float x;
int n1,n2;
double re_part,im_part;
```

You can not redeclare a variable like this:

```
int i=5;
cout << i;
float i=1.3;
cout << i;
```

but the rules for what **is** allowed are a little harder to state. You'll see that later in chapter 8.

Declarations can go pretty much anywhere in your program, but they need to come before the first use of the variable.

Note: it is legal to define a variable before the main program but such *global variables* are usually not a

good idea. Please only declare variables *inside* main (or inside a function et cetera).

**Review** 4.3. Which of the following are legal variable names?

1. `mainprogram`
2. `main`
3. `Main`
4. `1forall`
5. `one4all`
6. `one_for_all`
7. `onefor{all}`

### 4.3.2 Initialization

It is a possible to give a variable a value right when it's created. This is known as *initialization* and it's different from creating the variable and later assigning to it (section 4.3.3).

There are (at least) two ways of initializing a variable

```
int i = 5;
int j{6};
```

Note that writing

```
int i;
i = 7;
```

is not an initialization: it's a declaration followed by an assignment.

If you declare a variable but not initialize, you can not count on its value being anything, in particular not zero. Such implicit initialization is often omitted for performance reasons.

### 4.3.3 Assignments

Setting a variable

```
i = 5;
```

means storing a value in the memory location. It is not the same as defining a mathematical equality

let $i = 5$.

Once you have declared a variable, you need to establish a value. This is done in an *assignment* statement. After the above declarations, the following are legitimate assignments:

```
n = 3;
x = 1.5;
n1 = 7; n2 = n1 * 3;
```

These are not math equations: variable on the lhs gets the value of the rhs.

You see that you can assign both a simple value or an *expression*.

You can set the value of a variable multiple times.

Update:

```
x = y-2; y = 2*x + 5;

x = x+2; y = y/3;
// can be written as
x += 2; y /= 3;
```

Integer add/subtract one:

```
i++; j--;  /* same as: */ i=i+1; j=j-1;
```

**Exercise 4.5.** Which of the following are legal? If they are, what is their meaning?

1. `n = n;`
2. `n = 2n;`
3. `n = n2;`
4. `n = 2*k;`
5. `n/2 = k;`
6. `n /= k;`

There are various levels of programming errors. The following program uses the variable `i` without having given it a value.

**Exercise 4.6.**

```
#include <iostream>
using std::cout;
int main() {
  int i;
  int j = i+1;
  cout << j << "\n";
  return 0;
}
```

What happens?

1. Compiler error
2. Output: `1`
3. Output is undefined
4. Error message during running the program.

### 4.3.4 Datatypes

You have seen a couple of datatypes that variables can have. We'll go into the issue of datatypes into a little more detail.

Variables come in different types;

- We call a variable of type `int`, `float`, `double` a *numerical variable*.
- *Complex numbers* will be discussed later.
- For characters: `char`. Strings are complicated; see later.
- Truth values: `bool`
- You can make your own types. Later.

For complex numbers see section 24.1. For strings see chapter 11.

#### 4.3.4.1 Integers

Mathematically, integers are a special case of real numbers. In a computer, integers are stored very differently from real numbers – or technically, floating point numbers.

You might think that C++ integers are stored as binary number with a sign bit, but the truth is more subtle. For now, know that within a certain range, approximately symmetric around zero, all integer values can be represented.

**Exercise 4.7.** These days, the default amount of storage for an `int` is 32 bits. After one bit is used for the sign, that leave 31 bits for the digits. What is the representable integer range?

The integer type in C++ is `int`:

```
int my_integer;
my_integer = 5;
cout << my_integer << "\n";
```

For more integer types, see chapter 25; if you're wondering how large integers and such can get, see section 25.4 in particular.

Integer constants can be represented on several *bases*.

Integers are normally written in decimal, and stored in 32 bits. If you need something else:

```
int d = 42;
int o = 052; // start with zero
int x = 0x2a;
int X = 0X2A;
int b = 0b101010;
long ell = 42L;
```

Binary numbers are new to C++17.

#### 4.3.4.2 Floating point numbers

*Floating point number* is the computer science-y name for scientific notation: a number written like

$$+6 \cdot 022 \times 10^{23}$$

with:

- an optional sign;

- an integral part;
- a decimal point, or more generally *radix point* in other number bases;
- a fractional part, also known as *mantissa* or *significand*;
- and an *exponent part*: base to some power.

Floating point numbers are by default of type `double`, which standard for 'double precision'. Double of what? We will discuss that in section 25.4. For now, let's discuss only the matter of how they are represented.

Without further specification, a floating point literal is of type `double`:

```
1.5
1.5e+5
```

Use a suffix `1.5f` for type `float` which stands for 'single precision':

```
1.5f
1.5e+5f
```

Use a suffix `1.5L` for `long double`: quadruple precision.

```
1.5L
1.5e+5L
```

There is a way to give a hexadecimal representation of floating points number, but this is complicated.

*4.3.4.2.1* **Other precisions** If you need numerical precision beyond double, *quadruple precision* is supported in C++23 through `std::float128_t`.

*4.3.4.2.2* **Limitations** Floating point numbers are also referred to as 'real numbers' (in fact, in the Fortran language they are defined with the keyword `Real`), but this is sloppy wording. Since only a finite number of bits/digits is available, only terminating fractions are representable. For instance, since computer numbers are binary, $1/2$ is representable but $1/3$ is not.

**Exercise 4.8.** Can you think of a way that non-terminating fractions, including such numbers such as $\sqrt{2}$, would still be representable?

- You can assign variables of one type to another, but this may lead to truncation (assigning a floating point number to an integer) or unexpected bits (assigning a single precision floating point number do a double precision).

Floating points numbers do not behave like mathematical numbers; for extensive discussion, see HPC book [9], section 3.3 and later.

Floating point arithmetic is full of pitfalls.

- Don't count on `3*(1./3)` being exactly 1.
- Not even associative.

Complex numbers exist, see section 24.1.

#### 4.3.4.3 *Boolean values*

So far you have seen integer and real variables. There are also *boolean values* which represent truth values. There are only two values: `true` and `false`.

```
bool found{false};
found = true;
```

## 4.4 Input/Output, or I/O as we say

A program typically produces output. For now we will only display output on the screen, but output to file is possible too. Regarding input, sometimes a program has all information for its computations, but it is also possible to base the computation on user input.

Terminal (console) output with `cout`:

```
float x = 5;
cout << "Here is the root: " << sqrt(x) << '\n';
```

Note the newline character.
Alternatively: `std::endl`, less efficient.

You can get input from the keyboard with `cin`, which accepts arbitrary strings, as long they don't have spaces.

```
string name; int age;
cout << "Your name?\n";
cin >> name;
cout << "age?\n";
cin >> age;
cout << age << " is a nice age, "
     << name << '\n';
```

```
> ./cin
Your name?
Victor
age?
18
18 is a nice age, Victor
> ./cin
Your name?
THX 1138
age?
1138 is a nice age, THX
```

For more flexible input, see section 12.5.

For fine-grained control over the output, see section 12.1. For other I/O related matters, such as file I/O, see chapter 12.

## 4.5 Expressions

The most basic step in computing is to form expressions such as sums, products, logical conjunctions, string concatenations from variables and constants.

Let's start by discussing constants: numbers, truth values, strings.

### 4.5.1 Numerical expressions

Expressions in programming languages for the most part look the way you would expect them to.

- Mathematical operators: `+ - / ` and `*` for multiplication.
- Integer modulus: `5%2`
- You can use parentheses: `5*(x+y)`. Use parentheses if you're not sure about the precedence rules for operators.
- C++ does not have a power operator (Fortran does): 'Power' and various mathematical functions are realized through library calls.

Math functions are in *cmath*:

```
#include <cmath>
.....
x = pow(3,.5);
```

For squaring, usually better to write `x*x` than `pow(x,2)`.

**Exercise 4.9.** Write a program that :

- displays the message `Type a number`,
- accepts an integer number from you (use **cin**),
- makes another variable that is three times that integer plus one,
- and then prints out the second variable.

Fine points of integers and integer expression are discussed in section 25.1.

### 4.5.2 Truth values

In addition to numerical types, there are truth values, **true** and **false**, with all the usual logical operators defined on them.

- Relational operators: `== != < > <= >=`
- Boolean operators: `not, and, or` (oldstyle: `! && ||`);

### 4.5.3 Type conversions

Since a variable has one type, and will always be of that type, you may wonder what happens with

```
float x = 1.5;
int i;
i = x;
```

or

```
int i = 6;
float x;
x = i;
```

- Assigning a floating point value to an integer truncates the latter.
- Assigning an integer to a floating point variable fills it up with zeros after the decimal point.

**Exercise 4.10.** Try out the following:

- What happens when you assign a positive floating point value to an integer variable? What happens when you assign a negative floating point value to an integer variable? Does your compiler give warnings? Is there a way you can trick the compiler into not understanding what you are doing?
- What happens when you assign a `float` to a `double`? Try various numbers for the original float. Print out the result, and if they look the same, see if the difference is actually zero.

The rules for type conversion in expressions are not entirely logical. Consider

```
float x; int i=5,j=2;
x = i/j;
```

This will give `2` and not `2.5`, because `i/j` is an integer expression and is therefore completely evaluated as such, giving `2` after truncation. The fact that it is ultimately assigned to a floating point variable does not cause it to be evaluated as a computation on floats.

You can force the expression to be computed in floating point numbers by writing

```
x = (1.*i)/j;
```

or any other mechanism that forces a conversion, without changing the result. Another mechanism is the *cast*; this will be discussed in section 27.2.

**Exercise 4.11.** Write a program that asks for two integer numbers `n1,n2`.

- Assign the integer ratio $n_1/n_2$ to an integer variable.
- Can you use this variable to compute the modulus

$$n_1 \mod n_2$$

(without using the `%` modulus operator!)
Print out the value you get.
- Also print out the result from using the modulus operator:`%`.
- Investigate the behavior of your program for negative inputs. Do you get what you were expecting?

**Exercise 4.12.** Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

$$C = (F - 32) \cdot 5/9, \qquad F = 9/5\, C + 32$$

Check your program for the freezing and boiling point of water.
(Do you know the temperature where Celsius and Fahrenheit are the same?)

Can you use Unix pipes to make one accept the output of the other?

**Review** 4.4. True of false?

1. Within a certain range, all integers are available as values of an integer variable.
2. Within a certain range, all real numbers are available as values of a float variable.
3. `5(7+2)` is equivalent to `45`.
4. `1--1` is equivalent to zero.
5. `int i = 5/3.;` The variable `i` is 2.
6. `float x = 2/3;` The variable `x` is approximately `0.6667`.

### 4.5.4 Characters and strings

In this course we are mostly concerned with numerical data, but string and character data can be useful for purposes of output.

#### *4.5.4.1 Strings*

Strings, that is, strings of characters, are not a C++ built-in datatype. Thus, they take some extra setup to use. See chapter 11 for a full discussion.

For characters there is the **char** data type, and for strings *string*, If you want to use strings:

- Add the following at the top of your file:

```
#include <string>
using std::string;
```

- Declare string variables as

```
string name;
```

- And you can now **cin** and **cout** them.

A character is enclosed in single quotes:

```
'x'
```

while a general string is enclosed in double quotes:

```
"The quick brown fox"
```

**Exercise 4.13.** Write a program that asks for the user's first name, uses **cin** to read that, and prints something like `Hello, Susan!` in response.

What happens if you enter first and last name?

## 4.6 Advanced topics

### 4.6.1 The main program and the return statement

The *main* program has to be of type **int**; however, many compilers tolerate deviations from this, for instance accepting **void**, which is not language standard.

The arguments to main can be:

```
int main()
int main( int argc,char* argv[] )
int main( int argc,char **argv )
```

The returned `int` can be specified several ways:

- If no `return` statement is given, implicitly `return 0` is performed. (This is also true in C99.)
- You can explicitly pass an integer to the operating system, which can then be queried in the *shell* as a *return code*:

**Code:**

```
int main() {
  return 1;
}
```

**Output**
**[basic] return:**

```
./return ; \
        if [ $? -ne 0 ] ;
    then \
            echo "Program
    failed" ; \
        fi
Program failed
```

- For cleanliness, you can use the values `EXIT_SUCCESS` and `EXIT_FAILURE` which are defined in `cstdlib.h`.
- You can also use the `exit` function:

```
void exit(int);
```

### 4.6.2 Identifier names

Variable names, or more correctly: *identifier*s, have to start with a non-digit. To be precise, this can be

- a Latin letter, which is the most common case;
- an *underscore*, which is the convention for private members of a class, and other 'internal' names; or
- *Unicode* characters of class `XID_Start`.

Any following character can be Unicode characters of class `XID_Continue`.

On the topic of underscores, a leading *double underscore* should not be used since such names are reserved for the compiler.

**Remark 1** *General Unicode characters became allowed in C++23, but this convention was then applied retro-actively to earlier standards.*

## 4.7 C differences

### 4.7.1 Boolean

Traditionally, C did not have a type for boolean values; instead `int` and `short` was used, where zero was false, and any nonzero value true. In C99 the type `_Bool` was introduced. This only serves legibility: there are no true/false constants, and variables of type `_Bool` still have to be treated as integers in `printf`.

```
_Bool tf = 1;
printf("True: %d\n",tf);
```

The *stdbool.h* defines **bool**, **true**, and **false** as aliases.

## 4.8 Review questions

**Review** 4.5. What is the output of:

```
int m=32, n=17;
cout << n%m << "\n";
```

**Review** 4.6. Given

```
int n;
```

give an expression that uses elementary mathematical operators to compute n-cubed: $n^3$. Do you get the correct result for all $n$? Explain.

How many elementary operations does the computer perform to compute this result?

Can you now compute $n^6$, minimizing the number of operations the computer performs?

# Chapter 5

# Conditionals

A program consisting of just a list of assignment and expressions would not be terribly versatile. At least you want to be able to say 'if some condition, do one computation, otherwise compute something else', or: 'until some test is true, iterate the following computations'. The mechanism for testing and choosing an action accordingly is called a *conditional*. (Iterating is discussed in chapter 6.)

## 5.1 Conditionals

Here are some forms a conditional can take.

A single statement, executed if the test is true:

```
if (x<0)
  x = -x;
```

Single statement in the true branch, and likewise a single statement in the false branch:

```
if (x>=0)
  x = 1;
else
  x = -1;
```

Both in the true and the false branch multiple statements are allowed, if you enclose them in curly braces:

```
if (x<0) {
  x = 2*x; y = y/2;
} else {
  x = 3*x; y = y/3;
}
```

You can chain conditionals by extending the `else` part. In this example the dots stand for omitted code:

```
if (x>0) {
  ....
} else if (x<0) {
  ....
} else {
  ....
}
```

Conditionals can also be nested:

```
if (x>0) {
  if (y>0) {
    ....
  } else {
    ....
  }
} else {
  ....
}
```

- In that last example the outer curly brackets in the true branch are optional. But it's safer to use them anyway.
- When you start nesting constructs, use indentation to make it clear which line is at which level. A good editor helps you with that.

**Exercise 5.1.** For what values of $x$ will the left code print 'b'?
For what values of $x$ will the right code print 'b'?

```
float x = /* something */
if ( x > 1 ) {
   cout << "a" << endl;
   if ( x > 2 )
    cout << "b" << endl;
}
```

```
float x = /* something */
if ( x > 1 ) {
   cout << "a" << endl;
} else if ( x > 2 ) {
    cout << "b" << endl;
}
```

## 5.2 Operators

You have already seen arithmetic expressions; now we need to look at logical expressions: just what can be tested in a conditional. For the most part, logical expressions are intuitive. However, note that they can be chained only in certain ways:

```
bool x,y,z;
if ( x or y or z ) ; //good
int i,j,k;
if ( i < j < k ) ; // WRONG
```

Here are the most common *logic operator*s and *comparison operator*s:

| Operator | meaning | example |
|---|---|---|
| == | equals | x==y-1 |
| != | not equals | x*x!=5 |
| > | greater | y>x-1 |
| >= | greater or equal | sqrt(y)>=7 |
| <,<= | less, less equal | |
| &&,\|\| | and, or | x<1 && x>0 |
| and,or | and, or | x<1 and x>0 |
| ! | not | !( x>1 && x<2 ) |
| not | | not ( x>1 and x<2 ) |

*Precedence* rules of operators are common sense. When in doubt, use parentheses.

**Exercise 5.2.** The following code claims to detect if an integer has more than 2 digits.

**Code:**

```
int i;
cin >> i;
if ( i>100 )
  cout << "That number " << i
       << " had more than 2 digits"
       << '\n';
```

**Output**
**[basic] if:**

```
... with 50 as input ....
... with 150 as input ....
That number 150 had more than 2
    digits
```

Fix the small error in this code. Also add an 'else' part that prints if a number is negative.

*You can base this off the file `if.cxx` in the repository*

**Exercise 5.3.** Read in an integer. If it is even, print 'even', otherwise print 'odd':

```
if ( /* your test here */ )
  cout << "even" << endl;
else
  cout << "odd" << endl;
```

Then, rewrite your test so that the true branch corresponds to the odd case.

In exercise 5.3 it didn't matter whether you used the test for even or for odd, but sometimes it does make a difference how you arrange complicated conditionals. In the following exercise, think about how to arrange the tests. There is more than one way.

**Exercise 5.4.** Read in a positive integer. If it's a multiple of three print 'Fizz!'; if it's a multiple of five print 'Buzz!'. It is a multiple of both three and five print 'Fizzbuzz!'. Otherwise print nothing.

Note:

- Capitalization.
- Exclamation mark.
- Your program should display at most one line of output.

### 5.2.1 Bitwise logic

Above we only considered the **and** and **or** logical operators, also spelled `&&` and `||`. There are also *bitwise operators*, that look confusingly similar to the latter notation:

**Code:**

```
int x=6,y=3;
cout << "6|3 = " << (x|y) << '\n';
cout << "6&3 = " << (x&y) << '\n';
```

**Output**
**[basic] bitor:**

```
6|3 = 7
6&3 = 2
```

To understand what's happening here, realize that

$$6 \equiv 110 \quad \text{and} \quad 3 \equiv 011$$

You will probably not use the bitwise operators often, but the following idiom is sometimes encountered:

```
const int
  STATE_1 = 1, STATE_2 = 1<<1, STATE_3 = 1<<2;
int state = /* stuff */;
if ( state & ( STATE_1 | STATE_3 ) )
  cout << "We are in state 1 or 3";
```

### 5.2.2 Review

**Review** 5.1. True or false?

- The tests `if (i>0)` and `if (0<i)` are equivalent.
- The test

```
if (i<0 && i>1)
  cout << "foo"
```

  prints `foo` if $i < 0$ and also if $i > 1$.
- The test

```
if (0<i<1)
  cout << "foo"
```

  prints `foo` if $i$ is between zero and one.

**Review** 5.2. Any comments on the following?

```
bool x;
// ... code with x ...
if ( x == true )
  // do something
```

## 5.3 Switch statement

If you have a number of cases corresponding to specific integer values, there is the `switch` statement.

Cases are executed consecutively until you 'break' out of the switch statement:

**Code:**

```
switch (n) {
case 1 :
case 2 :
  cout << "very small" << '\n';
  break;
case 3 :
  cout << "trinity" << '\n';
  break;
default :
  cout << "large" << '\n';
}
```

**Output**

**[basic] switch:**

```
for v in 1 2 3 4 5 ; do \
          echo $v | ./switch ; \
        done
very small
very small
trinity
large
large
```

**Exercise 5.5.** Suppose the variable `n` is a nonnegative integer. Write a `switch` statement that has the same effect as:

```
if (n<5)
  cout << "Small" << endl;
else
  cout << "Not small" << endl;
```

It is possible that the compiler generates more efficient code from a **`switch`** statement than from a conditional. Otherwise, there are no things you can do with a **`switch`** that you can not do with a conditional.

## 5.4 Scopes

The true and false branch of a conditional can consist of a single statement, or of a block in curly brackets. Such a block creates a *scope* where you can define local variables.

```
if ( something ) {
  int i;
  .... do something with i
}
// the variable 'i' has gone away.
```

See chapter 8 for more on scopes.

## 5.5 Advanced topics

### 5.5.1 Short-circuit evaluation

C++ logic operators have a feature called *short-circuit evaluation*: a logical operator stops evaluating in strict left-to-right order when the result is clear. For instance, in

```
clause1 and clause2
```

the second clause is not evaluated if the first one is **`false`**, because the truth value of this conjunction is already determined.

Likewise, in

```
clause1 or clause2
```

the second clause is not evaluated if the first one is **`true`**, because the value of the **`or`** conjunction is already clear.

This mechanism allows you to write

```
if ( x>=0 and sqrt(x)<10 ) { /* ... */ }
```

Without short-circuit evaluation the square root operator could be applied to negative numbers.

### 5.5.2 Ternary if

The true and false branch of a conditional contain whole statements. For example

```
if (foo)
  x = 5;
else
  y = 6;
```

But what about the case where the true and false branch assign to the same variable, but with a different expression? You can not write

Original code:

```
if (foo)
  x = 5;
else
  x = 6;
```

Not legal syntax for 'simplification':

```
x = if (foo) 5; else 6;
```

For this case there is the *ternary if*, which acts as if it's an expression itself, but chosen between two expressions. The previous assignment to $x$ then becomes:

```
x = foo ? 5 : 6;
```

Surprisingly, this expression can even be in the left-hand side:

```
foo ? x : y = 7;
```

### 5.5.3 Initializer

The C++17 standard introduced a new form of the `if` and `switch` statement: it is possible to have a single statement of declaration prior to the test. This is called the *initializer*.

**Code:**

```
if ( char c = getchar(); c!='a' )
  cout << "Not an a, but: " << c
       << '\n';
else
  cout << "That was an a!"
       << '\n';
```

**Output**
**[basic] ifinit:**

```
for c in d b a z ; do \
           echo $c |
    ./ifinit ; \
        done
Not an a, but: d
Not an a, but: b
That was an a!
Not an a, but: z
```

This is particularly elegant if the init statement is a declaration, because the declared variable is then local to the conditional. Previously one would have had to write

```
char c;
c = getchar();
if ( c!='a' ) /* ... */
```

with the variable defined outside of the scope of the conditional.

## 5.6 Review questions

**Review** 5.3. T/F: the following is a legal program:

```
#include <iostream>
int main() {
  if (true)
    int i = 1;
  else
    int i = 2;
  std::cout << i;
  return 0;
}
```

**Review** 5.4. T/F: the following are equivalent:

```
if (cond1)
  i = 1;
else if (cond2)
  i = 2;
else
  i = 3;
```

compare:

```
if (cond1)
  i = 1;
else {
  if (cond2)
    i = 2;
  else
    i = 3;
}
```

# Chapter 6

# Looping

There are many cases where you want to repeat an operation or series of operations:

- A time-dependent numerical simulation executes a fixed number of steps, or until some stopping test.
- Recurrences:

$$x_{i+1} = f(x_i).$$

- Inspect or change every element of a database table.

The C++ construct for such repetitions is known as a *loop*: a number of statements that get repeated. The two types of loop statement in C++ are:

- the *for loop* which is typically associated with a pre-determined number of repetitions, and with the repeated statements being based on a counter of some sort; and
- the *while loop*, where the statements are repeated indefinitely until some condition is satisfied.

However, the difference between the two is not clear-cut: in many cases you can use either.

We will now first consider the **for** loop; the **while** loop comes in section 6.3.

## 6.1 The 'for' loop

In the most common case, a for loop has a *loop counter*, ranging from some initial value to some final value. An example showing the syntax for this simple case is:

```
int sum_of_squares{0};
for (int var=low; var<upper; var++) {
  sum_of_squares += var*var;
}
cout << "The sum of squares from "
     << low << " to " << upper
     << " is " << sum_of_squares << endl;
```

The **for** line is called the *loop header*, and the statements between curly brackets the *loop body*. Each execution of the loop body is called an *iteration*.

**Exercise 6.1.** Read an integer value with `cin`, and print 'Hello world' that many times.

**Exercise 6.2.** Extend exercise 6.1: the input 17 should now give lines

```
Hello world 1
Hello world 2
....
Hello world 17
```

Can you do this both with the loop index starting at 0 and 1?
Also, let the numbers count down.

We will now investigate the components of a loop.

### 6.1.1 Loop variable

First of all, most `for` loops have a *loop variable* or *loop index*. The first expression in the parentheses is usually concerned with the initialization of this variable: it is executed once, before the loop iterations. If you declare a variable here, it becomes local to the loop, that is, it only exists in the expressions of the loop header, and during the loop iterations.

The loop variable is usually an integer:

```
for ( int index=0; index<max_index; index=index+1) {
  ...
}
```

But other types are allowed too:

```
for ( float x=0.0; x<10.0; x+=delta ) {
  ...
}
```

Beware the stopping test for non-integral variables!

Usually, the loop variable only has meaning inside the loop so it should only be defined there. You do this by defining it in the loop header:

```
for (int var=low; var<upper; var++) {
```

However, it can also be defined outside the loop:

```
int var;
for (var=low; var<upper; var++) {
```

but you should only do this if the variable is actually needed after the loop. You will see an example where this makes sense below in section 6.3.

### 6.1.2 Stopping test

Next there is a test, which needs to evaluate to a boolean expression. This test is often called a 'stopping test', but to be technically correct it is actually executed at the start of each iteration, including the first one, and it is really a 'loop while this is true' test.

**Code:**

```
cout << "before the loop" << '\n';
for (int i=5; i<4; i++)
  cout << "in iteration "
       << i << '\n';
cout << "after the loop" << '\n';
```

**Output**
**[basic] pretest:**

```
before the loop
after the loop
```

- If this boolean expression is true, do the next iteration.
- Done before the first iteration too!
- Test can be empty. This means no test is applied.

```
for ( int i=0; i<N; i++) {...}
for ( int i=0; ; i++ ) {...}
```

Usually, the combination of the initialization and the stopping test determines how many iterations are executed. If you want to perform $N$ iterations you can write

```
for (int iter=0; iter<N; iter++)
```

or

```
for (int iter=1; iter<=N; iter++)
```

The former is slightly more idiomatic to C++, but you should write whatever best fits the problem you are coding.

The stopping test doesn’t need to be an upper bound. Here is an example of a loop that counts down to a lower bound.

```
for (int var=high; var>=low; var--) { ... }
```

The stopping test can be omitted

```
for (int var=low; ; var++) { ... }
```

if the loops ends in some other way. You’ll see this later.

### 6.1.3 Increment

Finally, after each iteration we need to update the loop variable. Since this is typically adding one to the variable we can informally refer to this as the ‘increment’, but it can be a more general update.

Increment performed after each iteration. Most common:

- `i++` for a loop that counts forward;
- `i--` for a loop that counts backward;

Others:

- `i+=2` to cover only odd or even numbers, depending on where you started;
- `i*=10` to cover powers of ten.

Even optional:

```
for (int i=0; i<N; ) {
  // stuff
  if ( something ) i+=1; else i+=2;
}
```

This is how a loop is executed.

- The initialization is performed.
- At the start of each iteration, including the very first, the stopping test is performed. If the test is true, the iteration is performed with the current value of the loop variable(s).
- At the end of each iteration, the increment is performed.

  *C difference:* Declaring the loop variable in the loop header is also a modern addition to the C language. Use compiler flag `-std=c99`.

**Exercise 6.3.** Take this code:

```
int sum_of_squares{0};
for (int var=low; var<upper; var++) {
  sum_of_squares += var*var;
}
cout << "The sum of squares from "
     << low << " to " << upper
     << " is " << sum_of_squares << '\n';
```

and modify it to sum only the squares of every other number, starting at `low`.

Can you find a way to sum the squares of the even numbers $\geq$ `low`?

**Review** 6.1. For each of the following loop headers, how many times is the body executed? (You can assume that the body does not change the loop variable.)

```
for (int i=0; i<7; i++)

for (int i=0; i<=7; i++)

for (int i=0; i<0; i++)
```

**Review** 6.2. What is the last iteration executed?

```
    for (int i=1; i<=2; i=i+2)

    for (int i=1; i<=5; i*=2)

    for (int i=0; i<0; i--)

    for (int i=5; i>=0; i--)
```

```
for (int i=5; i>0; i--)
```

### 6.1.4 Loop body

The loop body can be a single statement:

```
int s{0};
for ( int i=0; i<N; i++)
  s += i;
```

or a block:

```
int s{0};
for ( int i=0; i<N; i++) {
  int t = i*i;
  s += t;
}
```

If it is a block, it is a scope inside which you can declare local variables.

## 6.2 Nested loops

Quite often, the loop body will contain another loop. For instance, you may want to iterate over all elements in a matrix. Both loops will have their own unique loop variable.

```
for (int row=0; row<m; row++)
  for (int col=0; col<n; col++)
    ...
```

This is called *loop nest*; the `row` loop is called the *outer loop* and the `col` loop the *inner loop*.

Traversing an index space (whether that corresponds to an array object or not) by `row,col` is called the *lexicographic ordering*.

**Exercise 6.4.** Write an $i, j$ loop nest that prints out all pairs with

$$1 \leq i, j \leq 10, \quad j \leq i.$$

Output one line for each `i` value.

Now write an $i, j$ loop that prints all pairs with

$$1 \leq i, j \leq 10, \quad |i - j| < 2,$$

again printing one line per `i` value. Food for thought: this exercise is definitely easiest with a conditional in the inner loop, but can you do it without?

The mere fact that you need to traverse a rectangular range of $i, j$ indices, does not mean that you have to write a lexicographically indexed loop. Figure 6.1 illustrates that you can look at the $i, j$ indices by row/column or by diagonal. Just like rows and columns being defined as $i = \text{constant}$ and $j = \text{constant}$ respectively, a diagonal is defined by $i + j = \text{constant}$.

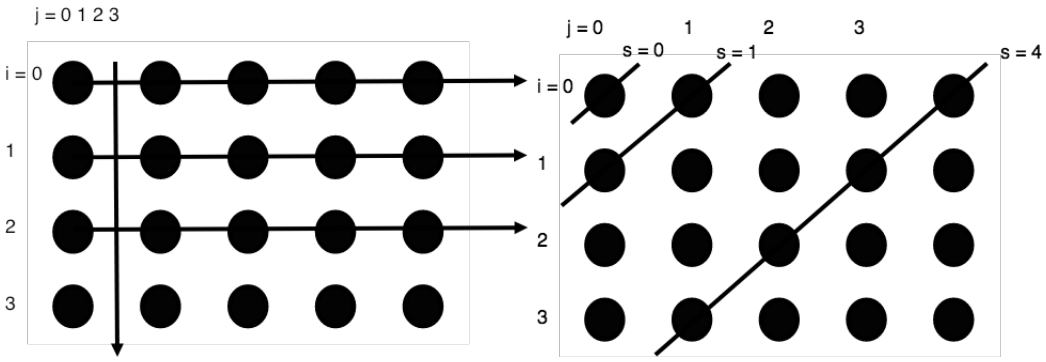



Figure 6.1: Lexicographic and diagonal ordering of an index set

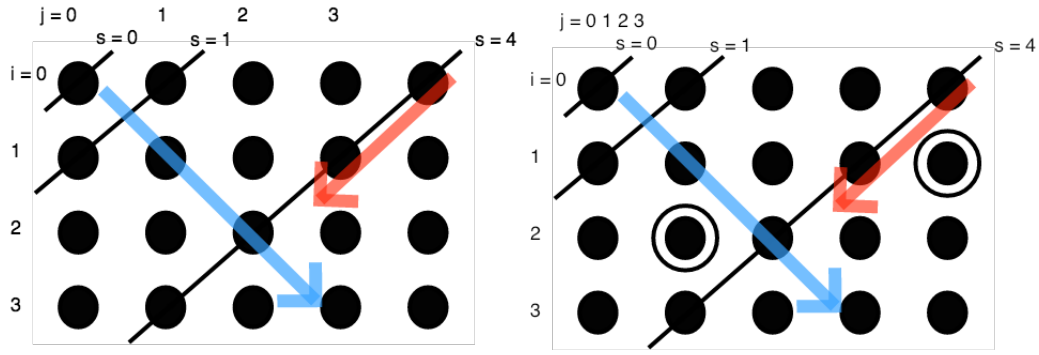



Figure 6.2: Illustration of the second part of exercise 6.6

.

## 6.3 Looping until

The basic for loop looks pretty deterministic: a loop variable ranges through a more-or-less prescribed set of values. This is appropriate for looping over the elements of an array, but not if you are coding some process that needs to run until some dynamically determined condition is satisfied. In this section you will see some ways of coding such cases.

First of all, the stopping test in the 'for' loop is optional, so you can write an indefinite loop as:

```
for (int var=low; ; var=var+1) { ... }
```

How do you end such a loop? For that you use the **break** statement. If the execution encounters this statement, it will continue with the first statement after the loop.

```
for (int var=low; ; var=var+1) {
  // statements;
  if (some_test) break;
  // more statements;
}
```

For the following exercise, see figure 6.2 for inspiration.

**Exercise 6.5.** Write a double loop over $0 \leq i, j < 10$ that prints all pairs $(i, j)$ where the product $i \cdot j > 40$.

*You can base this off the file `ijloop.cxx` in the repository*

**Exercise 6.6.** Write a double loop over $0 \leq i, j < 10$ that prints the first pair where the product of indices satisfies $i \cdot j > N$, where $N$ is a number your read in. A good test case is $N = 40$.

Secondly, find a pair with $i \cdot j > N$, but with the smallest value for $i + j$. (If there is more than one pair, report the one with lower $i$ value.) Can you traverse the $i, j$ indices such that they first enumerate all pairs $i + j = 1$, then $i + j = 2$, then $i + j = 3$ et cetera? Hint: write a loop over the sum value $1, 2, 3, \ldots$, then find $i, j$.

You program should print out both pairs, each on a separate line, with the numbers separated with a comma, for instance `8,5`.

**Exercise 6.7.** All three parts of a loop header are optional. What would be the meaning of

```
for (;;) { /* some code */ }
```

?

Suppose you want to know what the loop variable was when the break happened. You need the loop variable to be global:

```
int var;
... code that sets var ...
for ( ; var<upper; var++) {
  ... statements ...
  if (some condition) break
  ... more statements ...
}
... code that uses the breaking value of var ...
```

In other cases: define the loop variable in the header!

Example:

**Code:**

```
float minpos{0.f};
for ( ; ; minpos+=.5f ) {
  if (f(minpos)>90)
    break;
}
cout << "Minimum satisfying value: "
     << minpos << '\n';
```

**Output**

```
[loop] findmin:
Minimum satisfying value: 9.5
Minimum satisfying value: 9.5
```

**Exercise 6.8.** Can you make this loop more compact?

Instead of using a **`break`** statemeht, there can be other ways of ending the loop.

If the test comes at the start or end of an iteration, you can move it to the loop header:

```
bool need_to_stop{false};
for (int var=low; !need_to_stop ; var++) {
  ... some code ...
  if ( some condition )
    need_to_stop = true;
}
```

Another mechanism to alter the control flow in a loop is the `continue` statement. If this is encountered, execution skips to the start of the next iteration.

```
for (int var=low; var<N; var++) {
  statement;
  if (some_test) {
    statement;
    statement;
  }
}
```

Alternative:

```
for (int var=low; var<N; var++) {
  statement;
  if (!some_test) continue;
  statement;
  statement;
}
```

The only difference is in layout.

### 6.3.1 While loops

The other possibility for 'looping until' is a `while` loop, which repeats until a condition is met. The while loop does not have a counter or an update statement; if you need those, you have to create them yourself.

Syntax:

```
while ( condition ) {
  statements;
}
```

or

```
do {
  statements;
} while ( condition );
```

The two while loop variants can be described as 'pre-test' and 'post-test'. The choice between them entirely depends on context.

```
float money = inheritance();
while ( money < 1.e+6 )
  money += on_year_savings();
```

Let's consider an example: we read numbers from the input until one is positive. The following two code example use the `do .. while` and `while ... do` idiom respectively.

The first solution can be termed a 'pre-test':

**Code:**

```
cout << "Enter a positive number: " ;
cin >> invar; cout << '\n';
cout << "You said: " << invar << '\n';
while (invar<=0) {
  cout << "Enter a positive number: " ;
  cin >> invar; cout << '\n';
  cout << "You said: " << invar << '\n';
}
cout << "Your positive number was "
     << invar << '\n';
```

**Output**
**[basic] whiledo:**

```
Enter a positive number:
You said: -3
Enter a positive number:
You said: 0
Enter a positive number:
You said: 2
Your positive number was 2
```

Problem: code duplication.

The second one uses a 'post-test', and you see that here it solves the problem of code duplication;

**Code:**

```
int invar;
do {
  cout << "Enter a positive number: " ;
  cin >> invar; cout << '\n';
  cout << "You said: " << invar << '\n';
} while (invar<=0);
cout << "Your positive number was: "
     << invar << '\n';
```

**Output**
**[basic] dowhile:**

```
Enter a positive number:
You said: -3
Enter a positive number:
You said: 0
Enter a positive number:
You said: 2
Your positive number was: 2
```

The post-test syntax leads to more elegant code.

**Exercise 6.9.** At this point you are ready to do the exercises in the prime numbers project, section 46.3.

**Exercise 6.10.** A horse is tied to a post with a 1 meter elastic band. A spider that was sitting on the post starts walking to the horse over the band, at 1cm/sec. This startles the horse, which runs away at 1m/sec. Assuming that the elastic band is infinitely stretchable, will the spider ever reach the horse?

**Exercise 6.11.** One bank account has 100 dollars and earns a 5 percent per year interest rate. Another account has 200 dollars but earns only 2 percent per year. In both cases the interest is deposited into the account.

After how many years will the amount of money in the first account be more than in the second? Solve this with a `while` loop.

Food for thought: compare solutions with a pre-test and post-test, and also using a for-loop.

## 6.4 Advanced topics

### 6.4.1 Parallelism

At the start of this chapter we mentioned the following examples of loops:

- A time-dependent numerical simulation executes a fixed number of steps, or until some stopping test.
- Recurrences:

$$x_{i+1} = f(x_i).$$

- Inspect or change every element of a database table.

The first two cases actually need to be performed in sequence, while the last one corresponds more to a mathematical 'forall' quantor. You will later learn two different syntaxes for this in the context of arrays. This difference can also be exploited when you learn *parallel programming*. Fortran has a *do concurrent* loop construct for this.

## 6.5 Exercises

**Exercise 6.12.** Find all triples of integers $u, v, w$ under 100 such that $u^2 + v^2 = w^2$. Make sure you omit duplicates of solutions you have already found.

**Exercise 6.13.** The integer sequence

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{if } u_n \text{ is odd} \end{cases}$$

leads to the *Collatz conjecture*: no matter the starting guess $u_1$, the sequence $n \mapsto u_n$ will always terminate at 1.

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \cdots$$

(What happens if you keep iterating after reaching 1?)

Try all starting values $u_1 = 1, \ldots, 1000$ to find the values that lead to the longest sequence: every time you find a sequence that is longer than the previous maximum, print out the starting number.

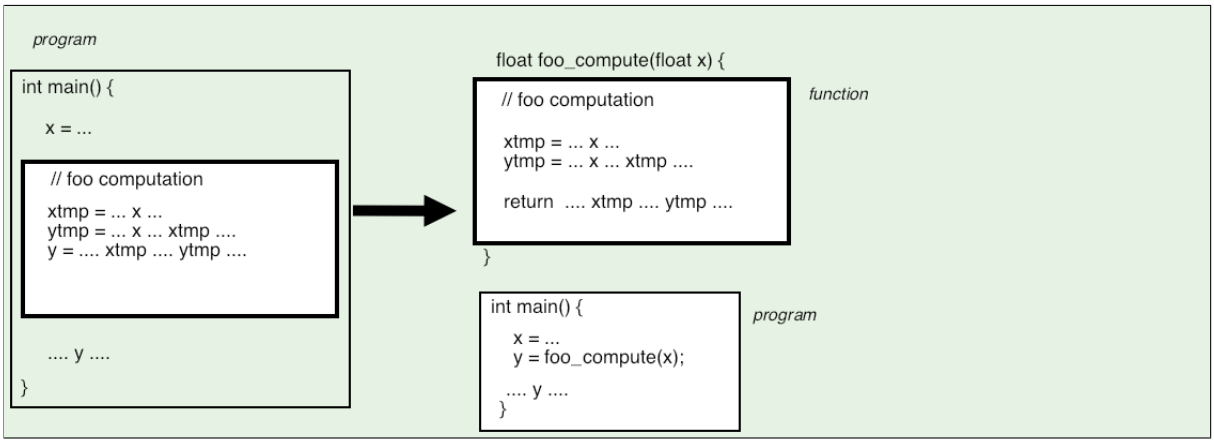
**Exercise 6.14.** Large integers are often printed with a comma (US usage) or a period (European usage) between all triples of digits. Write a program that accepts an integer such as $2542981$ from the input, and prints it as `2,542,981`.

# Chapter 7

# Functions

A *function* (or *subprogram*) is a way to abbreviate a block of code and replace it by a single line. This is foremost a code structuring device: by giving a function a relevant name you introduce the terminology of your application into your program.

- Find a block of code that has a clearly identifiable function.
- Turn this into a function: the function definition will contain that block, with a header that names it.
- The function is called by its name.




By introducing a function name you have introduced *abstraction*: your program now uses terms related to your problem, and not just the basic control structures such as **`for`**. With objects (chapter 9) you will learn further abstractions, so that instead of integers and arrays your program will use application terms, such as `Point` or `Line`.

## 7.1 Function definition and call

There are two aspects to a function:

- The *function definition* is done once, typically above the main program;
- a *function call* to any function can happen multiple times, inside the main program or inside other functions.

Let's consider a simple example program, in which we introduce functions. We code the *bisection* algorithm for finding the root of a function; see section 48.1 for details.

Example: zero-finding through bisection.

$$\underset{x}{?}: f(x) = 0, \qquad f(x) = x^3 - x^2 - 1$$

(where the question mark quantor stands for 'for which $x$').

First attempt at coding this: everything in the main program.

**Code:**

```
float left{0.},right{2.},
  mid;
while (right-left>.1) {
  mid = (left+right)/2.;
  float fmid =
    mid*mid*mid - mid*mid-1;
  if (fmid<0)
    left = mid;
  else
    right = mid;
}
cout << "Zero happens at: " << mid <<
    '\n';
```

**Output**

```
[func] bisect1:
Zero happens at: 1.4375
```

We modularize this in two steps. The first function we introduce is the objective function $f(x)$.

Introduce a function for the expression `m*m*m - m*m-1`:

```
float f(float x) {
  return x*x*x - x*x-1;
};
```

Used in main:

```
while (right-left>.1) {
  mid = (left+right)/2.;
  float fmid = f(mid);
  if (fmid<0)
    left = mid;
  else
    right = mid;
}
```

Next we introduce a function for the zero-finding algorithm.

Function:

```
float f(float x) {
  return x*x*x - x*x-1;
};
float find_zero_between
    (float l,float r) {
  float mid;
  while (r-l>.1) {
    mid = (l+r)/2.;
    float fmid = f(mid);
    if (fmid<0)
      l = mid;
    else
      r = mid;
  }
  return mid;
};
```

New main:

```
int main() {
  float left{0.},right{2.};
  float zero =
    find_zero_between(left,right);
  cout << "Zero happens at: "
       << zero << '\n';
  return 0;
}
```

The main now no longer contains any implementation details, such as local variables, or method used. This makes the main program shorter and more elegant: we have moved the variables for the midpoint inside the function. These are implementation details and should not be in the main program.

In this example, the function definition consists of:

- The keyword `float` indicates that the function returns a `float` result to the calling code.
- The name `find_zero_between` is picked by you.
- The parenthetical part `(float l,float r)` is called the 'parameter list': it says that the function takes two floats as input. For purposes of the function, the floats will have the names `l,r`, regardless any names in the main program.
- The 'body' of the function, the code that is going to be executed, is enclosed in curly brackets.
- A 'return' statement that transfers a computed result out of the function.

#### *7.1.0.1 Formal definition of a function definition*

Formally, a function definition consists of:

- *function result type*: you need to indicate the type of the result;
- name: you get to make this up;
- zero or more *function parameters*. These describe how many *function arguments* you need to supply as input to the function. Parameters consist of a type and a name. This makes them look like variable declarations, and that is how they function. Parameters are separated by commas. Then follows the:
- *function body*: the statements that make up the function. The function body is a *scope*: it can have local variables. (You can not nest function definitions.)
- a *return* statement. Which doesn't have to be the last statement, by the way.

The function can then be used in the main program, or in another function.

A function body defines a *scope*: the local variables of the function calculation are invisible to the calling program.

Functions can not be nested: you can not define a function inside the body of another function.

#### 7.1.0.2 Function call

The *function call* consists of

- The name of the function, and
- In between parentheses, any input argument(s).

The function call can stand on its own, or can be on the right-hand-side of an assignment.

**Exercise 7.1.** Make the bisection algorithm more elegant by introducing functions `new_l`, `new_r` used as:

```
l = new_l(l,mid,fmid);
r = new_r(r,mid,fmid);
```

*You can base this off the file* `bisect.cxx` *in the repository*

Question: you could leave out `fmid` from the functions. Write this variant. Why is this not a good idea?

The function call

1. copies the value of the *function argument* to the *function parameter*;
2. causes the function body to be executed, and
3. the function call is replaced by whatever you **`return`**.
4. (If the function does not return anything, for instance because it only prints output, you declare the return type to be **`void`**.)

To introduce two formal concepts:

- A function definition can have zero or more *parameter*s, or *formal parameter*s. These function as variable definitions local to the function.
- The function call has a corresponding number of *argument*s, or *actual parameter*s.

### 7.1.1 Why use functions?

In many cases, code that is written using functions can also be written without. So why would you use functions? There are several reasons for this.

Functions can be motivated as making your code more structured and intelligible. The source where you use the function call becomes shorter, and the function name makes the code more descriptive. This is sometimes called 'self-documenting code'.

Sometimes introducing a function can be motivated from a point of *code reuse*: if the same block of code appears in two places in your source (this is known as *code duplication*), you replace this by one function definition, and two (single line) function calls.

Suppose you do the same computation twice:

```
double x,y, v,w;
y = ..... computation from x .....
w = ..... same computation, but from v .....
```

With a function this can be replaced by:

```
double computation(double in) {
  return .... computation from 'in' ....
}

y = computation(x);
w = computation(v);
```

Example: multiple norm calculations:

Repeated code:

```
float s = 0;
for (int i=0; i<x.size(); i++)
  s += abs(x[i]);
cout << "One norm x: " << s << endl;
s = 0;
for (int i=0; i<y.size(); i++)
  s += abs(y[i]);
cout << "One norm y: " << s << endl;
```

becomes:

```
float OneNorm( vector<float> a ) {
  float sum = 0;
  for (int i=0; i<a.size(); i++)
    sum += abs(a[i]);
  return sum;
}
int main() {
  ... // stuff
  cout << "One norm x: "
       << OneNorm(x) << endl;
  cout << "One norm y: "
       << OneNorm(y) << endl;
```

(Don't worry about array stuff in this example)

A final argument for using functions is code maintainability:

- Easier to debug: if you use the same (or roughly the same) block of code twice, and you find an error, you need to fix it twice.
- Maintainance: if a block occurs twice, and you change something in such a block once, you have to remember to change the other occurrence(s) too.
- Localization: any variables that only serve the calculation in the function now have a limited *scope*.

```
void print_mod(int n,int d) {
  int m = n%d;
  cout << "The modulus of " << n << " and " << d
       << " is " << m << endl;
```

**Review** 7.1. True or false?

- The purpose of functions is to make your code shorter.
- Using functions makes your code easier to read and understand.
- Functions have to be defined before you can use them.
- Function definitions can go inside or outside the main program.

## 7.2 Anatomy of a function definition and call

Loosely, a function takes input and computes some result which is then returned. Just some simple examples:

```
int compute( float x, char c ) {
    /* code */
    return somevalue;
};
// in main:
i = compute(x,'c');
```

```
void compute( float x, char c ) {
    /* code */
};
// in main:
compute(x,'c');
```

So we need to discuss the function definition and its use.

## 7.3 Definition vs declaration

The C++ *compiler* translates your code in *one pass*: it goes from top to bottom through your code. This means you can not make reference to anything, such as a function name, that you haven't defined yet. For this reason, in the examples so far we put the function definition before the main program.

There is another solution. For the compiler to judge whether a function call is legal it does not need the full function definition: it can proceed once it know the name of the function, and the types of the inputs and result. This information is sometimes called a *function header*, *function prototype*, or *function signature*, but the technical term is a *function declaration*.

In the following example we put the function declaration before the main program, and the full function definition after it:

Some people like the following style of defining a function:

```
// declaration before main
int my_computation(int);

int main() {
  int result;
  result = my_computation(5);
  return 0;
};

// definition after main
int my_computation(int i) {
   return i+3;
}
```

This is purely a matter of style.

See chapter 19 for more details.

## 7.4 Void functions

Some functions do not return a result value, for instance because only write output to screen or file. In that case you define the function to be of type `void`.

```
void print_header() {
  cout << "****************" << endl;
  cout << "* Output       *" << endl;
  cout << "****************" << endl;
  }
int main() {
  print_header();
  cout << "The results for day 25:" << endl;
  // code that prints results ....
  return 0;
}
```

```
void print_result(int day,float value) {
  cout << "****************" << endl;
  cout << "* Output       *" << endl;
  cout << "****************" << endl;
  cout << "The results for day " << day << ":" << endl;
  cout << "    " << value << endl;
  }
int main() {
  print_result(25,3.456);
  return 0;
}
```

**Review** 7.2. True or false?

- A function can have only one input
- A function can have only one return result
- A void function can not have a `return` statement.

## 7.5 Parameter passing

C++ functions resemble mathematical functions: you have seen that a function can have an input and an output. In fact, they can have multiple inputs, separated by commas, but they have only one output.

$$a = f(x, y, i, j)$$

We start by studying functions that look like these mathematical functions. They involve a *parameter passing* mechanism called *passing by value*. Later we will then look at *passing by reference*.

### 7.5.1 Pass by value

The following style of programming is very much inspired by mathematical functions, and is known as *functional programming*[1].

1. There is more to functional programming. For instance, strictly speaking your whole program needs to be based on function calling; there is no other code than function definitions and calls.

- A function has one result, which is returned through a return statement. The function call then looks like

```
y = f(x1,x2,x3);
```

- Example:

**Code:**

```
double squared( double x ) {
  double y = x*x;
  return y;
}
/* ... */
number = 5.1;
cout << "Input starts as: "
     << number << '\n';
other = squared(number);
cout << "Output var is: "
     << other << '\n';
cout << "Input var is now: "
     << number << '\n';
```

**Output**
**[func] passvalue:**

```
Input starts as: 5.1
Output var is: 26.01
Input var is now: 5.1
```

- The definition of the C++ parameter passing mechanism says that input arguments are copied to the function, meaning that they don't change in the calling program:

**Code:**

```
double squared( double x ) {
  x = x*x;
  return x;
}
/* ... */
number = 5.1;
cout << "Input starts as: "
     << number << '\n';
other = squared(number);
cout << "Output var is: "
     << other << '\n';
cout << "Input var is now: "
     << number << '\n';
```

**Output**
**[func] passvaluelocal:**

```
Input starts as: 5.1
Output var is: 26.01
Input var is now: 5.1
```

We say that the input argument is *passed by value*: its value is copied into the function. In this example, the function parameter `x` acts as a local variable in the function, and it is initialized with a copy of the value of `number` in the main program.

Later we will worry about the cost of this copying.

**Exercise 7.2.** Write two functions

```
int biggest(int i,int j);
int smallest(int i,int j);
```

and a program that prints the results:

```
int i = 5, j = 17;
```

```
cout ... biggest(i,j) ...
cout ... smallest(i,j) ...
```

Passing a variable to a routine passes the value; in the routine, the variable is local. So, in this example the value of the argument is not changed:

**Code:**

```
void change_scalar(int i) {
  i += 1;
}
  /* ... */
  number = 3;
  cout << "Number is 3: "
       << number << '\n';
  change_scalar(number);
  cout << "is it still 3? Let's see: "
       << number << '\n';
```

**Output**
**[func] localparm:**

```
Number is 3: 3
is it still 3? Let's see: 3
```

**Exercise 7.3.** If you are doing the prime numbers project (chapter 46) you can now do exercise 46.6.

**Exercise 7.4.** If you are doing the zero-finding project (chapter 48) you can now do exercise 48.8.

### 7.5.2 Pass by reference

Having only one output is a limitation on functions. Therefore there is a mechanism for altering the input parameters and returning (possibly multiple) results that way. You do this by not copying values into the function parameters, but by turning the function parameters into aliases of the variables at the place where the function is called.

We need the concept of a *reference*: another variable to refers to the same 'thing' as another, already existing, variable.

A reference is indicated with an ampersand in its definition, and it acts as an alias of the thing it references.

**Code:**

```
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << '\n';
i *= 2;
cout << i << "," << ri << '\n';
ri -= 3;
cout << i << "," << ri << '\n';
```

**Output**
**[basic] ref:**

```
5,5
10,10
7,7
```

(You will not use references often this way.)

Correct:

```
float x{1.5};
```

```
float &xref = x;
```

Not correct:

```
float x{1.5};
float &xref;
xref = x;

float &threeref = 3; // WRONG: only reference to 'lvalue'
```

You can make a function parameter into a reference of a variable in the main program. This makes the function parameter into another name referring to the same thing.

The function parameter `n` becomes a reference to the variable `i` in the main program:

```
void f(int &n) {
  n = /* some expression */ ;
};
int main() {
  int i;
  f(i);
  // i now has the value that was set in the function
}
```

Reference syntax is cleaner than C 'pass by reference'

Using the ampersand, the parameter is *passed by reference*: instead of copying the value, the function receives a reference, so that the parameter becomes a reference to the thing in the *calling environment*.

**Remark 2** *The* pass by reference mechanism in C *was different and should not be used in C++. In fact it was not a true pass by reference, but passing an address by value.*

We also the following terminology for function parameters:

- *input* parameters: passed by value, so that it only functions as input to the function, and no result is output through this parameter;
- *output* parameters: passed by reference so that they return an 'output' value to the program.
- *throughput* parameters: these are passed by reference, and they have an initial value when the function is called. In C++, unlike Fortran, there is no real separate syntax for these.

**Code:**

```
void f( int &i ) {
  i = 5;
}
int main() {

  int var = 0;
  f(var);
  cout << var << '\n';
```

**Output**
**[basic] setbyref:**

```
5
```

Compare the difference with leaving out the reference.

As an example, consider a function that tries to read a value from a file. With anything file-related, you always have to worry about the case of the file not existing and such. So our function returns:

- a boolean value to indicate whether the read succeeded, and
- the actual value if the read succeeded.

The following is a common idiom, where the success value is returned through the **return** statement, and the value through a parameter.

```
bool can_read_value( int &value ) {
  // this uses functions defined elsewhere
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status==0;
}

int main() {
  int n;
  if (!can_read_value(n)) {
    // if you can't read the value, set a default
    n = 10;
  }
  ..... do something with 'n' ....
```

This latter example can also be solved, perhaps more idiomatically, with **std**::*optional*; section 24.5.2.

**Exercise 7.5.** Write a **void** function *swap* of two parameters that exchanges the input values:

**Code:**

```
int i=1,j=2;
cout << i << "," << j << '\n';
swap(i,j);
cout << i << "," << j << '\n';
```

**Output**
**[func] swap:**

```
1,2
2,1
```

**Exercise 7.6.** Write a divisibility function that takes a number and a divisor, and gives:

- a **bool** return result indicating that the number is divisible, and
- a remainder as output parameter.

**Code:**

```
cout << number;
if
    (is_divisible(number,divisor,remainder))
  cout << " is divisible by ";
else
  cout << " has remainder "
       << remainder << " from ";
cout << divisor << '\n';
```

**Output**
**[func] divisible:**

```
8 has remainder 2 from 3
8 is divisible by 4
```

**Exercise 7.7.** If you are doing the geometry project, you should now do the exercises in section 47.1.

## 7.6 Recursive functions

In mathematics, sequences are often recursively defined. For instance, the sequence of factorials $n \mapsto f_n \equiv n!$ can be defined as

$$f_0 = 1, \qquad \forall_{n>0}: f_n = n \times f_{n-1}.$$

Instead of using a subscript, we write an argument in parentheses

$$F(n) = \begin{cases} n \times F(n-1) & \text{if } n > 0 \\ 1 & \text{otherwise} \end{cases}$$

This is a form that can be translated into a C++ function. The header of a factorial function can look like:

```
int factorial(int n)
```

So what would the function body be? We need a `return` statement, and what we return should be $n \times F(n-1)$:

```
int factorial(int n) {
  return n*factorial(n-1);
} // almost correct, but not quite
```

So what happens if you write

```
int f3; f3 = factorial(3);
```

Well,

- The expression `factorial(3)` calls the `factorial` function, substituting `3` for the argument `n`.
- The return statement returns `n*factorial(n-1)`, in this case `3*factorial(2)`.
- But what is `factorial(2)`? Evaluating that expression means that the `factorial` function is called again, but now with `n` equal to 2.
- Evaluating `factorial(2)` returns `2*factorial(1)`,...
- ... which returns `1*factorial(0)`,...
- ... which returns ...
- Uh oh. We forgot to include the case where `n` is zero. Let's fix that:

```
int factorial(int n) {
  if (n==0)
    return 1;
  else
    return n*factorial(n-1);
}
```

- Now `factorial(0)` is 1, so `factorial(1)` is `1*factorial(0)`, which is 1,...
- ... so `factorial(2)` is 2, and `factorial(3)` is 6.

**Exercise 7.8.** It is possible to define multiplication as repeated addition:

**Code:**

```
int times( int number,int mult ) {
  cout << "(" << mult << ")";
  if (mult==1)
    return number;
  else
    return number + times(number,mult-1);
}
```

**Output**
**[func] mult:**

```
Enter number and multiplier
recursive multiplication
 of 7 and 5: (5)(4)(3)(2)(1)35
```

Extend this idea to define powers as repeated multiplication.

*You can base this off the file* `mult.cxx` *in the repository*

**Exercise 7.9.** The *Egyptian multiplication* algorithm is almost 4000 years old. The result of multiplying $x \times n$ is:

if $n$ is even:
  twice the multiplication $x \times (n/2)$;
otherwise if $n == 1$:
  $x$
otherwise:
  $x$ plus the multiplication $x \times (n-1)$

Extend the code of exercise 7.8 to implement this.

Food for thought: discuss the computational aspects of this algorithm to the traditional one of repeated addition.

**Exercise 7.10.** The sum of squares:

$$S_n = \sum_{n=1}^{N} n^2$$

can be defined recursively as

$$S_1 = 1, \qquad S_n = n^2 + S_{n-1}.$$

Write a recursive function that implements this second definition. Test it on numbers that are input interactively.

Then write a program that prints the first 100 sums of squares.

How many squares do you need to sum before you get overflow? Can you estimate this number without running your program?

**Exercise 7.11.** Write a recursive function for computing Fibonacci numbers:

$$F_0 = 1, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

First write a program that computes $F_n$ for a value $n$ that is input interactively.

Then write a program that prints out a sequence of Fibonacci numbers; set interactively how many.

**Remark 3** *A function does not need to call itself directly to be recursive; if it does so indirectly we can call this* mutual recursion.

```
int f(int n) { return g(n-1); }
int g(int n) { return f(n); }
```

**Remark 4** *If you let your Fibonacci program print out each time it computes a value, you'll see that most values are computed several times. (Math question: how many times?) This is wasteful in running time. This problem is addressed in section 66.3.1.*

## 7.7 Other function topics

### 7.7.1 Math functions

Some *math functions*, such as `abs`, can be included through `cmath`:

```
#include <cmath>
using std::abs;
```

Note that `std::abs` is polymorphic. Without that namespace indication an integer function `abs` is used, and the compiler may suggest that you use `fabs` for floating point arguments.

Others math functions, such as *max*, are in the less common `algorithm` header (see section 14.2):

```
#include <algorithm>
using std::max;
```

### 7.7.2 Default arguments

Functions can have *default argument*(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
  ...
  d = distance(x); // distance to origin
  d = distance(x,y); // distance between two points
```

Any default argument(s) should come last in the parameter list.

Don't trace a function unless I say so:

```
void dosomething(double x,bool trace=false) {
  if (trace) // report on stuff
};
int main() {
  dosomething(1); // this one I trust
  dosomething(2); // this one I trust
```

```
dosomething(3,true); // this one I want to trace!
dosomething(4); // this one I trust
dosomething(5); // this one I trust
```

### 7.7.3 Polymorphic functions

You can have multiple functions with the same name:

```
double average(double a,double b) {
  return (a+b)/2; }
double average(double a,double b,double c) {
  return (a+b+c)/3; }
```

Distinguished by type or number of input arguments: can not differ only in return type.

```
int f(int x);
string f(int x); // DOES NOT WORK
```

### 7.7.4 Stack overflow

So far you have seen only very simple recursive functions. Consider the function

$$\forall_{n>1}\colon g_n = (n-1)\cdot g(n-1), \qquad g(1) = 1$$

and its implementation:

```
int multifact( int n ) {
  if (n==1)
    return 1;
  else {
    int oneless = n-1;
    return oneless*multifact(oneless);
  }
}
```

Now the function has a local variable. Suppose we compute $g(3)$. That involves

```
int oneless = 2;
```

and then the computation of $g_2$. But that computation involved

```
int oneless = 1;
```

Do we still get the right result for $g_3$? Is it going to compute $g_3 = 2 \cdot g_2$ or $g_3 = 1 \cdot g_2$?

Not to worry: each time you call `multifact` a new local variable `oneless` gets created 'on the *stack*'. That is good, because it means your program will be correct[2], but it also means that if your function has both

- a large amount of local data, and
- a large *recursion depth*,

it may lead to *stack overflow*.

2. Historical note: very old versions of Fortran did not do this, and so recursive functions were basically impossible.

## 7.8 Review questions

**Review** 7.3. What is the output of the following programs? Assume that each program starts with

```
#include <iostream>
using std::cout;
using std::endl;
```

```
int add1(int i) {
  return i+1;
}
int main() {
  int i=5;
  i = add1(i);
  cout << i << endl;
}
```

```
void add1(int i) {
  i = i+1;
}
int main() {
  int i=5;
  add1(i);
  cout << i << endl;
}
```

```
void add1(int &i) {
  i = i+1;
}
int main() {
  int i=5;
  add1(i);
  cout << i << endl;
}
```

```
int add1(int &i) {
  return i+1;
}
int main() {
  int i=5;
  i = add1(i);
  cout << i << endl;
}
```

**Review** 7.4. Suppose a function

```
bool f(int);
```

is given, which is true for some positive input value. Write a main program that finds the smallest positive input value for which `f` is true.

**Review** 7.5. Suppose a function

```
bool f(int);
```

is given, which is true for some negative input value. Write a code fragment that finds the (negative) input with smallest absolute value for which `f` is true.

**Review** 7.6. Suppose a function

```
bool f(int);
```

is given, which computes some property of integers. Write a code fragment that tests if $f(i)$ is true for some $0 \leq i < 100$, and if so, prints a message.

**Review** 7.7. Suppose a function

```
bool f(int);
```

is given, which computes some property of integers. Write a main program that tests if $f(i)$ is true for all $0 \leq i < 100$, and if so, prints a message.

# Chapter 8

# Scope

## 8.1 Scope rules

The concept of *scope* answers the question 'when is the binding between a name (read: variable) and the internal entity valid'.

### 8.1.1 Lexical scope

C++, like Fortran and most other modern languages, uses *lexical scope* rule. This means that you can textually determine what a variable name refers to.

```
int main() {
  int i;
  if ( something ) {
    int j;
    // code with i and j
  }
  int k;
  // code with i and k
}
```

- The lexical scope of the variables `i,k` is the main program including any blocks in it, such as the conditional, from the point of definition onward. You can think that the variable in memory is created when the program execution reaches that statement, and after that it can be referred to by that name.
- The lexical scope of `j` is limited to the true branch of the conditional. The integer quantity is only created if the true branch is executed, and you can refer to it during that execution. After execution leaves the conditional, the name ceases to exist, and so does the integer in memory.
- In general you can say that any *use* of a name has be in the lexical scope of that variable, and after its *definition*.

### 8.1.2 Shadowing

Scope can be limited by an occurrence of a variable by the same name:

**Code:**

```
bool something{true};
int i = 3;
if ( something ) {
  int i = 5;
  cout << "Local: " << i << '\n';
}
cout << "Global: " << i << '\n';
if ( something ) {
  float i = 1.2;
  cout << "Local again: " << i << '\n';
}
cout << "Global again: " << i << '\n';
```

**Output**
**[basic] shadowtrue:**

```
Local: 5
Global: 3
Local again: 1.2
Global again: 3
```

The first variable `i` has lexical scope of the whole program, minus the two conditionals. While its *lifetime* is the whole program, it is unreachable in places because it is *shadowed* by the variable `i` in the conditionals.

This is independent of dynamic / runtime behavior!

**Exercise 8.1.** What is the output of this code?

```
bool something{false};
int i = 3;
if ( something ) {
  int i = 5;
  cout << "Local: " << i << '\n';
}
cout << "Global: " << i << '\n';
if ( something ) {
  float i = 1.2;
  cout << i << '\n';
  cout << "Local again: " << i << '\n';
}
cout << "Global again: " << i << '\n';
```

**Exercise 8.2.** What is the output of this code?

```
for (int i=0; i<2; i++) {
  int j; cout << j << endl;
  j = 2; cout << j << endl;
}
```

### 8.1.3 Lifetime versus reachability

The use of functions introduces a complication in the lexical scope story: a variable can be present in memory, but may not be textually accessible:

```
void f() {
  ...
}
int main() {
```

```
  int i;
  f();
  cout << i;
}
```

During the execution of `f`, the variable `i` is present in memory, and it is unaltered after the execution of `f`, but it is not accessible.

A special case of this is recursion:

```
void f(int i) {
  int j = i;
  if (i<100)
    f(i+1);
}
```

Now each incarnation of `f` has a local variable `i`; during a recursive call the outer `i` is still alive, but it is inaccessible.

### 8.1.4 Scope subtleties

#### *8.1.4.1 Mutual recursion*

If you have two functions `f, g` that call each other,

```
int f(int i) { return g(i-1); }
int g(int i) { return f(i+1); }
```

you need a *forward declaration*

```
int g(int);
int f(int i) { return g(i-1); }
int g(int i) { return f(i+1); }
```

since the use of name `g` has to come after its declaration.

There is also forward declaration of *classes*. You can use this if one class contains a pointer to the other:

```
class B;
class A {
private:
  shared_ptr<B> myB;
};
class B {
private:
  int myint;
}
```

You can also use a forward declaration if one class is an argument or return type:

```
class B;
class A {
public:
  B GimmeaB();
};
class B {
public:
  B(int);
}
```

However, there is a subtlety here: in the definition of `A` you can not give the full definition of the function that return `B`:

```
class B;
class A {
public:
  B GimmeaB() { return B(5); }; // WRONG: does not compile
};
```

because the compiler does not yet know the form of the `B` constructor.

The right way:

```
class B;
class A {
public:
  B GimmeaB();
};
class B {
public:
  B(int);
}
B A::GimmeaB() { return B(5); };
```

#### 8.1.4.2 Closures

The use of lambdas or *closures* (chapter 13) comes with another exception to general scope rules. Read about 'capture'.

## 8.2 Static variables

Variables in a function have *lexical scope* limited to that function. Normally they also have *dynamic scope* limited to the function execution: after the function finishes they completely disappear. (Class objects have their *destructor* called.)

There is an exception: a *static variable* persists between function invocations.

```
void fun() {
  static int remember;
}
```

For example

```
int onemore() {
  static int remember++; return remember;
}
int main() {
  for ( ... )
    cout << onemore() << end;
  return 0;
}
```

gives a stream of integers.

**Exercise 8.3.** The static variable in the `onemore` function is never initialized. Can you find a mechanism for doing so? Can you do it with a default argument to the function?

## 8.3 Scope and memory

The notion of scope is connected to the fact that variables correspond to objects in memory. Memory is only reserved for an entity during the dynamic scope of the entity. This story is clear in simple cases:

```
int main() {
  // memory reserved for 'i'
  if (true) {
    int i; // now reserving memory for integer i
    ... code ...
  }
  // memory for 'i' is released
}
```

Recursive functions offer a complication:

```
int f(int i) {
  int itmp;
  ... code with 'itmp' ...
  if (something)
    return f(i-1);
  else return 1;
}
```

Now each recursive call of `f` reserves space for its own incarnation of `itmp`.

In both of the above cases the variables are said to be on the *stack*: each next level of scope nesting or recursive function invocation creates new memory space, and that space is released before the enclosing level is released.

Objects behave like variables as described above: their memory is released when they go out of scope. However, in addition, a *destructor* is called on the object, and on all its contained objects:

**Code:**

```
1  class SomeObject {
2  public:
3    SomeObject() {
4      cout << "calling the constructor"
5           << '\n';
6    };
7    ~SomeObject() {
8      cout << "calling the destructor"
9           << '\n';
10   };
11 };
```

**Output**
**[object] destructor:**

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

## 8.4 Review questions

**Review** 8.1. Is this a valid program?

```
void f() { i = 1; }
int main() {
  int i=2;
  f();
  return 0;
}
```

If yes, what does it do; if no, why not?

**Review** 8.2. What is the output of:

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
  int i=5;
  if (true) { i = 6; }
  cout << i << endl;
  return 0;
}
```

**Review** 8.3. What is the output of:

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
  int i=5;
  if (true) { int i = 6; }
  cout << i << endl;
  return 0;
}
```

**Review** 8.4. What is the output of:

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
  int i=2;
  i += /* 5;
  i += */ 6;
  cout << i << endl;
  return 0;
}
```

# Chapter 9

# Classes and objects

## 9.1 What is an object?

You have now learned about elementary data, control structures, and functions. Ultimately, that's all there is to programming: data and operations on them. However, to keep your programs manageable it is a good idea to structure them, and recognize that you really want to talk at a higher level of abstraction.

C++ offers an important mechanism of unifying data and operations to give a new level of abstraction: *object*s belonging to *class*es.

> An object is an entity that you can request to do certain things.
> When designing a class, first ask yourself: 'what functionality should the objects support'.

- The actions an object is capable of are the *methods* or *function member*s of the object; and
- to make these actions possible the object probably stores data, the *data member*s.
- Objects comes in *class*es. A class is like a datatype: you can make objects of a class like you make variables of a datatype.
- Objects of the same class have the same methods. They also have the same members, but with individual values.

Classes are like datatypes in that you can declare variables of that type, which can then be used in expressions. Unlike basic datatypes, they are not predefined, so you first need to define the class before you can make objects of that class.

- You need a class definition, typically placed before the main program.
- (In larger programs you would put it in a *include file* or *module*.)
- You can then declare multiple objects belonging to that class.
- Objects can then be used in expressions, passed as parameter, et cetera.

### 9.1.1 First examples: points in the plane

Let's look at a simple example: we are going to create a `Point` object, corresponding to a mathematical point in $\mathbb{R}^2$.

**Exercise 9.1.** Thought exercise: what are some of the actions that a point object should be capable of?

The first things we are going to do with a point are to query some of its properties: given a point, you could want to know its distance to the origin or its angle with the $x$-axis.

Small illustration: point objects.

**Code:**

```
Point p(1.,2.); // make point (1,2)
cout << "distance to origin "
     << p.distance_to_origin()
     << '\n';
p.scaleby(2.);
cout << "distance to origin "
     << p.distance_to_origin()
     << '\n'
     << "and angle " << p.angle()
     << '\n';
```

**Output**
**[object] functionality:**

```
distance to origin 2.23607
distance to origin 4.47214
and angle 1.10715
```

Note the 'dot' notation.

**Exercise 9.2.** Thought exercise:
What data does the object need to store to be able to calculate angle and distance to the origin?
Is there more than one possibility?

Food for thought: you may be tempted to write methods for getting the $x$ and $y$ coordinate. However, ask yourself if those should be publicly visible methods. Is getting the $x$ coordinate a linear algebra operation? When you have methods such as 'get the distance to the origin' or 'shift this point rightward', do you explicitly need the coordinates?

The above example used a `Point` object without saying how it was created. That's what we are going to look at next.

- First define the class, with data and function members:

  ```
  class MyObject {
    // define class members
    // define class methods
  };
  ```

  (details later) typically before the `main`.
- You create specific objects with a declaration

  ```
  MyObject
    object1( /* .. */ ),
    object2( /* .. */ );
  ```

- You let the objects do things:

  ```
  object1.do_this();
  x = object2.do_that( /* ... */ );
  ```

Best practice we will use:

```
class MyClass {
private:
  // data members
public:
  // methods
}
```

- Data is private: not visible outside of the objects.
- Methods are public: can be used in the code that uses objects.
- You can have multiple private/public sections, in any order.

Let's now introduce the details of all these steps.

### 9.1.2 Constructor

First we'll look at creating class objects, and we'll stick with the point example.

Since a point can be defined by its $x, y$ coordinates, you can imagine that

- the point object stores these coordinates, and
- when you create a point object, you do that by specifying the coordinates.

Here are the relevant fragments of code:

The declaration of an object `x` of class `Point`; the coordinates of the point are initially set to `1.5,2.5`.

```
Point x(1.5, 2.5);
```

```
class Point {
private: // data members
  double x,y;
public: // function members
  Point
    ( double x_in,double y_in ) {
      x = x_in; y = y_in;
  };
  /* ... */
};
```

Study the implementation closely. The class is named `Point`, and there is something that looks like a function definition, also named `Point`. However, unlike a regular function, it does not have a return type, not even **`void`**.

This function is named the *constructor* of the class, and it is characterized by:

- The constructor has the same name as the class, and
- it looks like a function definition, except that it has no return type.

When you create an object, in the manner you've seen in above examples, you actually call this constructor.

Usually you write your own constructor, for instance to initialize data members. In the case of the **`class`** `Point` the function of the constructor is to take the coordinates of the point and to copy them to private members of the `Point` object.

If the object you create can have sensible default values, you can also use a *default constructor*, which has no argument. We will get to that below; section 9.1.7.

### 9.1.3 Data members, introduction

In the examples so far, your created a point object from its coordinates

```
Point oneone(1.,1.);
```

and the `Point` object stored these coordinates. However, this connection between outward usage and internal implementation can be very different. Maybe you have an application that works in polar coordinates, in which case storing $r, \theta$ is more natural, or at least more convenient for computation. But you may still want to create a point from its Cartesian coordinates.

The arguments of the constructor imply nothing about what data members are stored!

Example: create a point in `x,y` Cartesian coordinates, but store `r,theta` polar coordinates:

```
#include <cmath>
class Point {
private: // members
  double r,theta;
public: // methods
  Point( double x,double y ) {
    r = sqrt(x*x+y*y);
    theta = atan2(y/x);
  }
```

Note: no change to outward API.

- Keyword **`private`** indicates that data is internal: not accessible from outside the object; can only be used inside function members.
- Keyword **`public`** indicates that the constructor function can be used in the program.

### 9.1.4 Methods, introduction

Methods are things you can ask your class objects to do. For instance, in the `Point` class, you could ask a point to report its distance to the origin, or you could ask it to scale its distance by some number.

Let's start with the simpler of these two: measuring the distance to the origin. Without classes and objects, you would write a function with $x, y$ coordinates as input, and a single number as output

```
float x = ..., y = ...;
float d = distance_to_origin(x,y);
```

For an object method this looks like:

```
float x=..., y=...;
Point p( x,y );
float d = p.distance_to_origin();
```

To point out differences and similarities:

- You're still using a function with a scalar output, but
- instead of input parameters we use the coordinates that are stored in the point object. These act as 'global variables', at least within the object.

- To apply this to a point, we use the 'dot' notation. You could pronounce this as 'p's distance to the origin'.

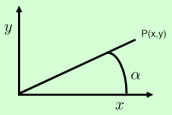
**Code:**

```
class Point {
private:
  float x,y;
public:
  Point(float ux,float uy) { x = ux; y
    = uy; };
  float distance_to_origin() {
    return sqrt( x*x + y*y );
  };
};
  /* ... */
  Point p1(1.0,1.0);
  float d = p1.distance_to_origin();
```

**Output**
**[geom] pointodist:**

```
Distance to origin: 1.41421
```

**Exercise 9.3.** Add a method `angle` to the `Point` class. How many parameters does it need?




Hint: use the function `atan` or `atan2`.

*You can base this off the file `pointclass.cxx` in the repository*

**Exercise 9.4.** Make a class `GridPoint` which can have only integer coordinates. Implement a function `manhattan_distance` which gives the distance to the origin counting how many steps horizontal plus vertical it takes to reach that point.

### 9.1.5 Initialization

There are various ways of setting the initial values of the data members of an object.

#### *9.1.5.1 Default values*

Sometimes it makes sense for objects to have default values if nothing else is specified. This can be done with setting default values on the data members:

Class members can have default values, just like ordinary variables:

```
class Point {
private:
  float x=3., y=.14;
public:
  // et cetera
}
```

Each object will have its members initialized to these values.

#### 9.1.5.2 *Initialization in the constructor*

Above you saw some examples of constructors that are used to initialize the object data. There is more than one way to do that.

First of all, you can copy the constructor arguments in the body of the constructor. However, the preferred way is by using *member initializer*s, which takes a new notation.

The naive way:

```
class Point {
private:
  double x,y;
public:
  Point( double in_x,
         double in_y ) {
    x = in_x; y = in_y;
  };
```

The preferred way:

```
class Point {
private:
  double x,y;
public:
  Point( double userx,
         double usery )
    : x(userx),y(usery) {
  }
```

(See section 10.6 for why member initializers are preferred.)

You can even save yourself from having to think of too many names:

**Code:**

```
class Point {
private:
  double x,y;
public:
  Point( double x,double y )
      : x(x),y(y) {
  }
  /* ... */
  Point p1(1.,2.);
  cout << "p1 = "
       << p1.getx() << "," << p1.gety()
       << '\n';
```

**Output**

```
[geom] pointinitxy:
p1 = 1,2
```

The initialization `x(x)` should be read as `membername(argumentname)`. Yes, having `x` twice is a little confusing.

### 9.1.6 Methods

You have just seen examples of class *method*s: a function that is only defined for objects of that class, and that has access to the private data of that object.

In exercise 9.3 you implemented an `angle` function, that computed the angle from the stored coordinates. You could have made other decisions.

**Exercise 9.5.** Discuss the pros and cons of this design:

```
class Point {
private:
  double x,y,r,theta;
public:
  Point(double xx,double yy) {
    x = xx; y = yy;
    r = // sqrt something
    theta = // something trig
  };
  double angle() { return alpha; };
};
```

By making these functions public, and the data members private, you define an Application Programmer Interface (API) for the class:

- You are defining operations for that class; they are the only way to access the data of the object.
- The methods can use the data of the object, or alter it. All data members, even when declared **private**, are global to the methods.
- Data members declared **private** are not accessible from outside the object.

**Review** 9.1. T/F?

- A class is primarily determined by the data it stores.
- A class is primarily determined by its methods.
- If you change the design of the class data, you need to change the constructor call.

Now let's look at some different types of objects. This is an informal classification, not necessarily corresponding to defined concepts in the C++ standard.

#### 9.1.6.1 *Changing state*

Objects usually have data members that maintain the *state* of the object. By changing the values of the members you change the state of the object. Doing so is usually done through a method.

For instance, you may want to scale a vector by some amount:

**Code:**

```
class Point {
  /* ... */
  void scaleby( double a ) {
    x *= a; y *= a; };
  /* ... */
};
  /* ... */
  Point p1(1.,2.);
  cout << "p1 to origin "
       << p1.length() << '\n';
  p1.scaleby(2.);
  cout << "p1 to origin "
       << p1.length() << '\n';
```

**Output**
**[geom] pointscaleby:**

```
p1 to origin 2.23607
p1 to origin 4.47214
```

**Exercise 9.6.** Implement a method `shift_right` for the `Point` class.

**Exercise 9.7.** Take the `Point` class design that uses polar coordinates (see above). Implement a `rotate` method.

There is a subtlety here. Hint: imagine rotating a point sufficiently many times.

### *9.1.6.2 Methods that return objects*

The methods you have seen so far only returned elementary datatypes. It is also possible to return an object, even from the same class. For instance, instead of scaling the members of a vector object, you could create a new object based on the scaled members:

**Code:**

```
class Point {
  /* ... */
  Point scale( double a ) {
    auto scaledpoint =
        Point( x*a, y*a );
    return scaledpoint;
  };
  /* ... */
  cout << "p1 to origin "
       << p1.dist_to_origin()
       << '\n';
  Point p2 = p1.scale(2.);
  cout << "p2 to origin "
       << p2.dist_to_origin()
       << '\n';
```

**Output**
**[geom] pointscale:**

```
p1 to origin 2.23607
p2 to origin 4.47214
```

Create a point by scaling another point:

```
new_point = old_point.scale(2.81);
```

Two ways of handling the `return` statement:

Naive:

```
Point Point::scale( double a ) {
  Point scaledpoint =
    Point( x*a, y*a );
  return scaledpoint;
};
```

Creates point, copies it to `new_point`

Better:

```
Point Point::scale( double a ) {
  return Point( x*a, y*a );
};
```

Creates point, moves it directly to `new_point`

'move semantics' and 'copy elision':compiler is pretty good at avoiding copies

## 9.1.7 Default constructor

You have now seen some examples of classes and their constructors. These constructors took arguments that set the initial *state* of the object.

However, if your objects have sensible default values, you can use a *default constructor*. For example:

No constructor explicitly defined;
You recognize the default constructor in the main by the fact that an object is defined without any parameters.

**Code:**

```
class IamOne {
private:
  int i=1;
public:
  void print() {
    cout << i << '\n';
  };
};
  /* ... */
  IamOne one;
  one.print();
```

**Output**

```
[object] defaultno:

1
```

You can define a default constructor yourself, but the previous example had a *defaulted* default constructor: it acted like it had a constructor

```
IamZero() {};
```

Bear this in mind as you study the following code:

```
Point p1(1.,2.), p2;
cout << "p1 to origin " << p1.length() << '\n';
p2 = p1.scale(2.);
cout << "p2 to origin " << p2.length() << '\n';
```

With the `Point` class (and its constructor) as given before:

```
class Point {
private:
  float x,y;
public:
  Point(float ux,float uy) { x = ux; y = uy; };
  float distance_to_origin() {
    return sqrt( x*x + y*y );
  };
};
  /* ... */
  Point p1(1.0,1.0);
  float d = p1.distance_to_origin();
```

this will give an error message during compilation. The reason is that

```
Point p2;
```

calls the default constructor. Now that you have defined your own constructor, the default constructor no longer exists. So you need to define it explicitly:

```
Point() {};
Point( double x,double y )
  : x(x),y(y) {};
```

You now have a class with two constructors. The compiler will figure out which one to use. This is an example of *polymorphism*.

You can also indicate somewhat more explicitly that the *defaulted default constructor* needs to exist:

```
Point() = default;
Point( double x,double y )
  : x(x),y(y) {};
```

**Remark 5** *The default constructor has 'empty parentheses', but you use it specifying no parentheses. What would happen if you specified empty parentheses when you create an object?*

```
class MyClass {
public:
  MyClass() { cout << "Construct!" << '\n'; };
};

int main() {

  MyClass x;
  MyClass y();
```

```
constructparen.cxx:24:12: warning:
empty parentheses interpreted as a function declaration
  MyClass y();
           ^
constructparen.cxx:24:12: note:
 remove parentheses to declare a variable
  MyClass y();
           ^
1 warning generated.
```

### 9.1.8 Data member access; invariants

You may have noticed the keywords `public` and `private`. We made the data members private, and the methods public. The C++ language also has the `struct` construct, inherited from C. In that one, data members are (by default) public. Why don't we do that here?

Struct data is public:

```
struct Point {
  double x;
};
int main() {
  Point andhalf;
  andhalf.x = 1.5;
}
```

```
class Point {
public: // Bad! Idea!
  double x;
};
int main() {
  Point andhalf;
  andhalf.x = 2.6;
}
```

Objects are really supposed to be accessed through their functionality. While you could write methods such as `get_x`, (this is called an *accessor*; see also section 18.4 for some subtleties) to get the $x$ coordinate, ask yourself if that makes sense. If you need the $x$ coordinate to shift the point rightward, write a `shift_right` method instead.

- Interface: `public` functions that determine the functionality of the object; effect on data members is secondary.
- Implementation: data members, keep `private`: they only support the functionality.

This separation is a Good Thing:

- Protect yourself against inadvertent changes of object data.
- Possible to change implementation without rewriting calling code.

You should not write access functions lightly: you should first think about what elements of your class should conceptually be inspectable or changeable by the outside world. Consider for example a class where a certain relation holds between members. In that case only changes are allowed that maintain that relation. It is sometimes said that a class satisfies an *invariant*.

You already saw this phenomenon in action in exercise 9.7. What was the invariant there? Let's consider another example of the need of maintaining an invariant.

We make a class for points on the unit circle

```
class UnitCirclePoint {
private:
  float x,y;
public:
  UnitCirclePoint(float x) {
    setx(x); };
  void setx(float newx) {
    x = newx; y = sqrt(1-x*x);
  };
```

You don't want to be able to change just one of `x,y`!
In general: enforce invariants on the members.

Section 9.5.4 has some further discussion on ways of directly accessing internal data.

### 9.1.9 Examples

So far, we have looked at examples of objects that represent 'object-like' things in the real world. However, we can also make objects for things that are more abstract. In the next example, we look at 'infinite objects', such as the set of all integers. Clearly, there is no way to store the data of such object, but the crucial question here is: what are the methods for an object that is the set of all integers? One possible design is that you could ask this object 'give me the next integer'.

Objects can model fairly abstract things:

**Code:**

```
class Stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

int main() {
  Stream ints;
  cout << "Next: "
       << ints.next() << '\n';
  cout << "Next: "
       << ints.next() << '\n';
  cout << "Next: "
       << ints.next() << '\n';
```

**Output**

```
[object] stream:
Next: 0
Next: 1
Next: 2
```

**Exercise 9.8.**

- Write a class `multiples_of_two` where every call of `next` yields the next multiple of two.
- Write a class `multiples` used as follows:

```
multiples multiples_of_three(3);
```

  where the `next` call gives the next multiple of the argument of the constructor.

*You can base this off the file `stream.cxx` in the repository*

**Exercise 9.9.** If you are doing the prime project (chapter 46), now is a good time to do exercise in section 46.6.

## 9.2 Inclusion relations between classes

The data members of an object can be of elementary datatypes, or they can be objects. For instance, if you write software to manage courses, each `Course` object will likely have a `Person` object, corresponding to the teacher.

```
class Person {
  string name;
  ....
}
class Course {
private:
  int year;
  Person the_instructor;
  vector<Person> students;
}
```

Designing objects with relations between them is a great mechanism for writing structured code, as it makes the objects in code behave like objects in the real world. The relation where one object contains another, is called a *has-a relation* between classes.

### 9.2.1 Literal and figurative has-a

Sometimes a class can behave as if it includes an object of another class, while not storing this other object. Consider the example of a line segment, that is, the segment from a starting point to an ending point. We want to offer the API:

```
int main() {
  Segment somesegment( /* something */ );
  Point somepoint = somesegment.get_the_end_point();
```

We can support this by letting the `Segment` class actually store the starting and ending points:

```
class Segment {
private:
  Point starting_point,ending_point;
}
```

or letting it store a distance and angle from the starting point:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
}
```

In both cases the code using the object is written as if the segment object contains two points. This illustrates how object-oriented programming can decouple the API of classes from their actual implementation.

Related to this decoupling, a class can also have two very different constructors.

```
class Segment {
  private:
  // up to you how to implement!
  public:
  Segment( Point start,float length,float angle )
  { .... }
  Segment( Point start,Point end ) { ... }
```

Depending on how you actually implement the class, the one constructor will simply store the defining data, and the other will do some conversion from the given data to the actually stored data.

This is another strength of object-oriented programming: you can change your mind about the implementation of a class without having to change the program that uses the class.

When you have a has-a relation between classes, the 'default constructor' problem (section 9.1.7) can pop up again:

Class for a person:

```
class Person {
private:
  string name;
public:
  Person( string name ) {
    /* ... */
  };
};
```

Class for a course, which contains a person:

```
class Course {
private:
  Person instructor;
  int enrollment;
public:
  Course( string instr,int n ) {
    /* ???? */
  };
};
```

You want to use this as `Course("Eijkhout",65);`

Possible constructor:

```
Course( string teachername,int nstudents ) {
  instructor = Person(teachername);
  enrollment = nstudents;
};
```

Preferred:

```
Course( string teachername,int nstudents )
  : instructor(Person(teachername)),
    enrollment(nstudents) {
};
```

```
class Inner { /* ... */ };
class Outer {
private:
  Inner inside_thing;
```

Two possibilities for constructor:

```
Outer( Inner thing )
: inside_thing(thing) {};
```

The `Inner` object is copied during construction of `Outer` object.

```
Outer( Inner thing ) {
  inside_thing = thing;
};
```

The `Outer` object is created, including construction of `Inner` object, then the argument is copied into place: ⇒ needs default constructor on `Inner`.

**Exercise 9.10.** If you are doing the geometry project, this is a good time to do the exercises in section 47.3.

#### 9.2.1.1 *Shorthand for objects*

For classes with a constructor, you can use a shorthand for an object, giving a brace-delimited initializer list.

*Initializer lists* can be used as denotations.

```
Point(float ux,float uy) {
/* ... */
Rectangle(Point bl,Point tr) {
/* ... */
Point origin{0.,0.};
Rectangle lielow( origin, {5,2} );
```

## 9.3 Inheritance

In addition to the has-a relation, there is the *is-a relation*, also called *inheritance*. Here one class is a special case of another class. Typically the object of the *derived class* (the special case) then also inherits the data and methods of the *base class* (the general case).

General `FunctionInterpolator` class with method `value_at`. Derived classes:

- `LagranceInterpolator` with `add_point_and_value`;
- `HermiteInterpolator` with `add_point_and_derivative`;
- `SplineInterpolator` with `set_degree`.

How do you define a derived class? The general code schema for use of a base class and derived class goes like this:

Base class, general case:

```
class General {
protected: // note!
 int g;
public:
 void general_method() {};
};
```

Derived class, special case:

```
class Special : public General {
public:
  void special_method() {
    ... g ...
  };
};
```

These are the various aspects of declaring a derived class:

- You need to indicate what the base class is:

  ```
  class Special : public General { .... }
  ```

- The base class needs to declare its data members as **`protected`**: this is similar to private, except that they are visible to derived classes
- The methods of the derived class can then refer to data of the base class;
- Any method or data member defined for the base class is available for a derived class object.

The derived class has its own constructor, with the same name as the class name, but when it is invoked, it also calls the constructor of the base class. This can be the default constructor, but often you want to

call the base constructor explicitly, with parameters that are describing how the special case relates to the base case.

In the following example, we have a general case, depending on two independent parameters. The special case comes from having a certain relationship between these parameters.

```
class General {
public:
  General( double x,double y ) {};
};
class Special : public General {
public:
  Special( double x ) : General(x,x+1) {};
};
```

Methods and data can be

- **private**, because they are only used internally;
- **public**, because they should be usable from outside a class object, for instance in the main program;
- **protected**, because they should be usable in derived classes.

**Exercise 9.11.** If you are doing the geometry project, you can now do the exercises in section 47.4.

### 9.3.1 Methods of base and derived classes

Above, it was assumed that derived classes use the methods of the base class unchanged. Sometimes, however, you may want the derived class have a different version of a method. This is done through the **virtual** and *override* keywords.

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class can *override* a base class method:

```
class Base {
public:
  virtual f() { ... };
};
class Deriv : public Base {
public:
  virtual f() override { ... };
};
```

**Code:**

```
class Base {
protected:
  int i;
public:
  Base(int i) : i(i) {};
  virtual int value() { return i; };
};

class Deriv : public Base {
public:
  Deriv(int i) : Base(i) {};
  virtual int value() override {
    int ivalue = Base::value();
    return ivalue*ivalue;
  };
};
```

**Output**

```
[object] virtual:

25
```

### 9.3.2 Virtual methods

The methods of base and derived classes can relate in a number of ways.

- Method defined in base class: can be used in any derived class.
- Method define in derived class: can only be used in that particular derived class.
- Method defined both in base and derived class, marked `override`: derived class method replaces (or extends) base class method.
- Virtual method: base class only declares that a routine has to exist, but does not give base implementation.
  A class is called *abstract class* if it has virtual methods; pure virtual if all methods are virtual.
  You can not make abstract objects.

Special syntax for *abstract method*:

```
class Base {
public:
  virtual void f() = 0;
};
class Deriv : public Base {
public:
  virtual void f() { ... };
};
```

```
class VirtualVector {
private:
public:
  virtual void setlinear(float) = 0;
  virtual float operator[](int) = 0;
};
```

Suppose `DenseVector` derives from `VirtualVector`:

```
DenseVector v(5);
v.setlinear(7.2);
cout << v[3] << '\n';
```

```
class DenseVector : VirtualVector {
private:
  vector<float> values;
public:
  DenseVector( int size ) {
    values = vector<float>(size,0);
  };
  void setlinear( float v ) {
    for (int i=0; i<values.size();
    i++)
      values[i] = i*v;
  };
  float operator[](int i) {
    return values.at(i);
  };
};
```

**Exercise 9.12.** Write a small 'integrator' library for Ordinary Diffential Equations (ODEs)s. Assuming 'autonomous ODEs', that is $u' = f(t)$ with no $u$-dependence, there are two simple integration schemes:

- explicit: $u_{n+1} = u_n + \Delta t\, f_n$; and
- implicit: $u_{n+1} = u_n + \Delta t\, f_{n+1}$.

Write an abstract `Integrator` class where the `nextstep` method (which integrates for another $\Delta t$) is pure virtual; then write `ExplicitIntegrator` and `ImplicitIntegrator` classes deriving from this.

```
double stepsize = .01;
auto integrate_linear =
  ForwardIntegrator( [] (double x) { return x*x; }, stepsize );
double int1 = integrate_linear.to(1.);
```

You can hardcode the function to be integrated, or try to pass a function pointer.

### 9.3.3 Friend classes

A *friend* class can access private data and methods even if there is no inheritance relationship.

```
/* forward definition: */ class A;
class B {
  // A objects can access B internals:
  friend class A;
private:
  int i;
};
class  A {
public:
  void f(B b) { b.i; }; // friend access
};
```

### 9.3.4 Multiple inheritance

- Multiple inheritance: an X is-a A, but also is-a B.
  This mechanism is somewhat dangerous.
- Virtual base class: you don't actually define a function in the base class, you only say 'any derived class has to define this function'.

**Exercise 9.13.** If you are doing the geometry project, this is a good time to do the exercises in section 47.2.

## 9.4 More about constructors

### 9.4.1 Delegating constructors

If you have two constructors where one is a special case of the other, there is an elegant mechanism for expressing that: *delegating constructors*.

As an example, consider a class that contains a vector, and you want to set that vector in the constructor. We could implement that as:

```
class HasVector {
private:
  vector<int> values;
public:
  HasVector( vector<int> initvalues )
  : values( initvalues ) {};
```

Now suppose we want the possibility that the vector of initial values is only the front part of the stored vector. Now we need a constructor that accepts the initial values, and an integer indicating the finished size.

```
  HasVector( vector<int> init,int size )
  : values( vector<int>(size) ) {
    int loc=0;
    for ( auto i : init )
      values[loc++] = i;
  };
```

(Question: this constructor is somewhat dangerous. What is the problem and how would you guard against it?)

With this we can simplify the first constructor we wrote:

```
  HasVector( vector<int> init )
  : HasVector( init,init.size() ) {};
```

Here we used the colon-notation to 'delegate' the constructor: one constructor is expressed in terms of another. (Question: in the context of classes, what are two other uses of the colon-notation?)

Everything together:

```
class HasVector {
private:
  vector<int> values;
public:
  HasVector( vector<int> init )
  : HasVector( init,init.size() ) {};
  HasVector( vector<int> init,int size )
  : values( vector<int>(size) ) {
    int loc=0;
    for ( auto i : init )
      values[loc++] = i;
  };
```

### 9.4.2 Copy constructor

Just like the default constructor which is defined if you don't define an explicit constructor, there is an implicitly defined *copy constructor*. There are two ways you can do a copy, and they invoke two slightly different constructors:

```
my_object y(something); // regular or default constructor
my_object x(y);         // copy constructor
my_object x = y;        // copy assignment constructor
```

Usually the copy constructor that is implicitly defined does the right thing: it copies all data members. (If you want to define your own copy constructor, you need to know its prototype. We will not go into that.)

As an example of the copy constructor in action, let's define a class that stores an integer as data member:

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v
         << '\n';
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v
         << '\n';
    mine = v; };
  void printme() {
    cout << "I have: " << mine
         << '\n'; };
};
```

The following code shows that the data got copied over:

**Code:**

```
has_int an_int(5);
has_int other_int(an_int);
an_int.printme();
other_int.printme();
has_int yet_other = other_int;
yet_other.printme();
```

**Output**
**[object] copyscalar:**

```
set: 5
copy: 5
I have: 5
I have: 5
copy: 5
I have: 5
```

Class with a vector:

```
class has_vector {
private:
  vector<int> myvector;
public:
  has_vector(int v) { myvector.push_back(v); };
  void set(int v) { myvector.at(0) = v; };
  void printme() { cout
      << "I have: " << myvector.at(0) << '\n'; };
};
```

Copying is recursive, so the copy has its own vector:

**Code:**

```
has_vector a_vector(5);
has_vector other_vector(a_vector);
a_vector.set(3);
a_vector.printme();
other_vector.printme();
```

**Output**
**[object] copyvector:**

```
I have: 3
I have: 5
```

### 9.4.3 Destructor

Just as there is a constructor routine to create an object, there is a *destructor* to destroy the object. As with the case of the default constructor, there is a default destructor, which you can replace with your own.

A destructor can be useful if your object contains dynamically created data: you want to use the destructor to dispose of that dynamic data to prevent a *memory leak*. Another example is closing files for which the *file handle* is stored in the object.

The destructor is typically called without you noticing it. For instance, any statically created object is destroyed when the control flow leaves its scope.

Example:

**Code:**

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << '\n';
  };
  ~SomeObject() {
    cout << "calling the destructor"
         << '\n';
  };
};
```

**Output**
**[object] destructor:**

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

**Exercise 9.14.** Write a class

```
class HasInt {
private:
  int mydata;
public:
  HasInt(int v) { /* initialize */ };
  ...
}
```

used as

**Code:**

```
{
  HasInt v(5);
  v.set(6);
  v.set(-2);
}
```

**Output**
**[object] destructexercise:**

```
**** object created with 5 ****
**** object set to 6 ****
**** object set to -2 ****
**** object destroyed after 2
    updates ****
```

The destructor is called when you throw an exception:

**Code:**

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << '\n'; };
  ~SomeObject() {
    cout <<  "calling the destructor"
         << '\n'; };
};
  /* ... */
  try {
    SomeObject obj;
    cout << "Inside the nested scope" <<
    '\n';
    throw(1);
  } catch (...) {
    cout << "Exception caught" << '\n';
  }
```

**Output**
**[object] exceptdestruct:**

```
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

## 9.5 Advanced topics

*The remainder of this section is advanced material. Make sure you have studied section 15.3.*

### 9.5.1 Static variables and methods

Class members prefixed with **static** behave as if they are not unique to each object of that class, but shared between them.

The standard use for this is to count how many objects of a class have been created. The constructor would increment this static variable and assign it to a private variable:

```
Thing::Thing() {
  mynumber = n_things++; };
```

Declaring the static variable takes the keywords **static** and **inline**:

```
class Thing {
private:
  static inline int n_things=0; // global count
  int mynumber; // who am I?
```

#### *9.5.1.1 Static methods*

If you want to query the above static variables you can of course query them from any particular object, since they all have the same value. However, you can define a static method:

```
class Thing {
public:
  static int number_of_things() { return n_things; };
```

and this method can be called on the class itself;

```
cout << "Number of things: " << Thing::number_of_things() << '\n';
```

#### 9.5.1.2 Legacy syntax for initialization

Prior to C++17, initializing the static variable was done in a strange way. Currently, by adding the keyword `inline`, you can write:

**Code:**

```
class Thing {
private:
  static inline int number{0};
  int mynumber;
public:
  Thing() {
    mynumber = number++;
    cout << "I am thing "
         << mynumber << '\n';
  };
};
```

**Output**
**[object] static:**

```
I am thing 0
I am thing 1
I am thing 2
```

In case you come across it in legacy code, there is the C++11 syntax for static class members:

```
class myclass {
private:
  static int count;
public:
  myclass() { count++; };
  int create_count() { return count; };
};
  /* ... */
// in main program
int myclass::count=0;
```

### 9.5.2 Class signatures

For purposes of organizing your code, you may sometimes not want to include the full code of a method, for instance in a header file. This is the distinction between a *declaration* and *definition*.

You have seen this with functions:

```
float f(int);
```

is the *declaration* of a function, stating the name of the function, the types of the input parameters and the type of the return result. (This is sometimes also called the 'signature' or 'prototype' or 'header' of the function.) On the other hand

```
float f(int n) { return sqrt(n); }
```

is the *definition* of the function, giving its full code.

Similarly, you can write class declaration, giving only the data members and signatures of the class methods, and specify the full method later or elsewhere. This can for instance happen if you split your program over multiple files; see chapter 19 and in particular section 19.3.

Declaration:

```
class Point {
private:
  float x,y;
public:
  Point(float x,float y);
  float distance();
};
```

Definition:

```
Point::Point()
  : x(x),y(y) {};
float Point::distance() {
  return sqrt( x*x + y*y );
};
```

- Methods, including constructors, are only given by their function header in the class definition.
- Methods and constructors are then given their full definition elsewhere with 'classname-double-colon-methodname' as their name.
- (qualifiers like `const` are given in both places.)

### 9.5.3 Returning by reference

With all the above discussion of the API abstracting away from internals, sometimes you really want to access the internals of an object directly. The simplest solution is to return a copy:

```
class Foo {
private:
  int x;
public:
  int the_x() { return x; };
};
```

There are two problems with this:

- Returning a copy can be expensive if the internal data is a big object; and
- Maybe you actually want to alter the internal data.

So we have two scenarios:

- you want to get a reference to a data member, in order to alter it;
- you want to get a reference to a data member because copying is expensive, but you will not alter it.

First we show the general mechanism of returning a reference to private data.

Return a reference to a private member:

```
class Point {
private:
  double x,y;
public:
  double &x_component() { return x; };
};
int main() {
  Point v;
```

```
  v.x_component() = 3.1;
}
```

Only define this if you need to be able to alter the internal entity.

Next we show a mechanism that can be considered a performance optimization to this: we return a reference to private data, but in such a way that the calling side can not alter it.

Returning a reference saves you on copying.
Prevent unwanted changes by using a 'const reference'.

```
class Grid {
private:
  vector<Point> thepoints;
public:
  const vector<Point> &points() const {
    return thepoints; };
};
int main() {
  Grid grid;
  cout << grid.points()[0];
  // grid.points()[0] = whatever ILLEGAL
}
```

### 9.5.4 Accessor functions

It is a good idea to make the data in an object private, so that you can control who has outside access to it.

- Sometimes this private data is auxiliary, and there is no reason for outside access.
- Sometimes you do want outside access, but you want to precisely control how.

Accessor functions:

```
class thing {
private:
 float x;
public:
 float get_x() { return x; };
 void set_x(float v) { x = v; }
};
```

This has advantages:

- You can print out any time you get/set the value; great for debugging:

  ```
  void set_x(float v) {
    cout << "setting: " << v << endl;
    x = v; }
  ```

- You can catch specific values: if $x$ is always supposed to be positive, print an error (throw an exception) if non-positive.

Having two accessors can be a little clumsy. Is it possible to use the same accessor for getting and setting?

Use a single accessor for getting and setting:

**Code:**

```
class SomeObject {
private:
  float x=0.;
public:
  SomeObject( float v ) : x(v) {};
  float &xvalue() { return x; };
};

int main() {
  SomeObject myobject(1.);
  cout << "Object member initially :"
       << myobject.xvalue() << '\n';
  myobject.xvalue() = 3.;
  cout << "Object member updated   :"
       << myobject.xvalue() << '\n';
```

**Output**

```
[object] accessref:

Object member initially :1
Object member updated   :3
```

The function `xvalue` returns a reference to the internal variable `x`.

Of course you should only do this if you want the internal variable to be directly changeable!

### 9.5.5 Polymorphism

You can have multiple methods with the same name, as long as they can be distinguished by their argument types. This is known as *polymorphism*; see section 7.7.3.

### 9.5.6 Operator overloading

Instead of writing

```
myobject.plus(anotherobject)
```

you can actually redefine the + operator so that

```
myobject + anotherobject
```

is legal. This is known as *operator overloading*: you give your own definition of common arithmetic operators.

Syntax:

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

**Code:**

```
Point Point::operator*(double f) {
    return Point(f*x,f*y);
};
  /* ... */
  cout << "p1 to origin "
       << p1.dist_to_origin() << '\n';
  Point scale2r = p1*2.;
  cout << "scaled right: "
       << scale2r.dist_to_origin() <<
    '\n';
  // ILLEGAL Point scale2l = 2.*p1;
```

**Output**
**[geom] pointmult:**

```
p1 to origin 2.23607
scaled right: 4.47214
```

Can also:

```
void Point::operator*=(double factor);
```

**Exercise 9.15.** Write a `Fraction` class, and define the arithmetic operators on it.

Define both the + and += operators. Can you use one of them in defining the other?

**Exercise 9.16.** If you know about templates, you can do the exercises in section 22.3.

### 9.5.6.1 Functors

A special case of operator overloading is *overloading the parentheses*. This makes an object look like a function; we call this a *functor*.

Simple example:

**Code:**

```
class IntPrintFunctor {
public:
  void operator()(int x) {
    cout << x << '\n';
  }
};
  /* ... */
  IntPrintFunctor intprint;
  intprint(5);
```

**Output**
**[object] functor:**

```
5
```

**Exercise 9.17.** Extend that class as follows: instead of printing the argument directly, it should print it multiplied by a scalar. That scalar should be set in the constructor. Make the following code work:

**Code:**

```
IntPrintTimes printx2(2);
printx2(1);
for ( auto i : {5,6,7,8} )
  printx2(i);
```

**Output**

```
[object] functor2:

2
10
12
14
16
```

(The `for_each` is part of `algorithm`)

#### 9.5.6.2 Object outputting

Wouldn't it be nice if you could

```
MyObject x;
cout << x << '\n';
```

The reason this doesn't work, is that the 'double less' is a binary operator, which is not defined with your class as second operand.

See section 12.3 for the solution.

#### 9.5.6.3 Comparisons and the 'spaceship' operator

In section 9.5.6 above we discussed operator overloading. In particular, you can overload the comparison operators <, =, > et cetera. This quickly becomes a lot of work: there are six different operators.

The C++20 standards has simplified this with the *spaceship operator*.

```
bool operator<( const Point& two ) const {
  if (vx!=two.vx)
    return vx<two.vx;
  else return vy<two.vy;
}
auto operator<=>( const Point& other ) const = default;
```

### 9.5.7 Constructors and contained classes

Suppose we have a class where each object contains another object of some non-trivial class. Now we have to be aware of how the creation of the outer object relates to that of the inner.

Finally, if a class contains objects of another class,

```
class Inner {
public:
  Inner(int i) { /* ... */ }
};
class Outer {
private:
  Inner contained;
public:
```

```
};
```

then

```
Outer( int n ) {
  contained = Inner(n);
};
```

1. This first calls the default constructor
2. then calls the `Inner(n)` constructor,
3. then copies the result over the `contained` member.

```
Outer( int n )
  : contained(Inner(n)) {
  /* ... */
};
```

1. This creates the `Inner(n)` object,
2. placed it in the `contained` member,
3. does the rest of the constructor, if any.

**Remark 6** *The order of the* member initializer list *is ignored: the members specified will be initialized in the order in which they are declared. There are cases where this distinction matters, so best put both in the same order.*

### 9.5.8 'this' pointer

Inside an object, a *pointer* to the object is available as **this**:

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint; // 'this' redundant!
  };
};
```

You don't often need the **this** pointer. Example: you need to call a function inside a method that needs the object as argument)

```
/* forward definition: */ class someclass;
void somefunction(const someclass &c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(*this);
};
```

(Rare use of dereference star)

There is another interesting idiom that uses the 'this' pointer. Define a simple class

```
class number {
private:
  float x;
public:
  number(float x) : x(x) {};
```

```
  float value() { return x; };
```

Defining a method

```
  number addcopy(float y) {
    x += y;
    return *this;
  };
```

both alters the object, and returns a copy.

Changing the method to return a reference:

```
number& add(float y) {
  x += y;
  return *this;
};
number& multiply(float y) {
  x *= y;
  return *this;
};
```

has the interesting effect that no copy is created, but the returned 'object' is the object itself, by reference. This makes an interesting idiom possible:

**Code:**

```
number mynumber(1.0);
mynumber.add(.5);
cout << mynumber.value() << '\n';

mynumber.multiply(2.).add(1.).multiply(3.);
cout << mynumber.value() << '\n';
```

**Output**
**[object] this:**

```
1.5
12
```

### 9.5.9 Mutable data

Suppose you have a class and you want to return some complicated data member by const-ref:

```
  class has_stuff {
    private:
    complicated thing;
    public:
    const complicated& get_thing() const {
      return thing; };
    };
```

To make life interesting, the complicated thing should only be constructed when needed. You could try constructing it in the *get_thing* method:

```
  private:
  optional<complicated> thing = {};
  public:
  const complicated& get_thing() const {
    if (not thing.has_value())
    thing = complicated( /* current stuff */ );
    return thing.value(); };
    };
```

The problem is that the *get_thing* method now is no longer 'const'. Here you need to realize that **const** means that the routine is only outwardly constant: it can still alter internal data, if that is declared **mutable**:

```
private:
mutable optional<complicated> thing = {};
public:
const complicated& get_thing() const /* as above */
```

**Code:**

```
class has_stuff {
private:
  mutable optional<complicated> thing =
    {};
public:
  const complicated& get_thing() const {
    if ( not thing.has_value() )
      thing = complicated(5);
    else cout << "thing already there\n";
    return thing.value();
  };
};
```

**Output**
**[object] mutable:**

```
making complicated thing
thing already there
thing already there
```

## 9.6 Review questions

**Review** 9.2. Fill in the missing term

- The functionality of a class is determined by its...
- The state of an object is determined by its...

How many constructors do you need to specify in a class definition?

- Zero
- Zero or more
- One
- One or more

**Review** 9.3. Describe various ways to initialize the members of an object.

# Chapter 10

# Arrays

An *array*[1] is an indexed data structure that for each index stores an integer, floating point number, character, object, et cetera. In scientific applications, arrays often correspond to vectors and matrices, potentially of quite large size. (If you know about the Finite Element Method (FEM), you know that vectors can have sizes in the millions or beyond.)

In this chapter you will see the C++ `vector` construct, which implements the notion of an array of things, whether they be numbers, strings, objects.

> *C difference:* While C++ can use the C mechanisms for arrays, for almost all purposes it is better to use `vector`. In particular, this is a safer way to do dynamic allocation. The old mechanisms are briefly discussed in section 10.10.

## 10.1 Some simple examples

### 10.1.1 Vector creation

To use vectors, you first need the `vector` header from the STL. This allows you to declare a vector, specifying what type of element it contains. Next you may want to decide how many elements it contains; you can specify this when you declare the vector, or determine it later, dynamically.

We start with the most obvious way of creating a vector: enumerating its elements.

Short vectors can be created by enumerating their elements:

```
#include <vector>
using std::vector;

int main() {
  vector<int> evens{0,2,4,6,8};
  vector<float> halves = {0.5, 1.5, 2.5};
  auto halfloats = {0.5f, 1.5f, 2.5f};
  cout << evens.at(0) << '\n';
  return 0;
}
```

1. The term 'array' is used informally here. There is an `array` keyword, which is briefly discussed in section 10.4.

**Exercise 10.1.**

1. Take the above snippet, compile, run.
2. Add a statement that alters the value of a vector element. Check that it does what you think it does.
3. Add a vector of the same length, containing odd numbers, which are the even values plus 1?

*You can base this off the file `shortvector.cxx` in the repository*

A more sophisticated example:

```
vector<Point> diagonal =
    { {0.,0.}, {1.,1.}, {1.5,1.5}, {2.,2.}, {3.,3.} };
```

### 10.1.2 Initialization

There are various ways to declare a vector, and possibly initialize it.

More generally, vectors can be defined

- Without further specification, creating an empty vector:

  ```
  vector<float> some_numbers;
  ```

- With a size indicated, allocating that number of elements:

  ```
  vector<float> five_numbers(5);
  ```

  (This sets the elements to a default value; zero for numeric types.)
- You can initialize a vector with a constant:

  ```
  vector<float> x(25,3.15);
  ```

  which defines a vector `x` of size 25, with all elements initialized to 3.15.

If your vector is short enough, you can set all elements explicitly with an *initializer list*, and note that the size is not specified here, but is deduced from the length of the initializer list:

**Code:**

```
{
  vector<int> numbers{5,6,7,8,9,10};
  cout << numbers.at(3) << '\n';
}
{
  vector<int> numbers = {5,6,7,8,9,10};
  numbers.at(3) = 21;
  cout << numbers.at(3) << '\n';
}
```

**Output**

```
[array] dynamicinit:
8
21
```

**Review** 10.1. T/F?

- It is possible to write a valid C++ program where you define a variable `vector`.

### 10.1.3 Element access

There are two ways of accessing vector elements.

1. With the 'dot' notation that you know from structures and objects, you can use the `at` method:

**Code:**

```
vector<int> numbers = {1,4};
numbers.at(0) += 3;
numbers.at(1) = 8;
cout << numbers.at(0) << ","
     << numbers.at(1) << '\n';
```

**Output**
**[array] assignatfun:**

```
4,8
```

2. There is also a short-hand notation (which is the same as in C):

**Code:**

```
vector<int> numbers = {1,4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << ","
     << numbers[1] << '\n';
```

**Output**
**[array] assignbracket:**

```
4,8
```

Indexing starts at zero. Consequently, a vector declared as

```
vector<int> ints(N)
```

has elements $0, \ldots, N-1$.

As you see in this example, if `a` is a vector, and `i` an integer, then `a.at(i)` is the i'th element.

- The expression `a.at(i)` can be used to get the value of a vector element, or it can occur in the left-hand side of an assignment to set the value

```
vector<float> x(25);
x.at(2) = 3.14;
float y = y.at(2);
```

- The same holds for indexing with square brackets.

```
vector<float> x(25);
x[2] = 3.14;
float y = y[2];
```

- The *vector index* (or *vector subscript*) `i` starts numbering at zero.
- Therefore, if a vector has $n$ elements, its last element has index `n-1`.

### 10.1.4 Access out of bounds

Have you wondered what happens if you access a vector element outside the bounds of the vector?

```
vector<float> x(6); // size 6, index ranges 0..5
x.at(6) = 5.; // oops!
i = -2;
x[i] = 3; // also oops, but different.
```

Usually, it is hard for the compiler to determine that you are accessing an element outside the vector bounds. Most likely, it will only be detected at runtime. There is now a difference in how the two accessing methods do *vector bounds checking*.

1. Using the `at` method will always do a bounds test, and exit your program immediately if you access an element outside the vector bounds. (Technically, it throws an *exception*; see section 23.2.2 for how this works and how you can handle this.)
2. The bracket notation `a[i]` performs no bounds tests: it calculates a memory address based on the vector location and the index, and attempts to return what is there. As you may imagine, this lack of checking makes your code a little faster. However, it also makes your code unsafe:
   - Your program may crash with a *segmentation fault* or *bus error*, but no clear indication where and why this happened. (Such a crash can be caused by other things than vector access out of bounds.)
   - Your program may continue running, but giving wrong results, since reading from outside the vector probably gives you meaningless values. Writing outside the bounds of an vector may even change the data of other variables, leading to really strange errors.

For now, it is best to use the `at` method throughout.

Indexing out of bounds can go undetected for a while:

**Code:**

```
vector<float> v(10,2);
for (int i=5; i<6; i--)
  cout << "element " << i
       << " is " << v[i] << '\n';
```

**Output**

```
[array] segmentation:

element -5869 is 0
element -5870 is 2.8026e-45
element -5871 is 2.38221e-43
element -5872 is 1.00893e-41
element -5873 is 0
element -5874 is 0
element -5875 is 0
element -5876 is 0
/bin/sh: line 1: 48082
    Segmentation fault: 11
    (core dumped) ./segmentation
```

**Review** 10.2. The following codes are not correct in some sense. How will this manifest itself?

```
vector<int> a(5);
a[6] = 1.;
```

```
vector<int> a(5);
a.at(6) = 1.;
```

In some applications you will create an vector, and gradually fill it, for instance in a loop. However, sometimes your elements are known in advance and you can write them out. Specifying these values while creating the vector is called *vector initialization*, and there is more than one way to do so.

First of all, you can set a vector to a constant:

## 10.2 Going over all vector elements

If you need to consider all the elements in a vector, you typically use a **for** loop. There are various ways of doing this.

Conceptually, a *vector* can correspond to a set of things, and the fact that they are indexed is purely incidental, or it can correspond to an ordered set, and the index is essential. If your algorithm requires you to access all elements, it is important to think about which of these cases apply, since there are two different mechanism.

### 10.2.1 Ranging over a vector

First of all consider the cases where you consider the vector as a collection of elements, and the loop functions like a mathematical 'for all'.

You can write a *range-based for* loop, which considers the elements as a collection.

```
vector<float> my_data /* create */;
for ( float e : my_data )
  // statement about element e
for ( auto e : my_data )
  // same, with type deduced by compiler
```

**Code:**

```
vector<int> numbers = {1,4,2,6,5};
int tmp_max = -2000000000;
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max
     << " (should be 6)" << '\n';
```

**Output**
**[array] dynamicmax:**

```
Max: 6 (should be 6)
```

(You can spell out the type of the vector element, but such type specifications can be complex. In that case, using *type deduction* through the **auto** keyword is quite convenient.)

So-called *initializer list*s can also be used as a list denotation:

**Code:**

```
for ( auto i : {2,3,5,7,9} )
  cout << i << ",";
cout << '\n';
```

**Output**
**[array] rangedenote:**

```
2,3,5,7,9,
```

### 10.2.2 Ranging over the indices

If you actually need the index of the element, you can use a traditional **for** loop with loop variable.

You can write an *indexed for* loop, which uses an index variable that ranges from the first to the last element.

```
for (int i= /* from first to last index */ )
  // statement about index i
```

Example: find the maximum element in the vector, and where it occurs.

**Code:**

```
int tmp_idx = 0;
int tmp_max = numbers.at(tmp_idx);
for (int i=0; i<numbers.size(); i++) {
  int v = numbers.at(i);
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << '\n';
```

**Output**

```
[array] vecidxmax:
Max: 6.6 at index: 3
```

**Exercise 10.2.** Indicate for each of the following vector operations whether you prefer to use an indexed loop or a range-based loop. Give a short motivation.

- Count how many elements of a vector are zero.
- Find the location of the last zero.

**Exercise 10.3.** Find the element with maximum absolute value in a vector. Use:

```
vector<int> numbers = {1,-4,2,-6,5};
```

Hint:

```
#include <cmath>
..
absx = abs(x);
```

**Exercise 10.4.** Find the location of the first negative element in a vector.

Which mechanism do you use?

**Exercise 10.5.** Check whether a vector is sorted.

### 10.2.3 Ranging by reference

Range-based loop indexing makes a copy of the vector element. If you want to alter the vector, use a reference:

```
for ( auto &e : my_vector)
  e = ....
```

**Code:**

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2) << '\n';
```

**Output**
**[array] vectorrangeref:**

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

**Exercise 10.6.** If you do the prime numbers project, you can now do exercise 46.15.

In a `while` loop, if you need an index, you need to maintain that index explicitly. There are then certain common idioms.

**Code:**

```
    vector<int> numbers{3,5,7,8,9,11};
    int index{0};
    while ( numbers[index++]%2==1 ) ;
    cout << "The first even number\n"
         << "appears at index "
         << index << '\n';
```

**Output**
**[loop] plusplus:**

```
The first even number
appears at index 4
```

**Exercise 10.7.** Exercise: modify the preceding code so that after the while loop `index` is the number of leading odd elements.

## 10.3 Vector are a class

Above, you created vectors and used functions `at` and `size` on them. They used the dot-notation of class methods, and in fact vector form a `vector` class. You can have a vector of ints, floats, doubles, et cetera; the angle bracket notation indicates what the specific type stored in the vector is. You could say that the vector class is parameterized with the type (see chapter 22 for the details). We could say that `vector<int>` is a new data type, pronounced 'vector-of-int', and you can make variables of that type.

Vectors can be copied just like other datatypes:

**Code:**

```
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
     << vcopy.at(2) << '\n';
```

**Output**
**[array] vectorcopy:**

```
3.5,7
```

### 10.3.1 Vector methods

There are several *methods* to the `vector` class. Some of the simpler ones are:

- `at`: index an element
- `size`: give the size of the vector
- `front`: first element
- `back`: last element

There are also methods relating to dynamic storage management, which we will get to next.

**Exercise 10.8.** Create a `vector` $x$ of `float` elements, and set them to random values. (Use the C random number generator for now.)

Now normalize the vector in $L_2$ norm and check the correctness of your calculation, that is,

1. Compute the $L_2$ norm of the vector:

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. Divide each element by that norm;
3. The norm of the scaled vector should now by 1. Check this.
4. Bonus: your program may be printing 1, but is it actually 1? Investigate.

What type of loop are you using?

`vector` is a 'templated class': `vector<X>` is a vector-of-`X`.

Code behaves as if there is a class definition for each type:

```
class vector<int> {
public:
  size(); at(); // stuff
}
```

```
class vector<float> {
public:
  size(); at(); // stuff
}
```

Actual mechanism uses templating: the type is a parameter to the class definition. More later.

### 10.3.2 Vectors are dynamic

A vector can be grown or shrunk after its creation. For instance, you can use the `push_back` method to add elements at the end.

Extend a vector's size with `push_back`:

**Code:**

```
vector<int> mydata(5,2);
mydata.push_back(35);
cout << mydata.size() << '\n';
cout << mydata.back();
     << '\n';
```

**Output**

```
[array] vectorend:

6
35
```

Similar functions: `pop_back`, `insert`, `erase`. Flexibility comes with a price.

It is tempting to use `push_back` to create a vector dynamically.

Known vector size:

```
int n = get_inputsize();
vector<float> data(n);
for ( int i=0; i<n; i++ ) {
  auto x = get_item(i);
  data.at(i) = x;
}
```

Unknown vector size:

```
vector<float> data;
float x;
while ( next_item(x) ) {
  data.push_back(x);
}
```

If you have a guess as to size: `data.reserve(n)`.

(Issue with array-of-object: in left code, constructors are called twice.)

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

However, this dynamic resizing involves memory management, and maybe operating system functions. This will probably be inefficient. Therefore you should use such dynamic mechanisms only when strictly necessary. If you know the size, create a vector with that size. If the size is not precisely known but you have a reasonable upper bound, you can call `reserve` to reserve space for that many elements:

```
vector<int> iarray;
iarray.reserve(100);
while ( ... )
  iarray.push_back( ... );
```

The combination of using `reserve` and `push_back` can be preferable over creating the vector immediately with a certain size. Writing `vector<X> xs(100)`, where `X` is some object, causes the default constructor of `X` to be called on each vector element. For complicated objects this may not be advisable.

## 10.4 The Array class

In cases where an array will never change size it would be convenient to have a variant of the `vector` class that does not have the dynamic memory management facility. The `array` class seems to fulfill this role at first sight. However, it is limited to arrays where the size is known at compile time, so you can not for instance read it in as a parameter.

```
#include <array>
using std::array;
```

Array objects are declared as:

```
array<float,3> coordinate;
```

```
{
  array<float,5> v5;
  cout << "size: " << v5.size() << '\n';
  // WRONG: such function
  // v5.push_back(2);
}
```

### 10.4.1 Initialization

There are several ways to initialize a `std::array`. The most literal-minded way is

```
array<int,3> i3 = {1,2,3};
// or
array<int,3> i3 { {1,2,3} };
```

but as of C++14 *aggregate initialization* is allowed:

```
array<int,3> i3{1,2,3};
```

If it bothers you that the size of the array is redundant in an initialization, you can use C++17 *template argument deduction*:

```
array i3 = {1,2,3};
```

This does require you to be careful with the types:

```
// DOES NOT COMPILE:
array not4{1.5,2,3,4};
```

## 10.5 Vectors and functions

Vectors act like any other datatype, so they can be used with functions: you can pass a vector as argument, or have it as return type. We will explore that in this section.

### 10.5.1 Pass vector to function

The mechanisms of parameters passing (section 7.5) apply to vectors too: they can be passed by value and by reference.

First of all, there is passing by value; section 7.5.1. Here, the vector argument is copied to the function; the function receives a full copy of the vector, and any changes to that vector in the function do not affect the calling environment.

> *C difference:* There is a big difference here between C++ vectors and C arrays! In C the array is not copied: you pass the address by value. Not the contents.

**Code:**

```
void set0
  ( vector<float> v,float x )
{
  v.at(0) = x;
}
  /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << '\n';
```

**Output**
**[array] vectorpassnot:**

```
3.5
```

- Vector is copied
- 'Original' in the calling environment not affected
- Cost of copying?

**Exercise 10.9.** Revisit exercise 10.8 and introduce a function for computing the $L_2$ norm.

Next, there is passing by reference; section 7.5.2. Here, the parameter vector becomes alias to the vector in the calling environment, so changes to the vector in the function affect the argument vector in the calling environment.

**Code:**

```
void set0
  ( vector<float> &v,float x )
{
  v.at(0) = x;
}
  /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << '\n';
```

**Output**
**[array] vectorpassref:**

```
4.6
```

An important reason for wanting to pass by reference is that it avoids the possibly substantial cost in copying the argument in passing by value. So what if you want that efficiency, but you like to safeguard yourself against inadvertent changes to the argument vector? For this, you can declare the function parameter as 'const reference'.

Passing a vector that does not need to be altered:

```
int f( const vector<int> &ivec ) { ... }
```

- Zero copying cost
- Not alterable, so: safe!
- (No need for pointers!)

The general guideline for parameter passing was

- pass by value if the argument is not altered;
- pass by reference if the argument is altered.

For vectors this matter gets another dimension: passing by value means copying, which is potentially expensive for vectors. The way out here is to pass by *const reference* which both prevents copying and prevents accidental altering; see section 18.2.

### 10.5.2 Vector as function return

You can have a vector as return type of a function.
Example: this function creates a vector, with the first element set to the size:

**Code:**

```
vector<int> make_vector(int n) {
  vector<int> x(n);
  x.at(0) = n;
  return x;
}
  /* ... */
  vector<int> x1 = make_vector(10);
  // "auto" also possible!
  cout << "x1 size: " << x1.size() <<
    '\n';
  cout << "zero element check: " <<
    x1.at(0) << '\n';
```

**Output**

```
[array] vectorreturn:

x1 size: 10
zero element check: 10
```

**Exercise 10.10.** Write a function of one `int` argument $n$, which returns vector of length $n$, and which contains the first $n$ squares.

**Exercise 10.11.** Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_vector(length);
vector<float> sorted = sort(values);
```

This creates a vector of random values of a specified length, and then makes a sorted copy of it.

Instead of making a sorted copy, sort in-place
(overwrite original data with sorted data):

```
int length = 10;
vector<float> values = random_vector(length);
sort(values); // the vector is now sorted
```

Find arguments for/against that approach.

(Note: C++ has sorting functions built in.)

(See section 24.6.4 for the random fuction.)

**Exercise 10.12.** Write code to take a vector of integers, and construct two vectors, one containing all the odd inputs, and one containing all the even inputs. So:

```
input:
   5,6,2,4,5
output:
   5,5
   6,2,4
```

Can you write a function that accepts a vector and produces two vectors as described?

## 10.6 Vectors in classes

You may want a class of objects that contain a vector. For instance, you may want to name your vectors.

```
class named_field {
private:
  vector<double> values;
  string name;
```

The problem here is when and how that vector is going to be created.

- If the size of the vector is statically determined, you can of course declare it with that size:

  ```
  class named_field {
  private:
    vector<double> values(25);
    string name;
  ```

- ... but in the more interesting case the size is determined during the runtime of the program. In that case you would to declare:

  ```
  named_field velocity_field(25,"velocity");
  ```

  specifying the size in the constructor of the object.

So now the question is, how do you allocate that vector in the object constructor?

One solution would be to specify a vector without size in the class definition, create a vector in the constructor, and assign that to the vector member of the object:

```
named_field( int n ) {
  values = vector<int>(n);
};
```

However, this has the effect that

- The constructor first creates `values` as a zero size vector,
- then it creates an anonymous vector of size `n`,
- and by assigning it, destroys the earlier created zero size vector.

This is somewhat inefficient, and the optimal solution is to create the vector as part of the *member initializer* list:

Use initializers for creating the contained vector:

```
1 class named_field {
2 private:
3   string name;
4   vector<double> values;
5 public:
```

```
  named_field( string name,int n )
    : name(name),
      values(vector<double>(n)) {
  };
};
```

Less desirable method is creating in the constructor:

```
  named_field( string uname,int n ) {
    name = uname;
    values = vector<double>(n);
  };
```

### 10.6.1 Timing

Different ways of accessing a vector can have drastically different timing cost.

You can push elements into a vector:

```
  vector<int> flex;
/* ... */
  for (int i=0; i<LENGTH; i++)
    flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
  vector<int> stat(LENGTH);
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat.at(i) = i;
```

With subscript:

```
  vector<int> stat(LENGTH);
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

You can also use **new** to allocate*:

```
  int *stat = new int[LENGTH];
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

*Considered bad practice. Do not use.

For **new**, see section 17.6.2. However, note that this mode of allocation is basically never needed.

Timings are partly predictable, partly surprising:

```
  Flexible time: 2.445
  Static at time: 1.177
```

```
Static assign time: 0.334
Static assign time to new: 0.467
```

The increased time for `new` is a mystery.

So do you use `at` for safety or `[]` for speed? Well, you could use `at` during development of the code, and insert

```
#define at(x) operator[](x)
```

for production.

## 10.7 Wrapping a vector in an object

You may want to a create objects that contain a vector, for instance because you want to add some methods.

```
class namedvector {
private:
  string name;
  vector<int> values;
public:
  namedvector(int n,string name="unnamed")
    : name(name),values(vector<int>(n)) {
  };
  string rendered() {
    stringstream render;
    render << name << ":";
    for (auto v : values )
      render <<  " " << v << ",";
    return render.str();
  }
  /* ... */
};
```

Unfortunately this means you may have to recreate some methods:

```
int &at(int i) {
  return values.at(i);
};
```

## 10.8 Multi-dimensional cases

Unlike Fortran, C++ has little support for multi-dimensional arrays. If your multi-dimensional arrays are to model linear algebra objects, it would be good to check out the *Eigen* library. Here are some general remarks on multi-dimensional storage.

### 10.8.1 Matrix as vector of vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row);
```

Create a row vector, then store 10 copies of that:
vector of vectors.

This is not the best implementation of a matrix, for instance because the elements are not contiguous. However, let's continue with it for a moment.

**Remark 7** *More flexible strategy:*

```
// alternative:
vector<vector<float>> rows(10);
for ( auto &row : rows )
  row = vector<float>(20);
```

```
class matrix {
private:
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
```

**Exercise 10.13.** Write `rows()` and `cols()` methods for this class that return the number of rows and columns respectively.

**Exercise 10.14.** Write a method `void set(double)` that sets all matrix elements to the same value.

Write a method `double totalsum()` that returns the sum of all elements.

**Code:**

```
A.set(3.);
cout << "Sum of elements: "
     << A.totalsum() << '\n';
```

**Output**

```
[array] matrixsum:
Sum of elements: 30
```

*You can base this off the file `matrix.cxx` in the repository*

**Exercise 10.15.** Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

### 10.8.2 A better matrix class

You can make a 'pretend' matrix by storing a long enough `vector` in an object:

```
class matrix {
private:
  std::vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n)
    : m(m),n(n),the_matrix(m*n) {};
  void set(int i,int j,double v) {
    the_matrix.at( i*n +j ) = v;
  };
  double get(int i,int j) {
    return the_matrix.at( i*n +j );
  };
  /* ... */
};
```

**Exercise 10.16.** In the matrix class of the previous slide, why are `m,n` stored explicitly, and not in the previous case?

The most important advantage of this is that it is compatible with the storage traditionally used in many libraries and codes.

The syntax for `set` and `get` can be improved.

**Exercise 10.17.** Write a method `element` of type `double&`, so that you can write

```
A.element(2,3) = 7.24;
```

## 10.9 Advanced topics

### 10.9.1 Container copying

Using the *copy constructor* on containers such as vectors invokes the copy constructor of each individual element. Types such as `float` are called 'trivially constructible' or 'trivially copyable', and they are optimized for: copying a `vector<int>` is do by a `memcpy` or equivalent mechanism.

### 10.9.2 Failed allocation

If you ask for more data than your system can support, the allocation may fail, and a `bad_alloc` exception is thrown.

This is unlike the approach in C of returning `NULL` or `nullptr`.

### 10.9.3 Stack and heap allocation

Entities stored in memory (variables, structures, objects) can exist in two locations: the *stack* and the *heap*.

- Every time a program enters a new *scope*, entities declared there are placed on top of the stack, and they are removed by the end of the scope. Because of this automatic behavior, this is known as *automatic allocation*.
- By contrast, *dynamic allocation* creates a memory block that is not removed at the end of the scope, and so this block is placed on the heap. That block of memory can be returned to the free store at any time, so the heap can suffer from *fragmentation*.

#### *10.9.3.1 Illustrations in C*

Automatic memory allocation, or the allocation of static memory, uses scopes, just it it does for the creation of scalars:

```
// assume there are no variables i,f,str here
{ // enter scope
  int i;
  float f;
  char str[5];
  // stuff
}
// the names i,f,str are unknown again here.
```

Objects that obey scope are allocated on the stack, so that their memory is automatically freed control leaves the scope. On the other hand, overuse of automatic allocation may lead to *stack overflow*.

Dynamic memory allocation was done by a call to `malloc`, and by assigning the returned memory address to a variable that was defined outside the scope, the block is known outside the scope:

```
double *array;
{ // enter scope
  array = malloc(5*sizeof(double));
  // exit scope
}
array[4] = 1.5; // this is legal
free(array); // release the malloc'ed memory
```

Dynamically created objects, such as the target of a pointer, live on the heap because their lifetime is not subject to scope.

The existence of the second category is a source of *memory leak*s, since it's too easy to forget the `free` call.

#### *10.9.3.2 Illustrations in C++*

First of all, the `malloc` and `free` calls exist in C++, as do slightly more convenient variants **`new`**/**`delete`**:

```
double *array = new[5];
// stuff
delete array;
```

However, the idiomatic C++ way to create arrays with dynamically determined memory is by using **`std`**`::vector`.

```
int n = .... ;
{ // enter scope
  vector<int> array(n);
  // stuff
} // exit scope
```

This combines the best features of C allocation:

1. The storage for the vector is created on the heap, so you need not worry about stack overflow;
2. exiting the scope, both the definition of the vector goes away, and its dynamic memory is freed. This is technically known as *RAII*.

However, dynamically allocated memory can transcend the scope it's created in:

```
vector<int> f(int n) {
  return vector<int>(n); // vector created inside function scope
};
vector<int> v;
v = f(5);
```

What happens here is the following:

1. A vector is created inside the function;
2. the **return** statement copies the vector, with all its data, to the variable *v* in the calling environment;
3. but by an optimization, the copy is omitted, and the actual memory is now assigned to the variable *v*.

In effect, we have now achieved a safer version of the function example of the above C section.

Another option for dynamic memory that is not scope-bound is to use the *smart pointer* mechanism, which also guarantees against memory leaks. See chapter 16.

### 10.9.4 Vector of bool

Booleans variables take a whole byte, even though a boolean strictly only needs a bit. However, you could optimize an array of bits, and thereby *vector*<**bool**>, by packing the bits into an integer, giving a factor of 8 savings in space.

Unfortunately, this optimization means that you can not get a reference to the elements.

```
vector<bool> bits;
for ( auto& b : bits ) // DOES NOT COMPILE
  b = false;
auto& f = bits.front(); // DOES NOT COMPILE
```

### 10.9.5 Span

The old C style arrays allowed for some operations that are harder to do with vectors. For instance, you could create a subset of an array with:

```
double *x = (double*)malloc(N*sizeof(double));
double *subx = x+1;
subx[1] = 5.; // same as: x[2] = 5.;
```

In C++ you can write

```
vector<double> x(N);
vector<double> subx( x.begin()+1,x.end() );
```

but that allocates new storage.

If you really want two vector-like objects to share data there is the `span` class, which is in the STL of C++20.

A span is little more than a pointer and a size, so it allows for the above use case. Also, it does not have the overhead of creating a whole new vector.

```
vector<double> v;
auto v_span = gsl::span<double>( v.data(),v.size() );
```

The `span` object has the same `at`, `data`, and `size` methods, and you can iterate over it, but it has no dynamic methods.

In C++23 there is `mdspan`, a multi-dimensional span.

#### *10.9.5.1 Installing span before C++20*

```
clone the repo:
git clone https://github.com/martinmoene/gsl-lite.git

add to your compile line
-I${HOME}/Installation/gsl/gsl-lite/include (or whatever the path is to yo

in your source:
#include "gsl/gsl-lite.hpp"
using gsl::span;
```

### 10.9.6 Size and signedness

The `size` method returns an unsigned quantity, so with a sufficiently high warning level

```
for (int i=0; i<myarray.size(); i++ )
```

will complain about mixing signed and unsigned quantities.

You can either

```
for (size_t i=0; i<myarray.size(); i++ )
```

or in C++20 use the `ssize` method, which returns a signed size:

```
for (int i=0; i<myarray.ssize(); i++ )
```

https://stackoverflow.com/questions/56217283/why-is-stdssize-introduced-in-c20#56217338

## 10.10 C style arrays

Static arrays are really an abuse of the equivalence of arrays and addresses of the C programming language. This appears for instance in parameter passing mechanisms.

For small arrays you can use a different syntax.

**Code:**

```
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << '\n';
}
{
  int numbers[5]{5,4,3,2,1};
  numbers[3] = 21;
  cout << numbers[3] << '\n';
}
```

**Output**
**[array] staticinit:**

```
2
21
```

This has the (minimal) advantage of not having the overhead of a class mechanism. On the other hand, it has a number of disadvantages:

- You can not query the size of an array by its name: you have to store that information separately in a variable.
- Passing such an array to a function is really passing the address of its first element, so it is always (sort of) by reference.

### 10.10.1 Allocation

Traditionally, C arrays could only be allocated as

```
int a[5];
float b[6][7];
```

that is, with explicitly given array bounds. Some compilers supported as an extension so-called *variable length arrays*:

```
int n; scanf("%d",&n); // this reads n from the console
double x[n];
```

This mechanism was addded to the C99 standard, but since support of it was not universal, the C11 standard made them optional again. The macro `__STC_NO_VLA__` is set to `1` if such support is indeed lacking.

Another thing to be aware of is that these arrays are allocated on the *stack*, so creating a too-large array may give stack overflow. This will make your code bomb with no informative error.

### 10.10.2 Indexing and range-based loops

Range-based indexing works the same as with vectors:

**Code:**

```
int numbers[] = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should
    be 6)" << '\n';
```

**Output**
**[array] rangemax:**

```
Max: 6 (should be 6)
```

**Review** 10.3. The following codes are not correct in some sense. How will this manifest itself?

```
int a[5];
a[6] = 1.;

int a[5];
a.at(6) = 1.;
```

### 10.10.3 C-style arrays and subprograms

Arrays can be passed to a subprogram, but the bound is unknown there.

```
void set_array( double *x,int size) {
  for (int i=0; i<size; i++)
    x[i] = 1.41;
};
  /* ... */
  double array[5] = {11,22,33,44,55};
  set_array(array,5);
  cout << array[0] << "...." << array[4] << '\n';
```

**Exercise 10.18.** Rewrite the above exercises where the sorting tester or the maximum finder is in a subprogram.

Unlike with scalar arguments, array arguments can be altered by a subprogram: it is as if the array is always passed by reference. This is not strictly true: what happens is that the address of the first element of the array is passed. Thus we are really dealing with pass by value, but it is the array address that is passed rather than its value.

In subprograms, such static arrays are indistinguishable from pointers. This is known as *pointer decay*. The following code and error message illustrates this:

**Code:**

```
void std_f( int stat[] ) {
  printf(".. in function:
    %lu\n",std::size(stat));
}
//codesnippet  end
  /* ... */
  int stat[23];
  std_f( stat );
```

**Output**
**[array] carray:**

```
carray.cxx: In function
    'void std_f(int*)':
carray.cxx:18:43: error: no
    matching function for
    call to 'size(int*&)'
   18 |   printf(".. in
    function:
    %lu\n",std::size(stat));
       |

    ~~~~~~~~~~~~~~~
```

### 10.10.4 Size of arrays

What does the **sizeof** operator give on various types of arrays?

**Code:**

```
int a1[10];
cout << "static: " << sizeof(a1) <<
    '\n';
int *a2 = (int*) malloc( 10*sizeof(int)
    );
cout << "malloc: " << sizeof(a2) <<
    '\n';
vector<int> a3(10);
cout << "vector: " << sizeof(a3) <<
    '\n';
```

**Output**
**[array] staticsize:**

```
static: 40
malloc: 8
vector: 24
```

You may think that **sizeof** on a static array is useful, but that doesn't survive passing to a subprogram:

**Code:**

```
void stat_f( int stat[] ) {
  printf(".. in function:
    %lu\n",sizeof(stat));
}
//codesnippet
```

**Output**
**[c] carraystat:**

```
carray.c:16:40: warning:
    sizeof on array function
    parameter will return
    size of 'int *' instead
    of 'int []'
    [-Wsizeof-array-argument]
  printf(".. in function:
    %lu\n",sizeof(stat));

                ^
carray.c:15:18: note:
    declared here
void stat_f( int stat[] ) {
                  ^
1 warning generated.
Size of stat[23]: 92
.. in function: 8
```

Note the compiler warning

```
warning: sizeof on array function parameter will return size of 'int *' in
```

### 10.10.5 Multi-dimensional arrays

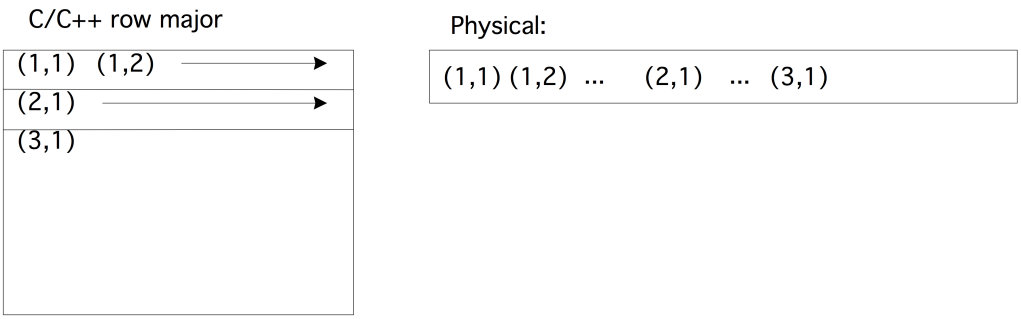
Multi-dimensional arrays can be declared and used with a simple extension of the prior syntax:

```
float matrix[15][25];

for (int i=0; i<15; i++)
  for (int j=0; j<25; j++)
    // something with matrix[i][j]
```

Passing a multi-dimensional array to a function, only the first dimension can be left unspecified:

```
void print12( int ar[][6] ) {
  cout << "Array[1][2]: " << ar[1][2] << '\n';
  return;
}
  /* ... */
    int array[5][6];
    array[1][2] = 3;
    print12(array);
```




### 10.10.6 Memory layout

Puzzling aspects of arrays, such as which dimensions need to be specified and which not in a function call, can be understood by considering how arrays are stored in memory. The question then is how a two-dimensional (or higher dimensional) array is mapped to memory, which is linear.

- A one-dimensional array is stored in contiguous memory.
- A two-dimensional array is also stored contiguously, with first the first row, then the second, et cetera.
- Higher dimensional arrays continue this notion, with contiguous blocks of the highest so many dimensions.

As a result of this, indexing beyond the end of a row, brings you to the start of the next row:

```
void print06( int ar[][6] ) {
  cout << "Array[0][6]: " << ar[0][6] << '\n';
  return;
```

```
}
/* ... */
  int array[5][6];
  array[1][0] = 35;
  print06(array);
```

We can now also understand how arrays are passed to functions:

- The only information passed to a function is the address of the first element of the array;
- In order to be able to find location of the second row (and third, et cetera), the subprogram needs to know the length of each row.
- In the higher dimensional case, the subprogram needs to know the size of all dimensions except for the first one.

## 10.11 Exercises

**Exercise 10.19.** Given a vector of integers, write two loops;

1. One that sums all even elements, and
2. one that sums all elements with even indices.

Use the right type of loop.

**Exercise 10.20.** Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

*Pascal's triangle* contains binomial coefficients:

```
Row    1:                      1
Row    2:                    1   1
Row    3:                  1   2   1
Row    4:                1   3   3   1
Row    5:              1   4   6   4   1
Row    6:            1   5  10  10   5   1
Row    7:          1   6  15  20  15   6   1
Row    8:        1   7  21  35  35  21   7   1
Row    9:      1   8  28  56  70  56  28   8   1
Row   10:    1   9  36  84 126 126  84  36   9   1
```

where

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

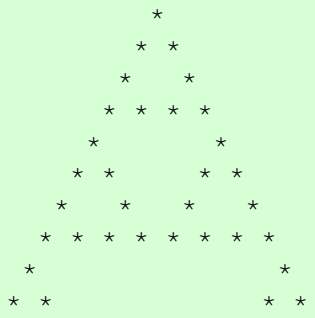
The coefficients can be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} & \end{cases}$$

(There are other formulas. Why are they less preferable?)

**Exercise 10.21.**

- Write a class `pascal` so that `pascal(n)` is the object containing $n$ rows of the above coefficients. Write a method `get(i,j)` that returns the $(i, j)$ coefficient.
- Write a method `print` that prints the above display.
- First print out the whole pascal triangle; then:
- Write a method `print(int m)` that prints a star if the coefficient modulo $m$ is nonzero, and a space otherwise.



- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.
- Your program should accept:
  1. an integer for the size
  2. any number of integers for the modulo; if this is zero, stop, otherwise print stars as described above.

**Exercise 10.22.** Extend the Pascal exercise:
Optimize your code to use precisely enough space for the coefficients.

**Exercise 10.23.** A knight on the chess board moves by going two steps horizontally or vertically, and one step either way in the orthogonal direction. Given a starting position, find a sequence of moves that brings a knight back to its starting position. Are there starting positions for which such a cycle doesn't exist?

**Exercise 10.24.** From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

**Exercise 10.25.** Revisit exercise 6.13, and generate the 'Collatz tree' (figure 10.1): at level $n$ (one-based counting) are the numbers that in $n - 1$ steps converge to 1.

Read in a number $n$ and print the first $n$ rows, each row on a new line, with the numbers separated by spaces.

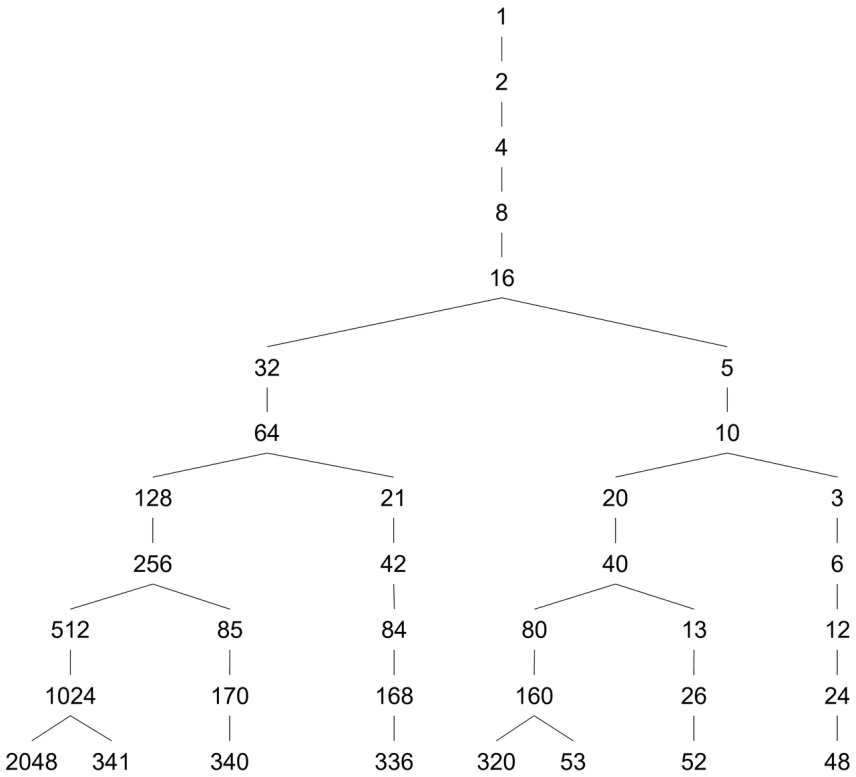



Figure 10.1: The ‘Collatz tree’

# Chapter 11

# Strings

## 11.1 Characters

- Type `char`;
- represents '7-bit ASCII': printable and (some) unprintable characters.
- Single quotes: `char c = 'a'`

Equivalent to (short) integer:

**Code:**

```
char ex = 'x';
int x_num = ex, y_num = ex+1;
char why = y_num;
cout << "x is at position " << x_num
     << '\n';
cout << "one further lies " << why
     << '\n';
```

**Output**
**[string] intchar:**

```
x is at position 120
one further lies y
```

Also: `'x'-'a'` is distance `a--x`

**Remark 8** *The translation from `'x'` to ascii code, and in particular the letters having consecutive values, are not guaranteed by the standard.*

**Exercise 11.1.** Write a program that accepts an integer $1 \cdots 26$ and prints the so-manieth letter of the alphabet.

Extend your program so that if the input is negative, it prints the minus-so-manieth uppercase letter of the alphabet.

## 11.2 Basic string stuff

```
#include <string>
using std::string;
```

```
// .. and now you can use 'string'
```

(Do not use the C legacy mechanisms.)

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

Normally, quotes indicate the start and end of a string. So what if you want a string with quotes in it?

You can escape a quote, or indicate that the whole string is to be taken literally:

**Code:**

```
string
  one("a b c"),
  two("a \"b\" c"),
  three( R"("a ""b """c)" );
cout << one << '\n';
cout << two << '\n';
cout << three << '\n';
```

**Output**
**[string] quote:**

```
a b c
a "b" c
"a ""b """c
```

Strings can be *concatenated*:

**Code:**

```
string my_string, space{" "};
my_string = "foo";
my_string += space + "bar";
cout << my_string << ": " <<
    my_string.size() << '\n';
```

**Output**
**[string] stringadd:**

```
foo bar: 7
```

You can query the *size*:

**Code:**

```
string five_text{"fiver"};
cout << five_text.size() << '\n';
```

**Output**
**[string] stringsize:**

```
5
```

or use subscripts:

**Code:**

```
string digits{"0123456789"};
cout << "char three: "
     << digits[2] << '\n';
cout << "char four : "
     << digits.at(3) << '\n';
```

**Output**
**[string] stringsub:**

```
char three: 2
char four : 3
```

Same as ranging over vectors.

Range-based for:

**Code:**

```
cout << "By character: ";
for ( char c : abc )
  cout << c << " ";
cout << '\n';
```

**Output**
**[string] stringrange:**

```
By character: a b c
```

Ranging by index:

**Code:**

```
string abc = "abc";
cout << "By character: ";
for (int ic=0; ic<abc.size(); ic++)
  cout << abc[ic] << " ";
cout << '\n';
```

**Output**
**[string] stringindex:**

```
By character: a b c
```

Range-based for makes a copy of the element
You can also get a reference:

**Code:**

```
for ( char &c : abc )
  c += 1;
cout << "Shifted: " << abc << '\n';
```

**Output**
**[string] stringrangeset:**

```
Shifted: bcd
```

```
for ( auto c : some_string)
  // do something with the character 'c'
```

**Review** 11.1. True or false?

1. `'0'` is a valid value for a `char` variable
2. `"0"` is a valid value for a `char` variable
3. `"0"` is a valid value for a `string` variable
4. `'a'+'b'` is a valid value for a `char` variable

**Exercise 11.2.** The oldest method of writing secret messages is the *Caesar cipher*. You would take

an integer $s$ and rotate every character of the text over that many positions:

$$s \equiv 3\text{: "acdz"} \Rightarrow \text{"dfgc".}$$

Write a program that accepts an integer and a string, and display the original string rotated over that many positions.

**Exercise 11.3.** (this continues exercise 11.2)
If you find a message encrypted with the Caesar cipher, can you decrypt it? Take your inspiration from the *Sherlock Holmes* story 'The Adventure of the Dancing Men', where he uses the fact that 'e' is the most common letter.

Can you implement a more general letter permutation cipher, and break it with the 'dancing men' approach?

Other methods for the vector class apply: `insert`, `empty`, `erase`, `push_back`, et cetera.

**Code:**

```
string five_chars;
cout << five_chars.size() << '\n';
for (int i=0; i<5; i++)
  five_chars.push_back(' ');
cout << five_chars.size() << '\n';
```

**Output**
**[string] stringpush:**

```
0
5
```

Methods only for `string`: `find` and such.

http://en.cppreference.com/w/cpp/string/basic_string

**Exercise 11.4.** Write a function to print out the digits of a number: `156` should print `one five six`. You need to convert a digit to a string first; can you think of more than one way to do that?

Start by writing a program that reads a single digit and prints its name.

For the full program it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

**Exercise 11.5.** Write a function to convert an integer to a string: the input `215` should give `two hundred fifteen`, et cetera.

**Exercise 11.6.** Write a pattern matcher, where a period `.` matches any one character, and `x*` matches any number of 'x' characters.

For example:

- The string `abc` matches `a.c` but `abbc` doesn't.
- The string `abbc` matches `ab*c`, as does `ac`, but `abzbc` doesn't.

## 11.3 String streams

You can concatenate string with the + operator. The less-less operator also does a sort of concatenation. It is attractive because it does conversion from quantities to string. Sometimes you may want a combination

of these facilities: conversion to string, with a string as result.

For this you can use a *string stream* from the `sstream` header.

Like cout (including conversion from quantity to string), but to object, not to screen.

- Use the `<<` operator to build it up; then
- use the `str` method to extract the string.

```
#include <sstream>
stringstream s;
s << "text" << 1.5;
cout << s.str() << endl;
```

## 11.4 Advanced topics

### 11.4.1 Raw string literals

You can include characters such as quotes or backslashes in a string by escaping them. This may get tiresome. The C++11 standard has a mechanism for *raw string literal*s.

In its simplest form:

**Code:**

```
    cout << R"(string with }

{ \weird\ stuf)" << '\n';
```

**Output**
**[string] raw1:**

```
string with }

{ \weird\ stuf
```

The obvious question is now of course how to include the closing-paren-quote sequence in a string. For this, you can specify your own multi character delimiter:

**Code:**

```
    cout << R"limit("(string with }

{ \weird\ stuff)")limit" << '\n';
```

**Output**
**[string] raw2:**

```
"(string with }

{ \weird\ stuff)"
```

### 11.4.2 String literal suffix

A string literal `"foo"` is often compatible with a `std::string` but it is not of that type. Should you need that, you can add a suffix to the literal, which is defined in the namespace `string_literals`:

**Code:**

```
void printfun( string s ) {
  cout << s << '\n';
}
void printfun_c( const string& s ) {
  cout << s << '\n';
}
  /* ... */
    using namespace
    std::string_literals;
    printfun( "abc" );
    printfun( "def"s );
    printfun_c( "ghi" );
    printfun_c( "jkl"s );
```

**Output**
**[string] stringsuffix:**

```
abc
def
ghi
jkl
```

### 11.4.3 Conversion to/from string

#### *11.4.3.1 Converting to string*

There are various mechanisms for converting between strings and numbers.

- The C legacy mechanisms `sprintf` and `itoa`.
- `to_string`
- Above you saw the `stringstream`; section 11.3. Another use of this header follows below.
- The *Boost* library has a *lexical cast*.

Additionally, in C++17 there is the *charconv* header with `to_chars` and `from_chars`. These are low level routines that do not throw exceptions, do not allocate, and are each other's inverses. The low level nature is for instance apparent in the fact that they work on character buffers (not null-terminated). Thus, they can be used to build more sophisticated tools on top of.

#### *11.4.3.2 Converting from string*

The `stringstream` object can be used to convert strings to numbers:

1. Initialize the string stream with a string;
2. Use a `cin`-like syntax to set a numerical variable from the stream.

```
string stringnum="12345";
int num;
stringstream numstream(stringnum);
numstream >> num;
```

### 11.4.4 Unicode

C++ strings are essentially vectors of characters. A character is a single byte. Unfortunately, in these interweb days there are more characters around than fit in one byte. In particular, there is the *Unicode* standard that covers millions of characters. The way they are rendered is by an *extendible encoding*, in particular *UTF8*. This means that sometimes a single 'character', or more correctly *glyph*, takes more than one byte.

## 11.5 C strings

In C a string is essentially an array of characters. C arrays don't store their length, but strings do have functions that implicitly or explicitly rely on this knowledge, so they have a terminator character: ASCII `NULL`. C strings are called *null-terminated* for this reason.

# Chapter 12

# Input/output

Most programs take some form of input, and produce some form of output. In a beginning course such as this, the output is of more importance than the input, and what output there is only goes to the screen, so that's what we start by focusing on. We will also look at output to file, and input.

## 12.1 Screen output

In examples so far, you have used `cout` with its default formatting. In this section we look at ways of customizing the `cout` output.

**Remark 9** *Even after the material below you may find* `cout` *not particularly elegant. In fact, if you've programmed in C before, you may prefer the* `printf` *mechanism. The C++20 standard has the* `format` *header, which is as powerful as* `printf`*, but considerably more elegant.*

*However, as of early 2022 this is not available in most compilers, so in section 12.6 we will give examples from* fmtlib*, the open source library that gave rise to* `std::format`*.*

From `iostream`: `cout` uses default formatting.

Possible manipulation in `iomanip` header: pad a number, use limited precision, format as hex, etc.

Normally, output of numbers takes up precisely the space that it needs:

**Code:**

```
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << i << '\n';
cout << '\n';
```

**Output**
**[io] cunformat:**

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

We will now look at some examples of non-standard formatting. You may for instance want to output several lines, with the numbers in them nicely aligned. The most common I/O manipulation is to set a uniform width, that is, use the same number of positions for each number, regardless how many they need.

- The `setw` specifies how many positions to use for the following number.
- Note the singular in the previous sentence: the `setw` specifier applies only once.
- By default, numbers are right-aligned in the space given for them, and if they require more positions, they overflow on the right.

You can specify the number of positions, and the output is right aligned in that space by default:

**Code:**

```
#include <iomanip>
using std::setw;
  /* ... */
  cout << "Width is 6:" << '\n';
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setw(6) << i << '\n';
  cout << '\n';

  // 'setw' applies only once:
  cout << "Width is 6:" << '\n';
  cout << ">"
       << setw(6) << 1 << 2 << 3 << '\n';
  cout << '\n';
```

**Output**
**[io] width:**

```
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

Width is 6:
>     123
```

Normally, padding is done with spaces, but you can specify other characters:

**Code:**

```
#include <iomanip>
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.')
         << setw(6) << i
         << '\n';
```

**Output**
**[io] formatpad:**

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Note: single quotes denote characters, double quotes denote strings.

Instead of right alignment you can do left:

**Code:**

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.')
         << setw(6) << i << '\n';
```

**Output**
**[io] formatleft:**

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Finally, you can print in different number bases than 10:

**Code:**

```
#include <iomanip>
using std::setbase;
using std::setfill;
  /* ... */
  cout << setbase(16)
       << setfill(' ');
  for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
      cout << i*16+j << " " ;
    cout << '\n';
  }
```

**Output**
**[io] format16:**

```
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

**Exercise 12.1.** Make the first line in the above output align better with the other lines:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

Hex output is useful for addresses (chapter 17.2):

**Code:**

```
int i;
cout << "address of i, decimal: "
     << (long)&i << '\n';
cout << "address of i, hex    : "
     << std::hex << &i << '\n';
```

**Output**
**[pointer] coutpoint:**

```
address of i, decimal:
    140732703427524
address of i, hex    :
    0x7ffee2cbcbc4
```

Back to decimal:

```
cout << hex << i << dec << j;
```

There is no standard modifier for outputting as binary. However, you can use the `bitset` header to print the bit pattern of an integer.

**Code:**

```
#include <bitset>
using std::bitset;
  /* ... */
  auto x255 = bitset<16>(255);
  cout << x255 << '\n';
```

**Output**
**[io] bits:**

```
0000000011111111
```

### 12.1.1 Floating point output

The output of floating point numbers is more tricky.

- How many positions are used for the digits before the decimal point?
- How many digits after the decimal point are printed?
- Is scientific notation used?

For floating point numbers, the **setprecision** modifier determines how many positions are used for the integral and fractional part together. If the integral part takes more positions, scientific notation is used.

Use `setprecision` to set the number of digits before and after decimal point:

**Code:**

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
  /* ... */
  x = 1.234567;
  for (int i=0; i<10; i++) {
    cout << setprecision(4) << x << '\n';
    x *= 10;
  }
```

**Output**
**[io] formatfloat:**

```
1.235
12.35
123.5
1235
1.235e+04
1.235e+05
1.235e+06
1.235e+07
1.235e+08
1.235e+09
```

This mode is a mix of fixed and floating point. See the **scientific** option below for consistent use of floating point format.

With the `fixed` modifier, `setprecision` applies to the fractional part.

Fixed precision applies to fractional part:

**Code:**

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setprecision(4) << x << '\n';
  x *= 10;
}
```

**Output**
**[io] fix:**

```
1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

(Notice the rounding)

The `setw` modifier, for fixed point output, applies to the total width of integral and fractional part, plus the decimal point.

Combine width and precision:

**Code:**

```
  x = 1.234567;
  cout << fixed;
  for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4)
    << x
         << '\n';
    x *= 10;
  }
```

**Output**
**[io] align:**

```
    1.2346
   12.3457
  123.4567
 1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

**Exercise 12.2.** Use integer output to print real numbers aligned on the decimal:

**Code:**

```
string quasifix(double);
int main() {
  for ( auto x : { 1.5, 12.32, 123.456,
    1234.5678 } )
    cout << quasifix(x) << '\n';
```

**Output**
**[io] quasifix:**

```
   1.5
  12.32
 123.456
1234.5678
```

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

Above you saw that `setprecision` may give both fixed and floating point output. To get strictly floating point 'scientific' notation output, use `scientific`.

Combining width and precision:

**Code:**

```
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
  cout << setw(10) << setprecision(4)
       << x << '\n';
  x *= 10;
}
cout << '\n';
```

**Output**
**[io] iofsci:**

```
1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

### 12.1.2 Saving and restoring settings

```
ios::fmtflags old_settings = cout.flags();

cout.flags(old_settings);

int old_precision = cout.precision();

cout.precision(old_precision);
```

## 12.2 File output

The `iostream` is just one example of a *stream*, which is a general mechanism for converting entities to exportable form.

In particular, file output works the same as screen output: after you create a stream variable, you can 'lessless' to it.

```
mystream << "x: " << x << '\n';
```

The following example uses an `ofstream`: an output file stream. This has an `open` method to associate it with a file, and a corresponding `close` method.

Use:

**Code:**

```
#include <fstream>
using std::ofstream;
  /* ... */
  ofstream file_out;
  file_out.open
    ("fio_example.out");
  /* ... */
  file_out << number << '\n';
  file_out.close();
```

**Output**
**[io] fio:**

```
echo 24 | ./fio ; \
          cat fio_example.out
A number please:
Written.
24
```

Compare: `cout` is a stream that has already been opened to your terminal 'file'.

The *open* call can have flags, for instance for appending:

```
file.open(name,std::fstream::out | std::fstream::app);
g
```

Binary output: write your data byte-by-byte from memory to file.
(Why is that better than a printable representation?)

**Code:**

```
ofstream file_out;
file_out.open
  ("fio_binary.out",ios::binary);
/* ... */
file_out.write( (char*)(&number),4);
```

**Output**
**[io] fiobin:**

```
echo 25 | ./fiobin ; \
          od fio_binary.out
A number please: Written.
0000000    000031  000000
0000004
```

## 12.3 Output your own classes

You have used statements like:

```
cout << "My value is: " << myvalue << "\n";
```

How does this work? The 'double less' is an operator with a left operand that is a stream, and a right operand for which output is defined; the result of this operator is again a stream. Recursively, this means you can chain any number of applications of << together.

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

Here we solve this in two steps:

Define a function that yields a string representing the object, and

```
  string as_string() {
    stringstream ss;
    ss << "(" << x << "," << y << ")";
    return ss.str();
  };
  /* ... */
std::ostream& operator<<
    (std::ostream &out,Point &p) {
  out << p.as_string(); return out;
};
```

Redefine the less-less operator to use this.

```
Point p1(1.,2.);
cout << "p1 " << p1
     << " has length "
     << p1.length() << '\n';
```

(See section 11.3 for `stringstream` and the `sstream` header.)

If you don't want to write that accessor function, you can declare the lessless operator as a **`friend`**:

```
class container {
private: double x;
public:
  friend ostream& operator<<( ostream& s,const container& c ) {
    s << c.x; /* no accessor */
    return s; };
};
```

## 12.4 Output buffering

In C, the way to get a newline in your output was to include the character `\n` in the output. This still works in C++, and at first it seems there is no difference with using **`endl`**. However, **`endl`** does more than breaking the output line: it performs a **`std`**`::flush`.

### 12.4.1 The need for flushing

Output is usually not immediately written to screen or disc or printer: it is saved up in buffers. This can be for efficiency, because output a single character may have a large overhead, or it may be because the device is busy doing something else, and you don't want your program to hang waiting for the device to free up.

However, a problem with buffering is the output on the screen may lag behind the actual state of the program. In particular, if your program crashes before it prints a certain message, does it mean that it crashed before it got to that line, or does it mean that the message is hanging in a buffer.

This sort of output, that absolutely needs to be handled when the statement is called, is often called *logging* output. The fact that **`endl`** does a flush would mean that it would be good for logging output. However, it also flushes when not strictly necessary. In fact there is a better solution: **`std`**`::cerr` works just like **`cout`**, except it doesn't buffer the output.

### 12.4.2 Performance considerations

If you want a newline in your output (whether screen or more general stream), using `endl` may slow down your program because of the flush it performs. More efficiently, you would add a newline character to the output directly:

```
somestream << "Value: " << x << '\n';
otherstream << "Total " << nerrors << " reported\n";
```

In other words, use `cout` for regular output, `cerr` for logging output, and use `\n` instead of `endl`.

## 12.5 Input

The `cin` command can be used to read integers and floating point formats.

**Code:**

```
float input;
cin >> input;
cout << "(I think I got: " << input <<
    ")\n";
```

**Output**
**[io] cinfloat:**

```
for n in \
            1.5  1.6  1.67
    1.67e5  2.5.6  \
            ; do \
                echo $n |
    ./cinfloat \
        ; done
(I think I got: 1.5)
(I think I got: 1.6)
(I think I got: 1.67)
(I think I got: 167000)
(I think I got: 2.5)
```

As is illustrated with the last number in this example, `cin` will read until the first character that does not fit the format of the variable, in this case the second period. On the other hand, the `e` in the number before it is interpreted as the exponent of a floating point representation.

It is better to use *getline*. This returns a string, rather than a value, so you need to convert it with the following bit of magic:

```
#include <iostream>
using std::cin;
using std::cout;
#include <sstream>
using std::stringstream;
  /* ... */
  std::string saymany;
  int howmany;

  cout << "How many times? ";
  getline( cin,saymany );
  stringstream saidmany(saymany);
  saidmany >> howmany;
```

You can not use `cin` and *getline* in the same program.

More info: http://www.cplusplus.com/forum/articles/6046/.

### 12.5.1 File input

Input file stream, method *open*, then use *getline* to read one line at a time:

```
#include <fstream>
using std::ifstream;
  /* ... */
  ifstream input_file;
  input_file.open("fox.txt");
  string oneline;
  while (getline(input_file,oneline)) {
    cout << "Got line: <<" << oneline << ">>" << '\n';
  }
```

There are several ways of testing for the end of a file

- For text files, the *getline* function returns **false** if no line can be read.
- The *eof* function can be used after you have done a read.
- *EOF* is a return code of some library functions; it is not true that a file ends with an EOT character. Likewise you can not assume a Control-D or Control-Z at the end of the file.

**Exercise 12.3.** Put the following text in a file:

```
the quick brown fox
jummps over the
lazy dog.
```

Open the file, read it in, and count how often each letter in the alphabet occurs in it

*Advanced note:* You may think that *getline* always returns a **bool**, but that's not true. If actually returns an *ifstream*. However, a conversion operator

```
explicit operator bool() const;
```

exists for anything that inherits from *basic_ios*.

### 12.5.2 Input streams

Tests, mostly for file streams: *is_eof* *is_open*

### 12.5.3 C-style file handling

The old *FILE* type should not be used anymore.

## 12.6 Fmtlib

### 12.6.1 basics

- *print* for printing,
  *format* gives **std**::*string*;
- Arguments indicated by curly braces;
- braces can contain numbers (and modifiers, see next)

**Code:**

```
auto hello_string = fmt::format
  ("{} {}!","Hello","world");
cout << hello_string << '\n';
fmt::print
  ("{0}, {0} {1}!\n","Hello","world");
```

**Output**
**[io] fmtbasic:**

```
Hello world!
Hello, Hello world!
```

API documentation: https://fmt.dev/latest/api.html

### 12.6.2 Align and padding

In `fmtlib`, the 'greater than' sign plus a number indicates right aligning and the width of the field.

**Code:**

```
for (int i=10; i<2000000000; i*=10)
  fmt::print("{:>6}\n",i);
```

**Output**
**[io] fmtwidth:**

```
    10
   100
  1000
 10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

**Code:**

```
for (int i=10; i<2000000000; i*=10)
  fmt::print("{0:.>6}\n",i);
```

**Output**
**[io] fmtleftpad:**

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

### 12.6.3 Construct a string

If you want to construct a string piecemeal, for instance because it involves a loop over something, you can use a *memory_buffer*:

```
fmt::memory_buffer b;
fmt::format_to(std::back_inserter(b),"[");
for ( auto i : indices )
  fmt::format_to(std::back_inserter(b),"{}, ",i);
fmt::format_to(std::back_inserter(b),"]");
cout << to_string(b) << endl;
```

### 12.6.4 Number bases

In `fmtlib`, you can indicate the base with which to represent an integer by specifying one of `box` for binary, octal, hex respectively.

**Code:**

```
fmt::print
  ("{0} = {0:b} bin,\n    {0:o} oct,\n
    {0:x} hex\n",
   17);
```

**Output**

**[io] fmtbase:**

```
17 = 10001 bin,
    21 oct,
    11 hex
```

### 12.6.5 Output your own classes

With *fmtlib* this takes a different approach: here you need to specialize the *formatter* struct/class.

**Code:**

```
template <> struct
    fmt::formatter<point> {
 constexpr
 auto parse(format_parse_context& ctx)
       -> decltype(ctx.begin()) {
   auto it = ctx.begin(),
     end = ctx.end();
   if (it != end && *it != '}')
     throw format_error("invalid
    format");
   return it;
  }
  template <typename FormatContext>
  auto format
      (const point& p, FormatContext&
    ctx)
       -> decltype(ctx.out()) {
    return format_to
      (ctx.out(),
       "{}", p.as_string());
  }
};
  /* ... */
    point p(1.1,2.2);
    fmt::print("{}\n",p);
```

**Output**
**[io] fmtstream:**

```
(1.1,2.2)
```

# Chapter 13

# Lambda expressions

The mechanism of *lambda expressions* (first added in C++11 and since expanded) makes dynamic definition of functions possible.

Traditional function usage:
explicitly define a function and apply it:

```
double sum(float x,float y) { return x+y; }
cout << sum( 1.2, 3.4 );
```

New:
apply the function recipe directly:

**Code:**

```
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

**Output**
**[func] lambdadirect:**

```
3.8
```

This example is of course fairly pointless, but it illustrates the syntax of a lambda expression:

```
[capture] ( inputs ) -> outtype { definition };
[capture] ( inputs ) { definition };
```

- The square brackets, in this case, but not in general, empty, are the *capture* part;
- then follows the usual argument list;
- with a stylized arrow you can indicate the return type, but this is optional if the compiler can figure it out by itself;
- and finally the usual function body, include `return` statement for non-void functions.

**Remark 10** *Lambda expressions are sometimes called* closures, *but this term has a technical meaning in programming language theory, that only partly coincides with C++ lambda expressions.*

There are uses for 'Immmediately Invoked Lambda Expression'.

Example: different constructors.

Does not work:

```
if (foo)
  MyClass x(5,5);
else
  MyClass x(6);
```

Solution:

```
MyClass x =
  [foo] () {
    if (foo)
      return MyClass(5,5);
    else
      return MyClass(6);
  }();
```

OpenMP:

```
const int nthreads = [] () -> int {
  int nt;
#pragma omp parallel
#pragma omp master
  nt = omp_get_num_threads();
  return nt;
}();
```

For a slightly more useful example, we can assign the lambda expression to a variable, and repeatedly apply it.

**Code:**

```
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << '\n';
cout << summing ( 3.7, 5.2 ) << '\n';
```

**Output**
**[func] lambdavar:**

```
3.8
8.9
```

- This is a variable declaration.
- Uses `auto` for technical reasons; see later.

Return type could have been omitted:

```
auto summing =
[] (float x,float y) { return x+y; };
```

**Exercise 13.1.** Do exercise 48.9 of the zero finding project.

## 13.1 Lambda expressions as function argument

Above, when we assigned a lambda expression to a variable, we used `auto` for the type. The reason for this is that each lambda expression gets its own unique type, that is dynamically generated. But that makes it hard to pass that variable to a function.

Suppose we want to pass a lambda expression to a function:

```
int main() {
    apply_to_5( [] (int i) { cout << i+1; } );
```

What type do we use for the function parameter?

```
void apply_to_5( /* what type are we giving? */ f ) {
    f(5);
}
```

Since the type of the lambda expression is dynamically generated, we can not specify that type in the function header.

The way out is to use the *functional* header:

```
#include <functional>
using std::function;
```

With this, you can declare parameters by their signature.

In the following example we write a function *apply_to_5* which

- takes a function *f*, and
- applies it to 5.

We call the *apply_to_5* function with a lambda expression as argument:

**Code:**

```
1 void apply_to_5
2     ( function< void(int) > f ) {
3   f(5);
4 }
5   /* ... */
6   apply_to_5
7     ( [] (int i) {
8       cout << "Int: " << i << '\n'; } );
```

**Output**
**[func] lambdapass:**

*Int*: 5

**Exercise 13.2.** Do exercise 48.10 of the zero-finding project.

### 13.1.1 Lambda members of classes

The fact that a lambda expression has a dynamically generated type also makes it hard to store it in an object. To do this we again use **std**::*function*.

In the following example we make a class *SelectedInts* which takes a boolean function in the constructor: an object will contain only those integers that satisfy the function.

A set of integers, with a test on which ones can be admitted:

```
#include <functional>
using std::function;
  /* ... */
class SelectedInts {
private:
  vector<int> bag;
  function< bool(int) > selector;
public:
  SelectedInts
      ( function< bool(int) > f ) {
    selector = f; };
  void add(int i) {
    if (selector(i))
      bag.push_back(i);
  };
  int size() {
    return bag.size(); };
  std::string string() {
    std::string s;
    for ( int i : bag )
      s += to_string(i)+" ";
    return s;
  };
};
```

We use the above class to construct an object as follows:

- we read an integer `divisor`,
- and accept only those integers into our object that are divisible by that number.

For this we write a lambda expression `is_divisible` that

- captures the divisor, and then
- takes an integer as (its only) argument,
- returning whether that argument is divisible.

**Code:**

```
cout << "Give a divisor: ";
cin >> divisor; cout << '\n';
cout << ".. using " << divisor
     << '\n';
auto is_divisible =
  [divisor] (int i) -> bool {
    return i%divisor==0; };
SelectedInts multiples( is_divisible );
for (int i=1; i<50; i++)
  multiples.add(i);
```

**Output**

```
[func] lambdafun:

Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```

## 13.2 Captures

A *capture* is a way to 'bake variables into' a function. Let's say we want a function that increments its input, and the increment amount is set when we define the function.

Increment function:

- scalar in, scalar out;
- the increment amount has been fixed through the capture.

**Code:**

```
int one=1;
auto increment_by_1 =
  [one] ( int input ) -> int {
    return input+one;
};
cout << increment_by_1 (5)  << '\n';
cout << increment_by_1 (12)  << '\n';
cout << increment_by_1 (25)  << '\n';
```

**Output**
**[func] lambdavalue:**

```
6
13
26
```

**Exercise 13.3.** Write a program that

- reads a `float` factor;
- defines a function `multiply` that multiplies its input by that factor.

You can capture more than one variable. Explicitly capturing variables is done with a comma-separated list.

Example: multiply by a fraction.

```
int d=2,n=3;
times_fraction = [d,n] (int i) ->int {
    return (i*d)/n;
}
```

**Exercise 13.4.**

- Set two variables

```
float low = .5, high = 1.5;
```

- Define a function of one variable that tests whether that variable is between `low`, `high`. (Hint: what is the signature of that function? What is/are input parameter(s) and what is the return result?)

**Exercise 13.5.** Do exercises 48.11 and 48.12 of the zero-finding project.

### 13.2.1 Capture by reference

Normally, captured variables are copied by value.

Attempting to change the captured variable doesn't even compile:

```
auto f = // WRONG DOES NOT COMPILE
    [x] ( float &y ) -> void {
        x *= 2; y += x; };
```

If you do want to alter the captured parameter, pass it by reference:

**Code:**

```
int stride = 1;
auto more_and_more =
  [&stride] ( int input ) -> void {
    cout << input << "=>" <<
    input+stride << '\n';
    stride++;
};
more_and_more(5);
more_and_more(6);
more_and_more(7);
more_and_more(8);
more_and_more(9);
cout << "stride is now: " << stride <<
    '\n';
```

**Output**
**[func] lambdareference:**

```
5=>6
6=>8
7=>10
8=>12
9=>14
stride is now: 6
```

Capturing by reference can for instance be useful if you are performing some sort of reduction. The capture is then the reduction variable, and the numbers to be reduced come in as function parameter to the lambda expression.

In this example we count how many of the input values test true under a certain function `f`:

Capture a variable by reference so that you can update it:

```
int count=0;
auto count_if_f =
    [&count] (int i) {
      if (f(i)) count++; }
for ( int i : int_data )
  count_if_f(i);
cout << "We counted: " << count;
```

(See the `algorithm` header section 14.2.)

### 13.2.2 Capturing 'this'

In addition to capturing specific variable, whether by reference or not, as you saw above, you can also capture the whole environment of a lambda. For this the following shorthands exist:

```
[=] () {} // capture everything by value
[&] () {} // capture everything by reference
```

In the context of a class method, this means you are capturing **`this`**. As of C++20, implicit capture of **`this`** by value is deprecated.

## 13.3 More

### 13.3.1 Making lambda stateful

Let's consider the issue of lambda expressions and mutable state, by which we mean no more than that a variable gets updated multiple times.

A simple example is a doing a count reduction: how many items satisfy some test. In the following lambda expression, the item is passed as an argument, while the count is captured by reference.

**Code:**

```
vector<int> moreints{8,9,10,11,12};
int count{0};
for_each
  ( moreints.begin(),moreints.end(),
    [&count] (int x) {
      if (x%2==0)
        count++;
    } );
cout << "number of even: " << count
  << '\n';
```

**Output**
**[stl] counteach:**

```
number of even: 3
```

How about if that count is not really needed in the calling environment of the lambda expression; can we somehow make it internal?

Lambda expressions are normally stateless:

**Code:**

```
float x = 2, y = 3;
auto f = [x] ( float &y ) -> void {
  int xx = x*2; y += xx; };
f(y);
cout << y << '\n';
f(y);
cout << y << '\n';
```

**Output**
**[func] nonmutable:**

```
7
11
```

but with the **mutable** keyword you can make them stateful:

**Code:**

```
float x = 2, y = 3;
aueto f = [x] ( float &y ) mutable ->
    void {
  x *= 2; y += x; };
f(y);
cout << y << '\n';
f(y);
cout << y << '\n';
```

**Output**
**[func] yesmutable:**

```
7
15
```

Here is a nifty application: printing a list of numbers, separated by commas, but without trailing comma:

**Code:**

```
vector x{1,2,3,4,5};
auto printdigit =
  [start=true] (auto xx) mutable ->
  string{
    if (start) {
      start = false;
      return to_string(xx);
    } else
      return ","+to_string(xx);
  };
for ( auto xx : x )
  cout << printdigit(xx);
cout << '\n';
```

**Output**
**[func] lambdaexch:**

```
1,2,3,4,5
```

### 13.3.2 Generic lambdas

The `auto` keyword can be used for *generic lambdas*:

```
auto compare = [] (auto a,auto b) { return a<b; };
```

Here the return type is clear, but the input types are generic. This is much like using a templated function: the compiler instantiates the expression with whatever types are needed.

### 13.3.3 Algorithms

The `algorithm` header (section 14.2) contains a number of functions that naturally use lambdas. For instance, `any_of` can test whether any element of a `vector` satisfies a condition. You can then use a lambda to specify the `bool` function that tests the condition.

### 13.3.4 C-style function pointers

The C language had a – somewhat confusing – notation for function pointers. If you need to interface with code that uses them, it is possible to use lambda functions to an extent: lambdas without captures can be converted to a function pointer.

**Code:**

```
int cfun_add1( int i ) {
  return i+1; };
int apply_to_5( int(*f)(int) ) {
  return f(5); };
//codesnippet  end
  /* ... */
  auto lambda_add1 =
    [] (int i) { return i+1; };
  cout << "C ptr: "
       << apply_to_5(&cfun_add1)
       << '\n';
  cout << "Lambda: "
       << apply_to_5(lambda_add1)
       << '\n';
```

**Output**
**[func] lambdacptr:**

```
C ptr: 6
Lambda: 6
```

# Chapter 14

# Iterators, Algorithms, Ranges

## 14.1 Iterators

Iterating through objects such as vectors isn't simply a process of keeping a counter that says where you are, and taking that element if needed. Many C++ classes have an *iterator* subclass, that gives a formal description of 'where you are' and what you can find there. Having iterators means that you can traverse structure that don't have an explicit index count, but there are many other conveniences as well.

By way of examples, here is some iterator manipulation.

- Iteratable containers have a `begin` and `end` iterator.
- The end iterator 'points' just beyond the last element.
- The '`*`' star operator gives the element that the iterator points to.
- You can increment and decrement them.

**Code:**

```
vector<int> counts{1,2,3,4};
auto second = counts.begin(); second++;
cout << "Second element: " << *second <<
    '\n';
auto last = counts.end(); last--;
cout << "Last element: " << *last << '\n';
```

**Output**

```
[iter] plusminus:

Second element: 2
Last element: 4
```

### 14.1.1 Iterators

You have seen how you can iterate over a vector

- by an indexed loop over the indices, and
- with a range-based loop over the indices.

There is a third way, which is actually the basic mechanism underlying the range-based looping.

An *iterator* is, in a metaphorical sense a pointer to a vector element. Mirroring the index-loop convention of

```
for (int i=0; i<hi; i++)
  element = vec.at(i);
```

you can iterate:

```
for (auto elt_ptr=vec.begin(); elt_ptr!=vec.end(); ++elt_ptr)
  element = *elt_ptr;
```

Some remarks:

- This is one of the very few places where you need the asterisk in C++. However, you're applying it to an iterator, not a pointer, and this is an operator you are applying.
- As with a normal loop, the `end` iterator point just beyond the end of the vector.
- You can do *pointer arithmetic* on iterators, as you can see in the `++elt_ptr` update part of the loop header.

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
  s += copy_of_int;
for ( auto &ref_to_int : myvector )
  ref_to_int = s;
for ( const auto& copy_of_thing :
    myvector )
  s += copy_of_thing.f();
```

is actually short for:

```
for
 ( std::vector<int>::iterator
     it=myvector.begin() ;
   it!=myvector.end() ; ++it )
   s += *it ; // note the deref
```

Range iterators can be used with anything that is iteratable
(vector, map, your own classes!)

There are still some things that you can do with iterators that can not be done with range-based iteration. Iterating backward through a container is one:

Reverse iteration can not be done with range-based syntax.

Use general syntax with reverse iterator: `rbegin`, `rend`.

Another illustration of pointer arithmetic on iterators is getting the last element of a vector:

**Code:**

```
1 vector<int> mydata(5,2);
2 mydata.push_back(35);
3 cout << mydata.size() << '\n';
4 cout << mydata.back();
5      << '\n';
```

**Output**
**[array] vectorend:**

```
6
35
```

**Code:**

```
1 vector<int> mydata(5,2);
2 mydata.push_back(35);
3 cout << mydata.size() << '\n';
4 cout << *( --mydata.end() ) << '\n';
```

**Output**
**[array] vectorenditerator:**

```
6
35
```

### 14.1.2 How iterators are like pointers

The container class has a subclass *iterator* that can be used to iterate through all elements of a container. This was discussed in section 14.1.1.

However, an *iterator* can be used outside of strictly iterating. You can consider an iterator as a sort of 'pointer into a container', and you can move it about.

Let's look at some examples of using the *begin* and *end* iterators. In the following example:

- We first assign the *begin* and *end* iterators to variables; the begin iterator points at the first element, but the end iterator points just beyond the last element;
- Given an iterator, you get the value of the corresponding element by applying the 'star' operator to it;
- A sort of 'pointer arithmetic' can be applied to iterators.

Use independent of looping:

**Code:**

```
vector<int> v{1,3,5,7};
auto pointer = v.begin();
cout << "we start at "
     << *pointer << '\n';
pointer++;
cout << "after increment: "
     << *pointer << '\n';

pointer = v.end();
cout << "end is not a valid element: "
     << *pointer << '\n';
pointer--;
cout << "last element: "
     << *pointer << '\n';
```

**Output**

```
[stl] iter:
we start at 1
after increment: 3
end is not a valid element: 0
last element: 7
```

Note that the star notation is a *unary star operator*, not a *pointer dereference*:

```
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); second++;
cout << "Dereference second: "
     << *second << '\n';
// DOES NOT COMPILE
// the iterator is not a type-star:
// int *subarray = second;
```

### 14.1.3 Forming sub-arrays

Iterators can be used to construct a *vector*. This can for instance be used to create a *subvector*. In the simplest case, you would make a copy of a vector using begin/end iterators:

```
vector<int> sub( othervec.begin(),othervec.end() );
```

Note that the subvector is formed as a copy of the original elements. Vectors completely 'own' their elements. For non-owning subvectors you would need **span**; section 10.9.5.

Some more examples:

**Code:**

```
  vector<int> vec{11,22,33,44,55,66};
  auto second = vec.begin(); second++;
  auto before = vec.end(); before--;
  //    vector<int> sub(second,before);
  vector<int>
    sub(vec.data()+1,vec.data()+vec.size()-1);
  cout << "no first and last: ";
  for ( auto i : sub ) cout << i << ",
    ";
  cout << '\n';
  vec.at(1) = 222;
  cout << "did we get a change in the
    sub vector? " << sub.at(0) << '\n';
/* ... */
  vector<int> vec{11,22,33,44,55,66};
  auto second = vec.begin(); second++;
  auto before = vec.end(); before--;
  //    vector<int> sub(second,before);
  vector<int> sub;
    sub.assign(second,before);
  cout << "vector at " <<
    (long)vec.data() << '\n';
  cout << "sub at " << (long)sub.data()
    << '\n';

  cout << "no first and last: ";
  for ( auto i : sub ) cout << i << ",
    ";
  cout << '\n';
  vec.at(1) = 222;
  cout << "did we get a change in the
    sub vector? " << sub.at(0) << '\n';
```

**Output**
**[iter] subvector:**

```
no first and last: 22, 33,
    44, 55,
did we get a change in the
    sub vector? 22
vector at 140444369443744
sub at 140444369443776
no first and last: 22, 33,
    44, 55,
did we get a change in the
    sub vector? 22
```

### 14.1.4 Vector operations through iterators

You have already seen that the length of a vector can be extended by the `push_back` method (section 10.3.2).

With iterators other operations are possible, such as copying, erasing, and inserting.

First we show the use of `copy` which takes two iterators in one container to define the range to be copied, and one iterator in the target container, which can be the same as the source. The copy operation will overwrite elements in the target, but without bound checking, so make sure there is enough space.

Copy a begin/end range of one container
to an iterator in another container::

**Code:**

```
vector<int> counts{1,2,3,4};
vector<int> copied(5);
copy( counts.begin(),counts.end(),
      copied.begin()+1 );
cout << copied[0]
     << ", " << copied[1]
     << ".." << copied[4] << '\n';
```

**Output**
**[iter] copy:**

```
0, 1..4
```

(No bound checking, so be careful!)

The erase operation `erase` takes two iterators, defining the inclusive lower and exclusive upper bound for the range to erase.

Erase from start to before-end:

**Code:**

```
vector<int> counts{1,2,3,4,5,6};
vector<int>::iterator second =
    counts.begin()+1;
auto fourth = second+2;
counts.erase(second,fourth);
cout << counts[0]
     << "," << counts[1] << '\n';
```

**Output**
**[iter] erase2:**

```
1,4
```

(Also single element without end iterator.)

The `insert` operation takes a target iterator after which the insertion takes place, and two iterators for the range that will be inserted. This will extend the size of the target container.

Insert at iterator: value, single iterator, or range:

**Code:**

```
vector<int> counts{1,2,3,4,5,6},
  zeros{0,0};
auto after_one = zeros.begin()+1;
zeros.insert
  ( after_one,
    counts.begin()+1,
    counts.begin()+3 );
cout << zeros[0] << ","
     << zeros[1] << ","
     << zeros[2] << ","
     << zeros[3]
     << '\n';
```

**Output**
**[iter] insert2:**

```
0,2,3,0
```

#### 14.1.4.1 Indexing and iterating

Functions that would return an array element or location, now return iterators. For instance:

- `find` returns an iterator pointing to the first element equal to the value we are finding;

- *max_element* returns an iterator pointing to the element with maximum value.

One of the arguments for range-based indexing was that we get a simple syntax if we don't need the index. Is it possible to use iterators and still get the index? Yes, that's what the function *distance* is for.

Find 'index' by getting the distance between two iterators:

**Code:**

```
vector<int> numbers{1,3,5,7,9};
auto it=numbers.begin();
while ( it!=numbers.end() ) {
  auto d = distance(numbers.begin(),it);
  cout << "At distance " << d
       << ": " << *it << '\n';
  it++;
}
```

**Output**

```
[loop] distance:

At distance 0: 1
At distance 1: 3
At distance 2: 5
At distance 3: 7
At distance 4: 9
```

**Exercise 14.1.** Use the above vector methods to return, given a `std::vector<float>`, the integer index of its maximum element.

### 14.1.5 Iterating over classes

You know that you can iterate over `vector` objects:

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
  s += copy_of_int;
for ( auto &ref_to_int : myvector )
  ref_to_int = s;
```

(Many other STL classes are iterable like this.)

This is not magic: it is possible to iterate over any class: a *class* is *iteratable* that has a number of conditions satisfied.

The class needs to have:

- a method *begin* with prototype

  ```
  iteratableClass iteratableClass::begin()
  ```

  That gives an object in the initial state, which we will call the 'iterator object'; likewise
- a method *end*

  ```
  iteratableClass iteratableClass::end()}
  ```

  that gives an object in the final state; furthermore you need
- an increment operator

  ```
  void iteratableClass::operator++()
  ```

  that advances the iterator object to the next state;
- a test

  ```
  bool iteratableClass::operator!=(const iteratableClass&)
  ```

  to determine whether the iteration can continue; finally

- a dereference operator

```
iteratableClass::operator*()
```

that takes the iterator object and returns its state.

#### 14.1.5.1 Example 1

Let's make a class, called a `bag`, that models a set of integers, and we want to enumerate them. For simplicity sake we will make a set of contiguous integers:

```
class bag {
  // basic data
private:
  int first,last;
public:
  bag(int first,int last) : first(first),last(last) {};
```

When you create an iterator object it will be copy of the object you are iterating over, except that it remembers how far it has searched:

```
private:
  int seek{0};
```

The `begin` method gives a `bag` with the seek parameter initialized:

```
public:
  bag &begin()  {
    seek = first; return *this;
  };
  bag end() {
    seek = last; return *this;
  };
```

These routines are public because they are (implicitly) called by the client code.

The termination test method is called on the iterator, comparing it to the `end` object:

```
bool operator!=( const bag &test ) const {
  return seek<=test.last;
};
```

Finally, we need the increment method and the dereference. Both access the `seek` member:

```
void operator++() { seek++; };
int operator*() { return seek; };
```

We can iterate over our own class:

**Code:**

```
bag digits(0,9);

bool find3{false};
for ( auto seek : digits )
  find3 = find3 || (seek==3);
cout << "found 3: " << boolalpha
     << find3 << '\n';

bool find15{false};
for ( auto seek : digits )
  find15 = find15 || (seek==15);
cout << "found 15: " << boolalpha
     << find15 << '\n';
```

**Output**
**[loop] bagfind:**

```
found 3: true
found 15: false
```

(for this particular case, use `std::any_of`)

**Code:**

```
bool find8 = any_of
  ( digits.begin(),digits.end(),
    [=] (int i) { return i==8; } );
cout << "found 8: " << boolalpha
     << find8 << '\n';
```

**Output**
**[loop] bagany:**

```
found 8: true
```

If we add a method `has` to the class:

```
bool has(int tst) {
  for (auto seek : *this )
    if (seek==tst) return true;
  return false;
};
```

we can call this:

```
cout << "f3: " << digits.has(3) << '\n';
cout << "f15: " << digits.has(15) << '\n';
```

Of course, we could have written this function without the range-based iteration, but this implementation is particularly elegant.

**Exercise 14.2.** You can now do exercise 46.19, implementing a prime number generator with this mechanism.

If you think you understand `const`, consider that the `has` method is conceptually `cost`. But if you add that keyword, the compiler will complain about that use of `*this`, since it is altered through the `begin` method.

**Exercise 14.3.** Find a way to make `has` a `const` method.

#### *14.1.5.2 Example 2: iterator class*

Recall that

Short hand:

```
vector<float> v;
for ( auto e : v )
  ... e ...
```

for:

```
for ( vector<float>::iterator
e=v.begin();
  e!=v.end(); e++ )
  ... *e ...
```

If we want

```
for ( auto e : my_object )
... e ...
```

we need an iterator class with methods such as *begin*, *end*, * and ++.

Ranging over a class with iterator subclass

Class:

```
class NewVector {
protected:
  // vector data
  int *storage;
  int s;
  /* ... */
public:
  // iterator stuff
  class iter;
  iter begin();
  iter end();

};
```

Main:

```
NewVector v(s);
/* ... */
for ( auto e : v )
  cout << e << " ";
```

Random-access iterator:

```
NewVector::iter& operator++();
int& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-( const NewVector::iter& other ) const;
NewVector::iter& operator+=( int add );
```

**Exercise 14.4.** Write the missing iterator methods. Here's something to get you started.

```
class NewVector::iter {
private: int *searcher;
  /* ... */
NewVector::iter::iter( int *searcher )
```

```
  : searcher(searcher) {};
NewVector::iter NewVector::begin() {
  return NewVector::iter(storage); };
NewVector::iter NewVector::end()   {
  return NewVector::iter(storage+NewVector::s); };
```

## 14.2 Algorithms using iterators

Many simple algorithms on arrays, testing 'there is' or 'for all', no longer have to be coded out in C++. They can now be done with a single function from `std::algorithm`.

So, even if you have learned a to code a specific algorithm yourself in the foregoing, you should study the following algorithms, or at least known that such algorithms exist. It's what distinguishes a novice programmer from an industrial-grade (for want of a better term) programmer.

### 14.2.1 Test Any/all

First we look at some algorithms that apply a predicate to the elements.

- Test if any element satisfies a condition: `any_of`.
- Test if all elements satisfy a condition: `all_of`.
- Test if no elements satisfy a condition: `none_of`.
- Apply an operation to all elements: `for_each`.

The object to which the function applies is not specified directly; rather, you have to specify a start and end iterator.

(See section 14.1.2 for iterators, in particular section 14.2.1 for algorithms with iterators, and chapter 13 for the use of lambda expressions.)

As an example of applying a predicate we look at a couple of examples of using `any_of`. This returns true or false depending on whether the predicate is every true; this uses *short-circuit evaluation*.

Reduction with boolean result:
See if any element satisfies a test

**Code:**

```
1    vector<int>
     ints{2,3,4,5,7,8,13,14,15};
2    bool there_was_an_8 =
3      any_of( ints.begin(),ints.end(),
4              [] ( int i ) -> bool {
5                return i==8;
6              }
7              );
8    cout << "There was an 8: " <<
     boolalpha << there_was_an_8 << '\n';
```

**Output**

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

(Why wouldn't you use a `accumulate` reduction?)

Here is an example using *any_of* to find whether there is any even element in a vector:

**Code:**

```
vector<int> integers{1,2,3,5,7,10};
auto any_even = any_of
  ( integers.begin(),integers.end(),
    [=] (int i) -> bool {
      return i%2==0; }
  );
if (any_even)
  cout << "there was an even" << '\n';
else
  cout << "none were even" << '\n';
```

**Output**
**[range] anyof:**

```
there was an even
```

### 14.2.2 Apply to each

The *for_each* algorithm applies a function to every element of a container. Unlike the previous algorithms, this can alter the elements.

To introduce the syntax, we look at the pointless example of outputting each element:

**Code:**

```
#include <algorithm>
using std::for_each;
  /* ... */
  vector<int> ints{3,4,5,6,7};
  for_each
    ( ints.begin(),ints.end(),
      [] (int x) -> void {
        if (x%2==0)
          cout << x << '\n';
      } );
```

**Output**
**[stl] printeach:**

```
4
6
```

Apply something to each array element:

**Code:**

```
#include <algorithm>
  /* ... */
    vector<int>
    ints{2,3,4,5,7,8,13,14,15};
    for_each( ints.begin(),ints.end(),
              [] ( int i ) -> void {
                cout << i << '\n';
              }
              );
```

**Output**
**[iter] each:**

```
2
3
4
5
7
8
13
14
15
```

**Exercise 14.5.** Use *for_each* to sum the elements of a vector.

Hint: the problem is how to treat the sum variable. Do not use a global variable!

Capture by reference, to update with the array elements.

**Code:**

```
vector<int>
ints{2,3,4,5,7,8,13,14,15};
int sum=0;
for_each( ints.begin(),ints.end(),
          [&sum] ( int i ) -> void {
            sum += i;
          }
          );
cout << "Sum = " << sum << '\n';
```

**Output**
**[iter] each:**

```
2
3
4
5
7
8
13
14
15
```

### 14.2.3 Iterator result

Some algorithms do not result in a value, but rather in an iterator that points to the location of that value. Examples: `min_element` takes a begin and end iterator, and returns the iterator in between where the minimum element is found. To find the actual value, we need to 'dereference' the iterator:

```
vector<float> elements{.5f,1.f,1.5f};
auto min_iter = std::min_element
  (elements.begin(),elements.end());
cout << "Min: " << *min_iter << '\n';
```

Similarly `max_element`.

### 14.2.4 Mapping

The `transform` algorithm applies a function to each container element, modifying it in place:

```
std::transform( vec, vec.begin(), [] (int i) { return i*i; } );
```

### 14.2.5 Reduction

Numerical *reductions* can be applied using iterators in `accumulate` in the `numeric` header. If no reduction operator is specified, a *sum reduction* is performed.

Default is sum reduction:

**Code:**

```
#include <numeric>
using std::accumulate;
  /* ... */
    vector<int> v{1,3,5,7};
    auto first = v.begin();
    auto last  = v.end();
    auto sum = accumulate(first,last,0);
    cout << "sum: " << sum << '\n';
```

**Output**
**[stl] accumulate:**

```
sum: 16
```

Other binary *arithmetic operators*s that can be used as *reduction operator* are found in `functional`:

- *plus*, *minus*, *multiplies*, *divides*,
- integers only: *modulus*
- boolean: *logical_and*, *logical_or*

This header also contains the unary *negate* operator, which can of course not be used for reductions.

As an example of an explicitly specified reduction operator:

```
auto p = std::accumulate
  ( x.begin(),x_end(),1.f,
    std::multiplies<float>()
  );
```

Note:

- that the operator is templated, and that it is followed by parentheses to become a functor, rather than a class;
- that the accumulate function is templated, and it takes its type from the init value. Thus, in the above example, a value of `1` would have turned this into an integer operation.

Using lambda functions (chapter 13) we can get more complicated effects.

Supply multiply operator:

**Code:**

```
using std::multiplies;
/* ... */
  vector<int> v{1,3,5,7};
  auto first = v.begin();
  auto last  = v.end();
  first++; last--;
  auto product =
    accumulate(first,last,2,
               multiplies<>());
  cout << "product: " << product <<
  '\n';
```

**Output**

```
[stl] product:
product: 30
```

Specific for the max reduction is *max_element*. This can be called without a comparator (for numerical max), or with a comparator for general maximum operations. The maximum and minimum algorithms return an iterator, rather than only the max/min value.

Example: maximum relative deviation from a quantity:

```
max_element(myvalue.begin(),myvalue.end(),
    [my_sum_of_squares] (double x,double y) -> bool {
        return fabs( (my_sum_of_squares-x)/x ) < fabs( (my_sum_of_squares-y)/y
    ); }
);
```

For more complicated lambdas used in *accumulate*,

- the first argument should be the reduce type,
- the second argument should be the iterated type

In the following example we accumulate one member of a class:

```
class x {
public:
  int i,j;
  x() {};
  x(int i,int j) : i(i),j(j) {};
};
```

```
std::vector< x > xs(5);
auto xxx =
  std::accumulate
  ( xs.begin(),xs.end(),0,
    [] ( int init,x x1 ) -> int
{ return x1.i+init; }
    );
```

### 14.2.6 Sorting

The *algorithm* header also has a function *sort*.

With iterators you can easily apply this to things such as vectors:

```
sort( myvec.begin(),myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```
sort( myvec.begin(),myvec.end(),
      [] (int i,int) { return i>j; }
    );
```

or

```
sort( people.begin(),people.end(),
      [] ( const Person& lhs,const Person& rhs ) {
         return lhs.name < rhs.name; }
    )
```

With iterators you can also do things like sorting a part of the vector:

**Code:**

```
1 vector<int> v{3,1,2,4,5,7,9,11,12,8,10};
2 cout << "Original vector: " <<
      vector_as_string(v) << '\n';
3
4 auto v_std(v);
5 std::sort(
      v_std.begin(),v_std.begin()+5 );
6 cout << "Five elements sorts: " <<
      vector_as_string(v_std) << '\n';
```

**Output**

```
[range] sortit:
Original vector: 3, 1, 2, 4,
    5, 7, 9, 11, 12, 8, 10,
Five elements sorts: 1, 2,
    3, 4, 5, 7, 9, 11, 12, 8,
    10,
```

## 14.3 Ranges

The C++20 standard contains a *ranges* header, which generalizes iteratable objects into as-it-were streams, that can be connected with *pipes*s.

We need to introduced two new concepts.

A *range* is an iteratable object. The containers of the pre-17 STL are ranges, but some new ones have been added.

First of all, ranges provide a clearer syntax:

```
vector data{2,3,1};
sort( begin(data),end(data) ); // open to accidents
ranges::sort(data);
```

A *view* is somewhat similar to a range, in the sense that you can iterate over it. The difference is that, unlike for instance a *vector*, a view is not a completely formed object. A view doesn't own any data, and any elements you view in it get formed as you iterate over it. This is sometimes called *lazy evaluation* or *lazy execution*. Stated differently, its elements are constructed as they are being requested by the iteration over the view.

### 14.3.1 Standard algorithms

Many of the STL algorithms now have a range version. Compare two versions of *min_element*, both giving an iterator:

```
vector<float> elements{.5f,1.f,1.5f};
auto min_iter = std::min_element
  (elements.begin(),elements.end());
cout << "Min: " << *min_iter << '\n';
```

```
namespace rng = std::ranges;
vector<float> elements{.5f,1.f,1.5f};
auto min_iter = rng::min_element
  (elements);
cout << "Min: " << *min_iter << '\n';
```

### 14.3.2 Views

A *view* can informally be considered as 'a range that does not own its element'.

Views are composable: you can take one view, and pipe it into another one. If you need the resulting object, rather than the elements as a stream, you can call *to_vector*.

First two simple examples of views:

1. one formed by *transform*, which applies a function to each element of the range or view in sequence;
2. one formed by *filter*, which only yields those elements that satisfy some boolean test.

(We use an auxiliary function to turn a vector into a string.)

**Code:**

```
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original vector: "
     << vector_as_string(v) << '\n';
auto times_two = v
  | transform( [] (int i) {
      return 2*i; } );
cout << "Times two: "
 << vector_as_string
     ( times_two | ranges::to_vector )
     << '\n';
auto over_five = times_two
  | filter( [] (int i) {
      return i>5; } );
cout << "Over five: "
 << vector_as_string
     ( over_five | ranges::to_vector )
     << '\n';
```

**Output**
**[range] ft1:**

```
Original vector: 1, 2, 3, 4,
    5, 6,
Times two: 2, 4, 6, 8, 10,
    12,
Over five: 6, 8, 10, 12,
```

Next to illustrate the composition of streams:

**Code:**

```
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original vector: "
     << vector_as_string(v) << '\n';
auto times_two_over_five = v
  | transform( [] (int i) {
      return 2*i; } )
  | filter( [] (int i) {
      return i>5; } );
cout << "Times two over five: "
 << vector_as_string
     ( times_two_over_five |
    ranges::to_vector )
     << '\n';
```

**Output**
**[range] ft2:**

```
Original vector: 1, 2, 3, 4,
    5, 6,
Times two over five: 6, 8,
    10, 12,
```

**Exercise 14.6.** Make a vector that contains both positive and negative numbers. Use ranges to compute the minimum square root of the positive numbers.

Other available operations:

- dropping initial elements: `std::views::drop`
- reversing a vector: `std::views::drop`

### 14.3.3 Example: sum of squares

For computing the sum of squares we can use the range `transform` method for constructing a 'lazy container' of the squares. However, C++20 does not have a range version of the algorithms in `numeric` header, such as `accumulate`. This will be fixed in C++23.

```
vector<float> elements{.5f,1.f,1.5f};
namespace rng = std::ranges;
```

```
namespace vw = std::views;
auto square_view = vw::transform
  (elements, [] (auto e) { return e*e; } );
auto sumsq = std::accumulate
  (square_view.begin(),square_view.end(),0.f);
cout << "Sum of squares: " << sumsq << '\n';
```

### 14.3.4 Range types

Types of ranges:

- `std::ranges::input_range` : iterate forward at least once, as if you're accepting input with `cin` and such.
- `std::ranges::forward_range` : can be iterated forward, (for instance with plus-plus), multiple times, as in a *single-linked list*.
- `std::ranges::bidirectional_range` : can be iterated in both directions, for instance with plus-plus and minus-minus.
- `std::ranges::random_access_range` items can be found in constant time, such as with square bracket indexing.
- `std::ranges::contiguous_range` : items are stored consecutively in memory, making address calculations possible.

Two concepts relate to storage, independent of the range concept:

- containers are ranges such as `vector` and `deque`, which own their data;
- views are ranges that do not own their data, for instance because they come from transforming a container or another view.

Adaptors take a range and return a view. They can do that by themselves, or chained:

```
auto v = std::views::reverse(vec);
auto v = vec | std::views::reverse;
auto v = vec | std::views::reverse | /* more adaptors */ ;
```

### 14.3.5 Infinite sequences

Since views are lazily constructed, it is possible to have an infinite object – as long as you don't ask for its last element.

```
auto result { view::ints(10) // VLE ???
    | views::filter( [] ( const auto& value ) {
                        return value%2==0; } )
    | views::take(10) };
```

# Chapter 15

# References

## 15.1 Reference

This section contains further facts about references, which you have already seen as a mechanism for parameter passing; section 7.5.2. Make sure you study that material first.

Passing a variable to a routine passes the value; in the routine, the variable is local.

```
void change_scalar(int i) {
  i += 1;
}
  /* ... */
  number = 3;
  cout << "Number is 3: "
       << number << '\n';
  change_scalar(number);
  cout << "is it still 3? Let's see: "
       << number << '\n';
```

If you do want to make the change visible in the *calling environment*, use a reference:

```
void change_scalar_by_reference(int &i) { i += 1; }
```

There is no change to the calling program. (Some people who are used to C find this bad, since you can not see from the use of a function whether it passes *by reference* or *by value*.)

## 15.2 Pass by reference

If you use a mathematical style of subprograms, where some values go in, and a new entity comes out, in effect all the inputs can be copied. This style is called *functional programming*, and there is much to be said for it. For instance, it makes it possible for the compiler to reason about your program. The only thing you have to worry about is the cost of copying, if the inputs are of non-trivial size, such as arrays.

However, sometimes you want to alter the inputs, so instead of a copy you need a way of accessing the actual input object. That's what *references* are invented for: to allow a subprogram access to the actual input entity.

A bonus of using references is that you do not incur the cost of copying. So what if you want this efficiency, but your program is really functional in design? Then you can use a *const reference*: the

argument is passed by reference, but you indicate explicitly that the subprogram does not alter it, again allowing compiler optimizations.

A reference makes the function parameter a synonym of the argument.

```
void f( int &i ) { i += 1; };
int main() {
  int i = 2;
  f(i); // makes it 3
```

```
class BigDude {
public:
   vector<double> array(5000000);
}

void f(BigDude d) {
  cout << d.array[0];
};

int main() {
   BigDude big;
   f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

Prevent changes:

```
void f( const BigDude &thing ) {
    .... };
```

## 15.3 Reference to class members

Here is the naive way of returning a class member:

```
class Object {
private:
  SomeType thing;
public:
  SomeType get_thing() {
    return thing; };
};
```

The problem here is that the return statement makes a copy of `thing`, which can be expensive. Instead, it is better to return the member by *reference*:

```
  SomeType &get_thing() {
    return thing; };
```

The problem with this solution is that the calling program can now alter the private member. To prevent that, use a *const reference*:

**Code:**

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return
    mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc();
    an_int.inc();
  cout << "Contained int is now: "
       << an_int.int_to_get() << '\n';
  /* Compiler error:
    an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: "
       << an_int.int_to_get() << '\n';
```

**Output**
**[const] constref:**

```
Contained int is now: 4
Contained int is now: 17
```

In the above example, the function giving a reference was used in the left-hand side of an assignment. If you would use it on the right-hand side, you would not get a reference. The result of an expression can not be a reference.

Let's again make a class where we can get a reference to the internals:

```
class myclass {
private:
  int stored{0};
public:
  myclass(int i) : stored(i) {};
  int &data() { return stored; };
};
```

Now we explore various ways of using that reference on the right-hand side:

**Code:**

```
myclass obj(5);
cout << "object data: "
     << obj.data() << '\n';
int dcopy = obj.data();
dcopy++;
cout << "object data: "
     << obj.data() << '\n';
int &dref = obj.data();
dref++;
cout << "object data: "
     << obj.data() << '\n';
auto dauto = obj.data();
dauto++;
cout << "object data: "
     << obj.data() << '\n';
auto &aref = obj.data();
aref++;
cout << "object data: "
     << obj.data() << '\n';
```

**Output**
**[func] rhsref:**

```
object data: 5
object data: 5
object data: 6
object data: 6
object data: 7
```

(On the other hand, after `const auto &ref` the reference is not modifiable. This variant is useful if you want read-only access, without the cost of copying.)

You see that, despite the fact that the method `data` was defined as returning a reference, you still need to indicate whether the left-hand side is a reference.

See section 18.1 for the interaction between `const` and references.

## 15.4 Reference to array members

You can define various operator, such as `+-*/` arithmetic operators, to act on classes, with your own provided implementation; see section 9.5.6. You can also define the parentheses and square brackets operators, so make your object look like a function or an array respectively.

These mechanisms can also be used to provide safe access to arrays and/or vectors that are private to the object.

Suppose you have an object that contains an `int` array. You can return an element by defining the subscript (square bracket) operator for the class:

```
class vector10 {
private:
  int array[10];
public:
  /* ... */
  int operator()(int i) {
    return array[i];
  };
  int operator[](int i) {
    return array[i];
  };
```

```
};
  /* ... */
  vector10 v;
  cout << v(3) << '\n';
  cout << v[2] << '\n';
  /* compilation error: v(3) = -2; */
```

Note that `return array[i]` will return a copy of the array element, so it is not possible to write

```
myobject[5] = 6;
```

For this we need to return a reference to the array element:

```
  int& operator[](int i) {
    return array[i];
  };
};
  /* ... */
  cout << v[2] << '\n';
  v[2] = -2;
  cout << v[2] << '\n';
```

Your reason for wanting to return a reference could be to prevent the *copy of the return result* that is induced by the `return` statement. In this case, you may not want to be able to alter the object contents, so you can return a *const reference*:

```
  const int& operator[](int i) {
    return array[i];
  };
};
  /* ... */
  cout << v[2] << '\n';
  /* compilation error: v[2] = -2; */
```

## 15.5 rvalue references

See the chapter about obscure stuff; section 27.3.3.

# Chapter 16

# Pointers

Pointers are an indirect way of associating an entity with a variable name. Consider for instance the circular-sounding definition of a list:

> A list consists of nodes, where a node contains some information in its 'head', and which has a 'tail' that is a list.

Naive code:

```
class Node {
private:
  int value;
  Node tail;
  /* ... */
};
```

This does not work: would take infinite memory.

Indirect inclusion: only 'point' to the tail:

```
class Node {
private:
  int value;
  PointToNode tail;
  /* ... */
};
```

This chapter will explain C++ *smart pointer*s, and give some uses for them.

## 16.1 The 'arrow' notation

- If `x` is object with member `y`:
  `x.y`
- If `xx` is pointer to object with member `y`:
  `xx->y`
- In class methods **`this`** is a pointer to the object, so:

  ```
  class x {
  ```

```
  int y;
  x(int v) {
    this->y = v; }
}
```

- Arrow notation works with old-style pointers and new shared/unique pointers.

## 16.2 Making a shared pointer

Smart pointers are used the same way as old-style pointers in C. If you have an object `Obj x` with a member `y`, you access that with `x.y`; if you have a pointer `x` to such an object, you write `x->y`.

So what is the type of this latter `x` and how did you create it?

Same as C-pointer syntax:

**Code:**

```
#include <memory>
using std::make_shared;

  /* ... */
  HasX xobj(5);
  cout << xobj.value() << '\n';
  xobj.set(6);
  cout << xobj.value() << '\n';

  auto xptr = make_shared<HasX>(5);
  cout << xptr->value() << '\n';
  xptr->set(6);
  cout << xptr->value() << '\n';
```

**Output**

```
[pointer] pointx:
5
6
5
6
```

You may think that you first create an object, and then set a pointer to it, the way it happens in many other languages, but smart pointers work differently: you create the object and the pointer to it in one call.

```
make_shared< ClassName >( constructor arguments );
```

The resulting object is of type `shared_ptr<ClassName>`, but you can save yourself spelling that out, and use `auto` instead.

Simple class that stores one number:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto get() { return x; };
  void set(double xx) { x = xx; };
};
```

Allocation of object and pointer to it in one:

```
auto X = make_shared<HasX>( /* args */ );

// or explicitly:

shared_ptr<HasX> X =
    make_shared<HasX>( /* constructor args */ );
```

Using smart pointers requires at the top of your file:

```
#include <memory>
using std::shared_ptr;
using std::make_shared;

using std::unique_ptr;
using std::make_unique;
```

Why do we use pointers?
Pointers make it possible for two variables to own the same object.

**Code:**

```
auto xptr = make_shared<HasX>(5);
auto yptr = xptr;
cout << xptr->get() << '\n';
yptr->set(6);
cout << xptr->get() << '\n';
```

**Output**
**[pointer] twopoint:**

```
5
6
```

The constructor syntax is a little involved for vectors:

```
auto x = make_shared<vector<double>>(vector<double>{1.1,2.2});
```

### 16.2.1 Pointers and arrays

The prototypical example for the use of pointers is in linked lists and graph data structures. See section 66.1.2 for code and discussion.

**Exercise 16.1.** If you are doing the geometry project (chapter 47) you can now do exercise **??**.

### 16.2.2 Smart pointers versus address pointers

The oldstyle `&y` address pointer can not be made smart:

```
auto
  p = shared_ptr<HasY>( &y );
p->y = 3;
cout << "Pointer's y: "
     << p->y << '\n';
```

gives:

```
address(56325,0x7fff977cc380) malloc: *** error for object
    0x7ffeeb9caf08: pointer being freed was not allocated
```

Smart pointers are much better than old style pointers

```
Obj *X;
*X = Obj( /* args */ );
```

There is a final way of creating a shared pointer where you cast an old-style **new** object to shared pointer

```
auto p = shared_ptr<Obj>( new Obj );
```

This is not the preferred mode of creation, but it can be useful in the case of *weak pointer*s; section 16.5.4.

## 16.3 Memory leaks and garbage collection

The big problem with C-style pointers is the chance of a *memory leak*. If a pointer to a block of memory goes out of scope, the block is not returned to the Operating System (OS), but it is no longer accessible.

```
// the variable 'array' doesn't exist
{
  // attach memory to 'array':
  double *array = new double[25];
  // do something with array
}
// the variable 'array' does not exist anymore
// but the memory is still reserved.
```

Shared and unique pointers do not have this problem: if they go out of scope, or are overwritten, the destructor on the object is called, thereby releasing any allocated memory.

An example.

We need a class with constructor and destructor tracing:

```
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

Just to illustrate that the destructor gets called when the object goes out of scope:

**Code:**

```
cout << "Outside\n";
{
  thing x;
  cout << "create done\n";
}
cout << "back outside\n";
```

**Output**
**[pointer] ptr0:**

```
Outside
.. calling constructor
create done
.. calling destructor
back outside
```

Now illustrate that the destructor of the object is called when the pointer no longer points to the object. We do this by assigning `nullptr` to the pointer. (This is very different from `NULL` in C: the null pointer is actually an object with a type.)

Let's create a pointer and overwrite it:

**Code:**

```
cout << "set pointer1"
     << '\n';
auto thing_ptr1 =
  make_shared<thing>();
cout << "overwrite pointer"
     << '\n';
thing_ptr1 = nullptr;
```

**Output**
**[pointer] ptr1:**

```
set pointer1
.. calling constructor
overwrite pointer
.. calling destructor
```

However, if a pointer is copied, there are two pointers to the same block of memory, and only when both disappear, or point elsewhere, is the object deallocated.

**Code:**

```
cout << "set pointer2" << '\n';
auto thing_ptr2 =
  make_shared<thing>();
cout << "set pointer3 by copy"
     << '\n';
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
     << '\n';
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
     << '\n';
thing_ptr3 = nullptr;
```

**Output**
**[pointer] ptr2:**

```
set pointer2
.. calling constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

- The object counts how many pointers there are:
- 'reference counting'
- A pointed-to object is deallocated if no one points to it.

**Remark 11** *A more obscure source of memory leaks has to do with exceptions:*

```
void f() {
  double *x = new double[50];
  throw("something");
```

```
    delete x;
    }
```

*Because of the exception (which can of course come from a nested function call) the* `delete` *statement is never reached, and the allocated memory is leaked. Smart pointers solve this problem.*

## 16.4 More about pointers

### 16.4.1 Get the pointed data

Most of the time, accessing the target of the pointer through the arrow notation is enough. However, if you actually want the object, you can get it with `get`. Note that this does not give you the pointed object, but a traditional pointer.

```
X->y;
// is the same as
X.get()->y;
// is the same as
( *X.get() ).y;
```

**Code:**

```
auto Y = make_shared<HasY>(5);
cout << Y->y << '\n';
Y.get()->y = 6;
cout << ( *Y.get() ).y << '\n';
```

**Output**

```
[pointer] pointy:
5
6
```

## 16.5 Advanced topics

### 16.5.1 Unique pointers

Shared pointers are fairly easy to program, and they come with lots of advantages, such as the automatic memory management. However, they have more overhead than strictly necessary because they have a *reference count* mechanism to support the memory management. Therefore, there exists a *unique pointer*, `unique_ptr`, for cases where an object will only ever be 'owned' by one pointer. In that case, you can use a C-style *bare pointer* for non-owning references.

### 16.5.2 Base and derived pointers

Suppose you have base and derived classes:

```
class A {};
class B : public A {};
```

Just like you could assign a `B` object to an `A` variable:

```
B b_object;
A a_object = b_object;
```

is it possible to assign a `B` pointer to an `A` pointer?

The following construct makes this possible:

```
auto a_ptr = shared_pointer<A>( make_shared<B>() );
```

Note: this is better than

```
auto a_ptr = shared_pointer<A>( new B() );
```

Again a reason we don't need `new` anymore!

### 16.5.3 Shared pointer to 'this'

Inside an object method, the object is accessible as `this`. This is a pointer in the classical sense. So what if you want to refer to 'this' but you need a shared pointer?

For instance, suppose you're writing a linked list code, and your `node` class has a method `prepend_or_append` that gives a shared pointer to the new head of the list.

Your code would start something like this, handling the case where the new node is appended to the current:

```
shared_pointer<node> node::append
    ( shared_ptr<node> other ) {
  if (other->value>this->value) {
    this->tail = other;
```

But now you need to return this node, as a shared pointer. But `this` is a `node*`, not a `shared_ptr<node>`.

The solution here is that you can return

```
    return this->shared_from_this();
```

if you have defined your node class to inherit from what probably looks like magic:

```
class node : public enable_shared_from_this<node>
```

Note that you can only return a `shared_from_this` if already a valid shared pointer to that object exists.

### 16.5.4 Weak pointers

In addition to shared and unique pointers, which own an object, there is also `weak_ptr` which creates a *weak pointer*. This pointer type does not own, but at least it knows when it dangles.

```
weak_ptr<R> wp;
shared_ptr<R> sp( new R );
sp->call_member();
wp = sp;
// access through new shared pointer:
auto p = wp.lock();
if (p) {
  p->call_member();
}
if (!wp.expired()) { // not thread-safe!
  wp.lock()->call_member();
}
```

There is a subtlety with weak pointers and shared pointers. The call

```
auto sp = shared_ptr<Obj>( new Obj );
```

creates first the object, then the 'control block' that counts owners. On the other hand,

```
auto sp = make_shared<Obj>();
```

does a single allocation for object and control block. However, if you have a weak pointer to such a single object, it is possible that the object is destructed, but not de-allocated. On the other hand, using

```
auto sp  = shared_ptr<Obj>( new Obj );
```

creates the control block separately, so the pointed object can be destructed and de-allocated. Having a weak pointer to it means that only the control block needs to stick around.

### 16.5.5 Null pointer

In C there was a macro *NULL* that, only by convention, was used to indicate *null pointers*: pointers that do not point to anything. C++ has the `nullptr`, which is an object of type `std::nullptr_t`.

There are some scenarios where this is useful, for instance, with polymorphic functions:

```
void f(int);
void f(int*);
```

Calling `f(ptr)` where the point is `NULL`, the first function is called, whereas with `nullptr` the second is called.

Unfortunately, `dereferencing a` *nullptr* does not give an exception.

### 16.5.6 Opaque pointer

The need for *opaque pointers* `void*` is a lot less in C++ than it was in C. For instance, contexts can often be modeled with captures in closures (chapter 13). If you really need a pointer that does not *a priori* knows what it points to, use `std::any`, which is usually smart enough to call destructors when needed.

**Code:**

```
std::any a = 1;
cout << a.type().name() << ": "
     << std::any_cast<int>(a) << '\n';
a = 3.14;
cout << a.type().name() << ": "
     << std::any_cast<double>(a) <<
    '\n';
a = true;
cout << a.type().name() << ": "
     << std::any_cast<bool>(a) << '\n';

try {
  a = 1;
  cout << std::any_cast<float>(a) <<
    '\n';
} catch (const std::bad_any_cast& e) {
  cout << e.what() << '\n';
}
```

**Output**
**[pointer] any:**

```
i: 1
d: 3.14
b: true
bad any cast
```

### 16.5.7 Pointers to non-objects

In the introduction to this chapter we argued that many of the uses for pointers that existed in C have gone away in C++, and the main one left is the case where multiple objects share 'ownership' of some other object.

You can still make shared pointers to scalar data, for instance to an array of scalars. You then get the advantage of the memory management, but you do not get the `size` function and such that you would have if you'd used a `vector` object.

Here is an example of a pointer to a solitary double:

**Code:**

```
// shared pointer to allocated double
auto array = shared_ptr<double>( new
    double );
double *ptr = array.get();
array.get()[0] = 2.;
cout << ptr[0] << '\n';
```

**Output**
**[pointer] ptrdouble:**

```
2
```

It is possible to initialize that double:

**Code:**

```
// shared pointer to initialized double
auto array = make_shared<double>(50);
double *ptr = array.get();
cout << ptr[0] << '\n';
```

**Output**
**[pointer] ptrdoubleinit:**

```
50
```

## 16.6 Smart pointers vs C pointers

We remark that there is less need for pointers in C++ than there was in C.

- To pass an argument *by reference*, use a *reference*. Section 7.5.
- Strings are done through `std::string`, not character arrays; see 11.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see 10.
- Traversing arrays and vectors can be done with ranges; section 10.2.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

Legitimate needs:

- Linked lists and Directed Acyclic Graphs (DAGs); see the example in section 66.1.2.
- Objects on the heap.
- Use `nullptr` as a signal.

# Chapter 17

# C-style pointers and arrays

## 17.1 What is a pointer

The term pointer is used to denote a reference to a quantity. The reason that people like to use C as high performance language is that pointers are actually memory addresses. So you're programming 'close to the bare metal' and are in far going control over what your program does. C++ also has pointers, but there are fewer uses for them than for C pointers: vectors and references have made many of the uses of C-style pointers obsolete.

## 17.2 Pointers and addresses, C style

You have learned about variables, and maybe you have a mental concept of variables as 'named memory locations'. That is not too far of: while you are in the (dynamic) scope of a variable, it corresponds to a fixed memory location.

**Exercise 17.1.** When does a variable not always correspond to the same location in memory?

There is a mechanism of finding the actual address of a variable: you prefix its name by an ampersand. This address is integer-valued, but its range is actually greater than of the `int` type.

If you have an `int i`, then `&i` is the address of `i`.

An address is a (long) integer, denoting a memory address. Usually it is rendered in *hexadecimal* notation.

**Code:**

```
int i;
cout << "address of i, decimal: "
     << (long)&i << '\n';
cout << "address of i, hex    : "
     << std::hex << &i << '\n';
```

**Output**

```
[pointer] coutpoint:
address of i, decimal:
    140732703427524
address of i, hex    :
    0x7ffee2cbcbc4
```

Using purely C:

**Code:**

```
int i;
printf("address of i: %ld\n",
       (long)(&i));
printf(" same in hex: %lx\n",
       (long)(&i));
```

**Output**
**[pointer] printfpoint:**

```
address of i: 140732690693076
 same in hex: 7ffee2097bd4
```

Note that this use of the ampersand is different from defining references; compare section 7.5.2. However, there is never a confusion which is which since they are syntactically different.

You could just print out the address of a variable, which is sometimes useful for debugging. If you want to store the address, you need to create a variable of the appropriate type. This is done by taking a type and affixing a star to it.

The type of '`&i`' is `int*`, pronounced 'int-star',
or more formally: 'pointer-to-int'.

You can create variables of this type:

```
int i;
int* addr = &i;
// exactly the same:
int *addr = &i;
```

Now `addr` contains the memory address of `i`.

Now if you have have a pointer that refers to an int:

```
int i;
int *iaddr = &i;
```

you can use (for instance print) that pointer, which gives you the address of the variable. If you want the value of the variable that the pointer points to, you need to *dereference* it.

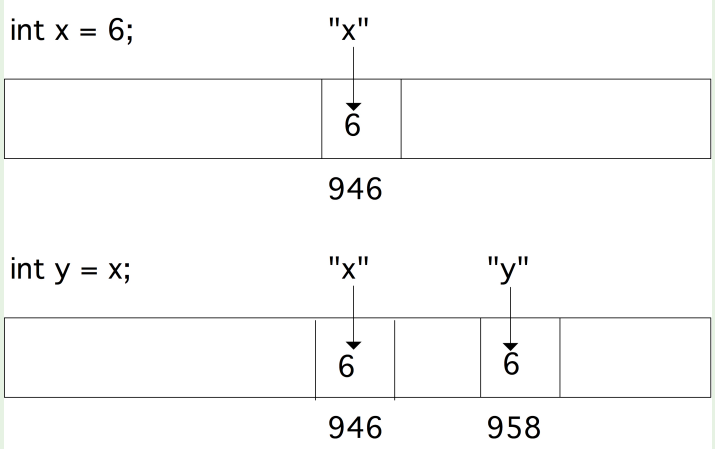
Using `*addr` 'dereferences' the pointer: gives the thing it points to;
the value of what is in the memory location.

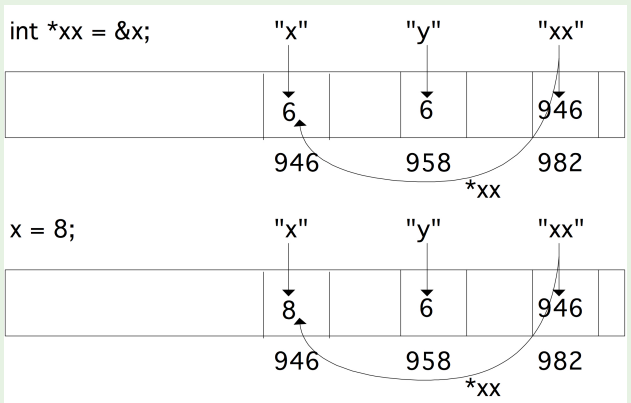
**Code:**

```
int i;
int* addr = &i;
i = 5;
cout << *addr << '\n';
i = 6;
cout << *addr << '\n';
```

**Output**
**[pointer] cintpointer:**

```
5
6
```




- `addr` is the address of `i`.
- You set `i` to 5; nothing changes about `addr`. This has the effect of writing `5` in the memory location of `i`.
- The first `cout` line dereferences `addr`, that is, looks up what is in that memory location.
- Next you change `i` to `6`, that is, you write `6` in its memory location.
- The second `cout` looks in the same memory location as before, and now finds `6`.

The syntax for declaring a pointer-to-sometype allows for a small variation, which indicates the two way you can interpret such a declaration.

Equivalent:

- `int* addr`: `addr` is an int-star, or
- `int *addr`: `*addr` is an int.

The notion `int* addr` is equivalent to `int *addr`, and semantically they are also the same: you could say that `addr` is an int-star, or you could say that `*addr` is an int.

## 17.3 Arrays and pointers

In section 10.10 you saw the treatment of static arrays in C++. Examples such as:

```
void set_array( double *x,int size) {
  for (int i=0; i<size; i++)
    x[i] = 1.41;
};
  /* ... */
  double array[5] = {11,22,33,44,55};
  set_array(array,5);
  cout << array[0] << "...." << array[4] << '\n';
```

show that, even though an parameters are normally passed by value, that is through copying, array parameters can be altered. The reason for this is that there is no actual array type, and what is passed is a pointer to the first element of the array. So arrays are still passed by value, just not the 'value of the array', but the value of its location.

So you could pass an array like this:

```
void array_set_star( double *ar,int idx,double val) {
  ar[idx] = val;
}
  /* ... */
  array_set_star(array,2,4.2);
```

Array and memory locations are largely the same:

**Code:**

```
1 double array[5] = {11,22,33,44,55};
2 double *addr_of_second = &(array[1]);
3 cout << *addr_of_second << '\n';
4 array[1] = 7.77;
5 cout << *addr_of_second << '\n';
```

**Output**
**[pointer] arrayaddr:**

```
22
7.77
```

When an array is passed to a function, it behaves as an address:

**Code:**

```
1 void set_array( double *x,int size) {
2   for (int i=0; i<size; i++)
3     x[i] = 1.41;
4 };
5   /* ... */
6   double array[5] = {11,22,33,44,55};
7   set_array(array,5);
8   cout << array[0] << "...." << array[4]
      << '\n';
```

**Output**
**[pointer] arraypass:**

```
1.41....1.41
```

Note that these arrays don't know their size, so you need to pass it.

You can dynamically reserve memory with **new**, which gives a something-star:

```
double *x;
x = new double[27];
```

The **new** operator is only for C++: in C you would use *malloc* to dynamically allocate memory. The above example would become:

```
double *x;
x = (double*) malloc( 27 * sizeof(double) );
```

Note that **new** takes the number of elements, and deduces the type (and therefore the number of bytes per element) from the context; *malloc* takes an argument that is the number of bytes. The **sizeof** operator then helps you in determining the number of bytes per element.

## 17.4 Pointer arithmetic

*pointer arithmetic* uses the size of the objects it points at:

```
double *addr_of_element = array;
cout << *addr_of_element;
addr_of_element = addr_of_element+1;
cout << *addr_of_element;
```

Increment add size of the array element, 4 or 8 bytes, not one!

**Exercise 17.2.** Write a subroutine that sets the i-th element of an array, but using pointer arithmetic: the routine should not contain any square brackets.

## 17.5 Multi-dimensional arrays

After

```
double x[10][20];
```

a row `x[3]` is a **double***, so is `x` a **double****?

Was it created as:

```
double **x = new double*[10];
for (int i=0; i<10; i++)
  x[i] = new double[20];
```

No: multi-d arrays are contiguous.

## 17.6 Parameter passing

C++ style functions that alter their arguments:

```
void inc(int &i) {
  i += 1;
}
int main() {
  int i=1;
  inc(i);
  cout << i << endl;
```

```
  return 0;
}
```

In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```
void inc(int *i) {
  *i += 1;
}
int main() {
  int i=1;
  inc(&i);
  cout << i << endl;
  return 0;
}
```

Now the function gets an argument that is a memory address: `i` is an int-star. It then increases `*i`, which is an int variable, by one.

Note how again there are two different uses of the ampersand character. While the compiler has no trouble distinguishing them, it is a little confusing to the programmer.

**Exercise 17.3.** Write another version of the `swap` function:

```
void swap( /* something with i and j */ {
  /* your code */
}
int main() {
  int i=1,j=2;
  swap( /* something with i and j */ );
  cout << "check that i is 2: " << i << endl;
  cout << "check that j is 1: " << i << endl;
  return 0;
}
```

Hint: write C++ code, then insert stars where needed.

### 17.6.1 Allocation

In section 10.10 you learned how to create arrays that are local to a scope:

Create an array with size depending on something:

```
if ( something ) {
  double ar[25];
} else {
  double ar[26];
}
ar[0] = // there is no array!
```

This Does Not Work

The array `ar` is created depending on if the condition is true, but after the conditional it disappears again. The mechanism of using **new** (section 17.6.2) allows you to allocate storage that transcends its scope:

C allocates in bytes:

```
double *array;
array  = (double*) malloc( 25*sizeof(double) );
```

C++ allocates an array:

```
double *array;
array  = new double[25];
```

Don't forget:

```
free(array); // C
delete array; // C++
```

Now dynamic allocation:

```
double *array;
if (something) {
  array = new double[25];
} else {
  array = new double[26];
}
```

Don't forget:

```
delete array;
```

```
void func() {
  double *array = new double[large_number];
  // code that uses array
}
int main() {
  func();
};
```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- $\Rightarrow$ *memory leak*.

```
for (int i=0; i<large_num; i++) {
  double *array = new double[1000];
  // code that uses array
}
```

Every iteration reserves memory, which is never released: another *memory leak*.

Your code will run out of memory!

Memory allocated with `malloc` / new does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```
free(array);
delete(array);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

No need for `malloc` or **`new`**

- Use **`std`**`::string` for character arrays, and
- **`std`**`::vector` for everything else.

No performance hit if you don't dynamically alter the size.

#### *17.6.1.1 Malloc*

The keywords **`new`** and **`delete`** are in the spirit of C style programming, but don't exist in C. Instead, you use `malloc`, which creates a memory area with a size expressed in bytes. Use the function **`sizeof`** to translate from types to bytes:

```
int n;
double *array;
array = malloc( n*sizeof(double) );
if (!array)
  // allocation failed!
```

#### *17.6.1.2 Allocation in a function*

The mechanism of creating memory, and assigning it to a 'star' variable can be used to allocate data in a function and return it from the function.

```
void make_array( double **a, int n ) {
  *a = new double[n];
}
int main() {
  double *array;
  make_array(&array,17);
}
```

Note that this requires a 'double-star' or 'star-star' argument:

- The variable `a` will contain an array, so it needs to be of type **`double`**`*`;
- but it needs to be passed by reference to the function, making the argument type **`double`**`**`;
- inside the function you then assign the new storage to the **`double`**`*` variable, which is `*a`.

Tricky, I know.

### 17.6.2 Use of `new`

*Before doing this section, make sure you study section 17.3.*

There is a dynamic allocation mechanism that is much inspired by memory management in C. Don't use this as your first choice.

Use of **new** uses the equivalence of array and reference.

```
void make_array( int **new_array, int length ) {
  *new_array = new int[length];
}
  /* ... */
  int *the_array;
  make_array(&the_array,10000);
```

Since this is not scoped, you have to free the memory yourself:

```
class with_array{
private:
  int *array;
  int array_length;
public:
  with_array(int size) {
    array_length = size;
    array = new int[size];
  };
  ~with_array() {
    delete array;
  };
};
  /* ... */
  with_array thing_with_array(12000);
```

Notice how you have to remember the array length yourself? This is all much easier by using a **std**::*vector*. See http://www.cplusplus.com/articles/37Mf92yv/.

The **new** mechanism is a cleaner variant of *malloc*, which was the dynamic allocation mechanism in C. Malloc is still available, but should not be used. There are even very few legitimate uses for **new**.

## 17.7 Memory leaks

Pointers can lead to a problem called a *memory leak*: there is memory that you have reserved, but you have lost the ability to access it.

In this example:

```
double *array = new double[100];
// ...
array = new double[105];
```

memory is allocated twice. The memory that was allocated first is never release, because in the intervening code another pointer to it may have been set. However, if that doesn't happen, the memory is both allocated, and unreachable. That's what memory leaks are about.

## 17.8 Const pointers

A pointer can be constant in two ways:

1. It points to a block of memory, and you can not change where it points.
2. What it points to is fixed, and the contents of that memory can also not be altered.

To illustrate the non-const behavior:

**Code:**

```
int value = 5;
int *ptr = &value;
*ptr += 1;
cout << "value: " << value << '\n';
cout << "*ptr: " << *ptr << '\n';
ptr += 1;
cout << "random memory: " << *ptr <<
    '\n';
```

**Output**
**[pointer] starconst1:**

```
value: 6
*ptr: 6
random memory: 73896
```

A pointer that is constant in the first sense:

```
int value = 5;
int *const ptr = &value;
*ptr += 1;
/* DOES NOT COMPILE: cannot assign to variable 'ptr' with const-qualified type
    'int *const'
ptr += 1;
*/
```

You can also make a pointer to a constant integer:

```
const int value = 5; // value is const
/* DOES NOT COMPILE: cannot convert const int* to int*
int *ptr = &value;
*/
```

# Chapter 18

# Const

The keyword `const` can be used to indicate that various quantities can not be changed. This is mostly for programming safety: if you declare that a method will not change any members, and it does so (indirectly) anyway, the compiler will warn you about this.

## 18.1 Const arguments

The use of const arguments is one way of protecting you against yourself. If an argument is conceptually supposed to stay constant, the compiler will catch it if you mistakenly try to change it.

Function arguments marked **const** can not be altered by the function code. The following segment gives a compilation error:

```
void f(const int i) {
  i++;
}
```

## 18.2 Const references

A more sophisticated use of const is the *const reference*:

```
void f( const int &i ) { .... }
```

This may look strange. After all, references, and the pass-by-reference mechanism, were introduced in section 7.5 to return changed values to the calling environment. The const keyword negates that possibility of changing the parameter.

But there is a second reason for using references. Parameters are passed by value, which means that they are copied, and that includes big objects such as `std::vector`. Using a reference to pass a vector is much less costly in both time and space, but then there is the possibility of changes to the vector propagating back to the calling environment.

Consider a class that has methods that return an internal member by reference, once as const reference and once not:

**Code:**

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return
    mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc();
    an_int.inc();
  cout << "Contained int is now: "
       << an_int.int_to_get() << '\n';
  /* Compiler error:
    an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: "
       << an_int.int_to_get() << '\n';
```

**Output**
**[const] constref:**

```
Contained int is now: 4
Contained int is now: 17
```

We can make visible the difference between pass by value and pass by const-reference if we define a class where the *copy constructor* explicitly reports itself:

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v
         << '\n';
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v
         << '\n';
    mine = v; };
  void printme() {
    cout << "I have: " << mine
         << '\n'; };
};
```

Now if we define two functions, with the two parameter passing mechanisms, we see that passing by value invokes the copy constructor, and passing by const reference does not:

**Code:**

```
void f_with_copy(has_int other) {
  cout << "function with copy" << '\n';
    };
void f_with_ref(const has_int &other) {
  cout << "function with ref" << '\n';
    };
/* ... */
cout << "Calling f with copy..." <<
  '\n';
f_with_copy(an_int);

cout << "Calling f with ref..." <<
  '\n';
f_with_ref(an_int);
```

**Output**
**[const] constcopy:**

```
Calling f with copy...
(calling copy constructor)
function with copy
Calling f with ref...
function with ref
... done
```

### 18.2.1 Const references in range-based loops

The same pass by value/reference issue comes up in range-based for loops. The syntax

```
for ( auto v : some_vector )
```

copies the vector elements to the `v` variable, whereas

```
for ( auto& v : some_vector )
```

makes a reference. To get the benefits of references (no copy cost) while avoiding the pitfalls (inadvertent changes), you can also use a const-reference here:

```
for ( const auto& v : some_vector )
```

## 18.3 Const methods

We can distinguish two types of methods: those that alter internal data members of the object, and those that don't. The ones that don't can be marked **const**. While this is in no way required, it contributes to a clean programming style:

Using **const** will catch mismatches between the prototype and definition of the method. For instance,

```
class Things {
  private:
  int var;
  public:
  f(int &ivar,int c) const {
    var += c; // typo: should be 'ivar'
  }
}
```

Here, the use of `var` was a typo, should have been `ivar`. Since the method is marked `const`, the compiler will generate an error.

It encourages a functional style, in the sense that it makes *side-effects* impossible:

```
class Things {
private:
  int i;
public:
  int get() const { return i; }
  int inc() { return i++; } // side-effect!
  void addto(int &thru) const { thru += i; }
}
```

## 18.4 Overloading on const

A const method and its non-const variant are different enough that you can use this for overloading.

**Code:**

```
class has_array {
private:
  vector<float> values;;
public:
  has_array(int l,float v)
    : values(vector<float>(l,v)) {};
  auto& at(int i) {
    cout << "var at" << '\n';
    return values.at(i); };
  const auto& at (int i) const {
    cout << "const at" << '\n';
    return values.at(i); };
  auto sum() const {
    float p;
    for ( int i=0; i<values.size(); i++)
      p += at(i);
    return p;
  };
};

int main() {

  int l; float v;
  cin >> l; cin >> v;
  has_array fives(l,v);
  cout << fives.sum() << '\n';
  fives.at(0) = 2;
  cout << fives.sum() << '\n';
```

**Output**

```
[const] constat:

const at
const at
const at
1.5
var at
const at
const at
const at
4.5
```

**Exercise 18.1.** Explore variations on this example, and see which ones work and which ones not.

1. Remove the second definition of `at`. Can you explain the error?
2. Remove either of the `const` keywords from the second `at` method. What errors do you get?

## 18.5 Const and pointers

Let's declare a class `thing` to point to, and a class `has_thing` that contains a pointer to a `thing`.

```
class thing {
private:
  int i;
public:
  thing(int i) : i(i) {};
  void set_value(int ii) { i = ii; };
  auto value() const { return i; };
};

class has_thing {
private:
  shared_ptr<thing>
      thing_ptr{nullptr};
public:
  has_thing(int i)
    : thing_ptr
      (make_shared<thing>(i)) {};
  void print() const {
    cout << thing_ptr->value() <<
    '\n'; };
```

If we define a method to return the pointer, we get a copy of the pointer, so redirecting that pointer has no effect on the container:

**Code:**

```
auto get_thing_ptr() const {
  return thing_ptr; };
/* ... */
  has_thing container(5);
  container.print();
  container.get_thing_ptr() =
    make_shared<thing>(6);
  container.print();
```

**Output**
**[const] constpoint2:**

```
5
5
```

If we return the pointer by reference we can change it. However, this requires the method not to be `const`. On the other hand, with the `const` method earlier we can change the object:

**Code:**

```
// Error: does not compile
// auto &get_thing_ptr() const {
auto &access_thing_ptr() {
  return thing_ptr; };
/* ... */
  has_thing container(5);
  container.print();
  container.access_thing_ptr() =
    make_shared<thing>(7);
  container.print();

    container.get_thing_ptr()->set_value(8);
  container.print();
```

**Output**
**[const] constpoint3:**

```
5
7
8
```

If you want to prevent the pointed object from being changed, you can declare the pointer as a `shared_ptr<const thing>`:

```
private:
  shared_ptr<const thing>
      const_thing{nullptr};
public:
  has_thing(int i,int j)
    : const_thing
      (make_shared<thing>(i+j)) {};
  auto get_const_ptr() const {
    return const_thing; };
```

```
  void crint() const {
    cout << const_thing->value() <<
    '\n'; };
/* ... */
  has_thing constainer(1,2);
  // Error: does not compile

  constainer.get_const_ptr()->set_value(9);
```

### 18.5.1 Old-style const pointers

For completeness, a section on const and pointers in C.

We can have the `const` keyword in three places in the declaration of a C pointer:

```
int *p;
const int *p;
int const * p;  // this is the same
int * const p;  // as this
```

For the interpretation, it is often said to read the declaration 'from right to left'. So:

```
int * const p;
```

is a 'const pointer to int'. This means that it is const what int it points to, but that int itself can change:

```
int i=5;
int * const ip = &i;
printf("ptr derefs to: %d\n",*ip);
*ip = 6;
printf("ptr derefs to: %d\n",*ip);
int j;
// DOES NOT COMPILE ip = &j;
```

On the other hand,

```
const int *p;
```

is a 'pointer to a const int'. This means that you can point it at different ints, but you can not change the value those through the pointer:

```
const int * jp = &i;
i = 7;
printf("ptr derefs to: %d\n",*jp);
// DOES NOT COMPILE *jp = 8;
int k = 9;
jp = &k;
printf("ptr derefs to: %d\n",*jp);
```

Finally,

```
const int * const p;
```

is a 'const pointer to const int'. This pointer can not be retargeted, and the value of its target can not be changed:

```
// DOES NOT WORK const int * const kp; kp = &k;
const int * const kp = &k;
printf("ptr derefs to: %d\n",*kp);
k = 10;
// DOES NOT COMPILE *kp = 11;
```

Because it can not be retargeted, you have to set its target when you declare such a pointer.

## 18.6 Mutable

Typical class with non-const update methods, and const readout of some computed quantity:

```
class Stuff {
private:
  int i,j;
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; };
  void setj(int jnew) { j = jnew; };
  int result () const { return i+j; };
```

Attempt at caching the computation:

```
class Stuff {
private:
  int i,j;
  int cache;
  void compute_cache() { cache = i+j; };
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; compute_cache(); };
  void setj(int jnew) { j = jnew; compute_cache(); };
  int result () const { return cache; };
```

But what if setting happens way more often than getting?

```
class Stuff {
private:
  int i,j;
  int cache;
  bool cache_valid{false};
  void update_cache() {
    if (!cache_valid) {
      cache = i+j; cache_valid = true;
  };
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; cache_valid = false; };
  void setj(int jnew) { j = jnew; cache_valid = false; }
  int result () const {
    update_cache(); return cache; };
```

This does not compile, because `result` is const, but it calls a non-const function.

We can solve this by declaring the cache variables to be **mutable**. Then the methods that conceptually don't change the object can still stay **const**, even while altering the state of the object. (It is better not to use **const_cast**.)

```
class Stuff {
private:
  int i,j;
  mutable int cache;
  mutable bool cache_valid{false};
  void update_cache() const {
    if (!cache_valid) {
      cache = i+j; cache_valid = true;
  };
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; cache_valid = false; };
  void setj(int jnew) { j = jnew; cache_valid = false; }
  int result () const {
    update_cache(); return cache; };
```

## 18.7 Compile-time constants

Compilers have long been able to simplify expressions that only contain constants:

```
int i=5;
int j=i+4;
f(j)
```

Here the compiler will conclude that `j` is 9, and that's where the story stops. it also becomes possible to let `f(j)` be evaluated by the compiler, if the function `f` is simple enough. C++17 added several more variants of *constexpr* usage.

The **const** keyword can be used to indicate that a variable can not be changed:

```
const int i=5;
// DOES NOT COMPILE:
i += 2;
```

The combination **if** *constexpr* is useful with templates:

```
template <typename T>
auto get_value(T t) {
  if constexpr (std::is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

To declare a function to be constant, use *constexpr*. The standard example is:

```
constexpr double pi() {
  return 4.0 * atan(1.0); };
```

but also

```
constexpr int factor(int n) {
  return n <= 1 ? 1 : (n*fact(n-1));
}
```

(Recursion in C++11, loops and local variables in C++14.)

- Can use conditionals, if testing on constant data;
- can use loops, if number of iterations constant;
- C++20 can allocate memory, if size constant.

# Chapter 19

# Declarations and header files

In this chapter you will learn techniques that you need for modular program design.

## 19.1 Include files

You have seen the `#include` directive in the context of *header files* of the STL, most notably the `iostream` header. But you can include arbitrary files, including your own.

To include files of your own, use a slightly different syntax:

```
#include "myfile.cxx"
```

(The angle bracket notation usually only works with files that are in certain system locations.) This statement acts as if the file is literally inserted at that location of the source.

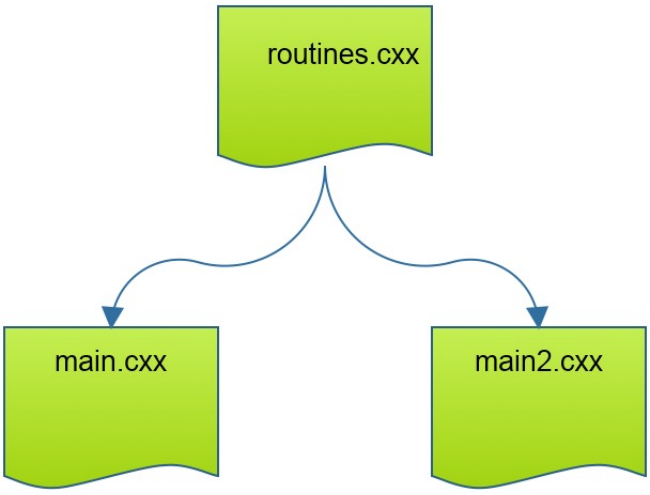



Figure 19.1: Using an include file for two main programs

This mechanism gives an easy solution to the problem of using some functions or classes in more than one main program; see figure 19.1.

The problem with this approach is that building the two main programs takes a time (roughly) equal to the sum of the compile times of the main programs and **twice** the compile time of the included file. Also, any time you change the included file you need to recompile the two main programs.

In a better scenario you would compile all three files once, and spend some little extra time tie-ing it all together. We will work towards this in a number of steps.

## 19.2 Function declarations

In most of the programs you have written in this course, you put any functions or classes above the main program, so that the compiler could inspect the definition before it encountered the use. However, the compiler does not actually need the whole definition, say of a function: it is enough to know its name, the types of the input parameters, and the return type.

Such a minimal specification of a function is known as *function prototype*; for instance

```
int tester(float);
```

A first use of declarations is *forward declarations*.

Some people like defining functions after the main:

```
int f(int);
int main() {
  f(5);
};
int f(int i) {
  return i;
}
```

versus before:

```
int f(int i) {
  return i;
}
int main() {
  f(5);
};
```

You also need forward declaration for mutually recursive functions:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

Declarations are useful if you spread your program over multiple files. You would put your functions in one file and the main program in another.

```
// file: def.cxx
int tester(float x) {
  .....
}
```

```
// file : main.cxx
int main() {
  int t = tester(...);
  return 0;
}
```

In this example a function `tester` is defined in a different file from the main program, so we need to tell `main` what the function looks like in order for the main program to be compilable:

```
// file : main.cxx
int tester(float);
int main() {
  int t = tester(...);
  return 0;
}
```

Splitting your code over multiple files and using *separate compilation* is good software engineering practice for a number of reasons.

1. If your code gets large, compiling only the necessary files cuts down on compilation time.
2. Your functions may be useful in other projects, by yourself or others, so you can reuse the code without cutting and pasting it between projects.
3. It makes it easier to search through a file without being distracted by unrelated bits of code.

(However, you would not do things like this in practice. See section 19.2.2 about header files.)

### 19.2.1 Separate compilation

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile
   ```
   icpc -c yourfile.cc
   ```
   which gives you a file `yourfile.o`, a so-called *object file*; and
2. Then you use the compiler as *linker* to give you the *executable file*:
   ```
   icpc -o yourprogram yourfile.o
   ```

In this particular example you may wonder what the big deal is. That will become clear if you have multiple source files: now you invoke the compile line for each source, and you link them only once.

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

At this point, you should learn about the *Make* tool for managing your programming project.

### 19.2.2 Header files

Even better than writing the declaration every time you need the function is to have a *header file*:

Header file contains only declaration:

```
// file: def.h
int tester(float);
```

The header file gets included both in the definitions file and the main program:

```
// file: def.cxx
#include "def.h"
int tester(float x) {
  .....
}
```

```
// file : main.cxx
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

What happens if you leave out the `#include "def.h"` in both cases?

Having a header file is an important safety measure:

- Suppose you change your function definition, changing its return type:
- The compiler will complain when you compile the definitions file;
- So you change the declaration in the header file;
- Now the compiler will complain about the main program, so you edit that too.

It is necessary to include the header file in the main program. It is not strictly necessary to include it in the definitions file, but doing so means that you catch potential errors: if you change the function definitions, but forget to update the header file, this is caught by the compiler.

**Remark 12** *By the way, why does that compiler even recompile the main program, even though it was not changed? Well, that's because you used a* makefile. *See the tutorial.*

**Remark 13** *Header files were able to catch more errors in C than they do in C++. With polymorphism of functions, it is no longer an error to have*

```
// header.h
int somefunction(int);
```

*and*

```
#include "header.h"

int somefunction( float x ) { .... }
```

### 19.2.3 C and C++ headers

You have seen the following syntaxes for including header files:

```
#include <header.h>
#include "header.h"
```

The first is typically used for system files, with the second typically for files in your own project. There are some header files that come from the C standard library such as `math.h`; the idiomatic way of including them in C++ is

```
#include <cmath>
```

## 19.3 Declarations for class methods

Section 9.5.2 explained how you can split a class declaration from its definition. You can then put the declaration in a header file that you include where is a class is used, while the definitions get compiled only once, and linked in when the executable of your program is built.

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file:

```
double something::dosomething( int i, char c ) {
  // do something with i,c
};
```

Data members, even private ones, need to be in the header file:

```
class something {
private:
  int localvar;
public:
  // declaration:
  double somedo(vector);
};
```

Implementation file:

```
// definition
double something::somedo(vector v) {
   .... something with v ....
   .... something with localvar ....
};
```

**Review** 19.1. For each of the following answer: is this a valid function definition or function declaration.

- `int foo();`
- `int foo() {};`
- `int foo(int) {};`

- `int foo(int bar) {};`
- `int foo(int) { return 0; };`
- `int foo(int bar) { return 0; };`

## 19.4 Header files and templates

See section 22.2.3.

## 19.5 Namespaces and header files

Namespaces (see chapter 20) are convenient, but they carry a danger in that they may define functions without the user of the namespace being aware of it.

Therefore, one should never put **`using namespace`** in a header file.

## 19.6 Global variables and header files

If you have a variable that you want known everywhere, you can make it a *global variable*:

```
int processnumber;
void f() {
  ... processnumber ...
}
int main() {
  processnumber = // some system call
};
```

It is then defined in the main program and any functions defined in your program file.

Warning: it is tempting to define variables global but this is a dangerous practice.

If your program has multiple files, you should not put '`int processnumber`' in the other files, because that would create a new variable, that is only known to the functions in that file. Instead use:

```
extern int processnumber;
```

which says that the global variable `processnumber` is defined in some other file.

What happens if you put that variable in a *header file*? Since the *preprocessor* acts as if the header is textually inserted, this again leads to a separate global variable per file. The solution then is more complicated:

```
//file: header.h
#ifndef HEADER_H
#define HEADER_H
#ifndef EXTERN
#define EXTERN extern
#fi
EXTERN int processnumber
#fi
```

```
//file: aux.cc
#include "header.h"

//file: main.cc
#define EXTERN
#include "header.h"
```

(This sort of preprocessor magic is discussed in chapter 21.)

This also prevents recursive inclusion of header files.

## 19.7 Modules

The C++20 standard is taking a different approach to header files, through the introduction of *modules*. (Fortran90 already had this for a long time; see chapter 38.) This largely dispenses with the need for header files included through the C Preprocessor (CPP). However, the CPP may still be needed for other purposes.

### 19.7.1 Program structure with modules

Using modules, the `#include` directive is no longer needed; instead, the `import` keyword indicates what module is to be used in a (sub)program:

```
import fg_module;
int main() {
  std::cout << "Hello world " << f(5) << '\n';
```

The module is in a different file; the `export` keyword defines the name of the module. This file can then have any number of functions; only the ones with the `export` keyword will be visible in a program that imports the module.

```
export module fg_module;

// internal function
int g( int i ) { return i/2; };

// exported function
export int f( int i ) {
  return g(i+1);
};
```

### 19.7.2 Implementation and interface units

A module can have a leveled structure, by using names with a `module:partition` structure.

This makes it possible to have separate

- Interface partitions, that define the interface to the using program; and
- Implementation partitions, that contain the code that needs to be shielded from the user.

Here is an implementation partition; there is no `import` keyword because this functionality is internal:

```
// implementation unit, nothing exported
module helper_module:helper;
// internal function
int g( int i ) { return i/2; };
```

Here is an interface partition, which uses the internal function, and exports a different function:

```
export module helper_module;
import :helper;

// exported function
export int f( int i ) {
  return g(i+1);
};
```

### 19.7.3 More

Importable headers:

```
import <header.h>
import "header.h"
```

Import declarations have to come before other module specifications, whether import or export of modules or functions.

You can export variables, namespaces.

# Chapter 20

# Namespaces

## 20.1 Solving name conflicts

In section 10.3 you saw that the C++ STL comes with a *vector* class, that implements dynamic arrays. You say

```
std::vector<int> bunch_of_ints;
```

and you have an object that can store a bunch of ints. And if you use such vectors often, you can save yourself some typing by `using namespace`. You put

```
using namespace std;
```

somewhere high up in your file, and write

```
vector<int> bunch_of_ints;
```

in the rest of the file.

More safe:

```
using std::vector;
```

But what if you are writing a geometry package, which includes a vector class? Is there confusion with the STL vector class? There would be if it weren't for the phenomenon *namespace*, which acts as as disambiguating prefix for classes, functions, variables.

You have already seen namespaces in action when you wrote `std::vector`: the '`std`' is the name of the namespace.

You can make your own namespace by writing

```
namespace a_namespace {
  // definitions
  class an_object {
  };
|
```

so that you can write

Qualify type with namespace:

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;
an_object myobject();
```

or

```
using a_namespace::an_object;
an_object myobject();
```

or

```
using namespace abc = space_a::space_b::space_c;
abc::func(x)
```

### 20.1.1 Namespace header files

If your namespace is going to be used in more than one program, you want to have it in a separate source file, with an accompanying header file:

There is a `vector` in the standard namespace and in the new `geometry` namespace:

```
#include <vector>
#include "geolib.h"
using namespace geometry;
int main() {
  std::vector< vector > vectors;
  vectors.push_back( vector( point(1,1),point(4,5) ) );
```

The header would contain the normal function and class headers, but now inside a named namespace:

```
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point() {};
    point( double x,double y );
    double x();
    double y();
  };
  class vector {
  private:
    point from,to;
  public:
    vector( point from,point to);
    double size();
  };
}
```

and the implementation file would have the implementations, in a namespace of the same name:

```
namespace geometry {
  point::point( double x,double y ) {
      xcoord = x; ycoord = y; };
  double point::x() { return xcoord; }; // 'accessor'
  double point::y() { return ycoord; };
  vector::vector( point from,point to) {
    this->from = from; this->to = to;
  };
  double vector::size() {
    double
      dx = to.x()-from.x(), dy = to.y()-from.y();
    return sqrt( dx*dx + dy*dy );
  };
}
```

## 20.2 Namespaces and libraries

As the introduction to this chapter argued, namespaces are a good way to prevent name conflicts. This means that it's a good idea to create a namespace for all your routines. You see this design in almost all published C++ libraries.

Now consider this scenario:

1. You write a program that uses a certain library;
2. A new version of the library is released and installed on your system;
3. Your program, using shared/dynamic libraries, starts using this library, maybe even without you realizing it.

This means that the old and new libraries need to be compatible in several ways:

1. All the classes, functions, and data structures defined in the earlier version also need to be defined in the new. This is not a big problem: new library versions typically add functionality. However,
2. The data layout of the new version needs to be the same.

That second point is subtle. To illustrate, consider the library is upgraded:

First version:

```
namespace geometry {
  class vector {
    private:
    std::vector<float> data;
    std::string name;
  }
}
```

New version:

```
namespace geometry {
  class vector {
    private:
    std::vector<float> data;
    int id;
    std::string name;
  }
}
```

The problem is that your compiled program has explicit information where the class members are located in the class object. By changing that structure of the objects, those references are no longer correct. This is called 'Application Binary Interface (ABI) breakage' and it leads to *undefined behavior*.

The library can prevent this by:

```
namespace geometry {
  inline namespace v1.0 {
    class vector {
      private:
      std::vector<float> data;
      std::string name;
    }
  }
}
```

and updating the version number in future version. The program using this library implicitly uses the namespace `geometry::v1.0::vector` so after a library update, trying to execute the program

## 20.3 Best practices

In this course we advocated pulling in functions explicitly:

```
#include <iostream>
using std::cout;
```

It is also possible to use

```
#include <iostream>
using namespace std;
```

The problem with this is that it may pull in unwanted functions. For instance:

This compiles, but should not:

```
#include <iostream>
using namespace std;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << '\n';
  return 0;
}
```

This gives an error:

```
#include <iostream>
using std::cout;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << '\n';
  return 0;
}
```

Even if you use `using namespace`, you only do this in a source file, not in a header file. Anyone using the header would have no idea what functions are suddenly defined.

# Chapter 21

# Preprocessor

In your source files you have seen lines starting with a hash sign, like

```
#include <iostream>
```

Such lines are interpreted by the *C preprocessor*.

We will see some of its more common uses here.

## 21.1 Include files

The `#include` pragma causes the named file to be included. That file is in fact an ordinary text file, stored somewhere in your system. As a result, your source file is transformed to another source file, in a source-to-source translation stage, and only that second file is actually compiled by the *compiler*.

Normally, this intermediate file with all included literally included is immediately destroyed again, but in rare cases you may want to dump it for debugging. See your compiler documentation for how to generate this file.

### 21.1.1 Kinds of includes

While you can include any kind of text file, normally you include a *header file* at the start of your source.

There are two kinds of includes

1. The file name can be included in angle brackets,

   ```
   #include <vector>
   ```

   which is typically used for system headers that are part of the compiler infrastructure;
2. the name can also be in double quotes,

   ```
   #include "somelib.h"
   ```

   which is typically used for files that are part of your code, or for libraries that you have downloaded and installed yourself.

### 21.1.2 Search paths

System headers can usually be found by the compiler because they are in some standard location. Including other headers may require additional action by you. If you

```
#include "foo.h"
```

the compiler only looks for that file in the current directory where you are compiling.

If the include file is part of a library that you have downloaded and installed yourself, say it is in

```
/home/yourname/mylibrary/include/foo.h
```

then you could of course

```
#include "/home/yourname/mylibrary/include/foo.h"
```

but that does not make your code very portable to other systems and other users. So how do you make

```
#include "foo.h"
```

be understood on any system?

The answer is that you can give your compiler one or more *include paths*. This is done with the `-I` flag.

```
icpc -c yourprogram.cxx -I/home/yourname/mylibrary/include
```

You can specify more than one such flag, and the compiler will try to find the `foo.h` file in all of them.

Are you now thinking that you have to type that path every time you compile? Now is the time to learn *makefiles* and the *Make* utility.

## 21.2 Textual substitution

Suppose your program has a number of arrays and loop bounds that are all identical. To make sure the same number is used, you can create a variable, and pass that to routines as necessary.

```
void dosomething(int n) {
  for (int i=0; i<n; i++) ....
}

int main() {
  int n=100000;

  double array[n];

  dosomething(n);
```

You can also use `#define` to define a *preprocessor macro*:

```
#define N 100000
void dosomething() {
  for (int i=0; i<N; i++) ....
}
```

```
int main() {
  double array[N];

  dosomething();
```

It is traditional to use all uppercase for such macros.

### 21.2.1 Dynamic definition of macros

Having numbers defined in the source takes away some flexibility. You can regain some of that flexibility by letting the macro be defined by the compiler, using the `-D` flag:

```
icpc -c yourprogram.cxx -DN=100000
```

Now what if you want a default value in your source, but optionally refine it with the compiler? You can solve this by testing for definition in the source with **`#ifndef`** 'if not defined':

```
#ifndef N
#define N 10000
#endif
```

### 21.2.2 A new use for 'using'

The **`using`** keyword that you saw in section 4.2.1 can also be used as a replacement for the **`#typedef`** pragma if it's used to introduce synonyms for types.

```
using Matrix = vector<vector<float>>;
```

### 21.2.3 Parameterized macros

Instead of simple text substitution, you can have *parameterized preprocessor macros*

```
#define CHECK_FOR_ERROR(i) if (i!=0) return i
...
ierr = some_function(a,b,c); CHECK_FOR_ERROR(ierr);
```

When you introduce parameters, it's a good idea to use lots of parentheses:

```
// the next definition is bad!
#define MULTIPLY(a,b) a*b
...
x = MULTIPLY(1+2,3+4);
```

Better

```
#define MULTIPLY(a,b) (a)*(b)
...
x = MULTIPLY(1+2,3+4);
```

Another popular use of macros is to simulate multi-dimensional indexing:

```
#define INDEX2D(i,j,n) (i)*(n)+j
...
double array[m,n];
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    array[ INDEX2D(i,j,n) ] = ...
```

**Exercise 21.1.** Write a macro that simulates 1-based indexing:

```
#define INDEX2D1BASED(i,j,n)  ????
...
double array[m,n];
for (int i=1; i<=m; i++)
  for (int j=n; j<=n; j++)
    array[ INDEX2D1BASED(i,j,n) ] = ...
```

## 21.3 Conditionals

There are a couple of *preprocessor conditions*.

### 21.3.1 Check on a value

The `#if` macro tests on nonzero. A common application is to temporarily remove code from compilation:

```
#if 0
  bunch of code that needs to
  be disabled
#endif
```

You can also test on numerical equality:

```
#if VARIANT == 1
  some code
#elif VARIANT == 2
  other code
#else
#error No such variant
#endif
```

### 21.3.2 Check for macros

The **`#ifdef`** test tests for a macro being defined. Conversely, **`#ifndef`** tests for a macro not being defined. For instance,

```
#ifndef N
#define N 100
#endif
```

Why would a macro already be defined? Well you can do that on the compile line:

```
icpc -c file.cc -DN=500
```

Another application for this test is in preventing recursive inclusion of header files; see section 19.6.

### 21.3.3 Including a file only once

It is easy to wind up including a file such as `iostream` more than once, if it is included in multiple other header files. This adds to your compilation time, or may lead to subtle problems. A header file may even circularly include itself. To prevent this, header files often have a structure

```
// this is foo.h
#ifndef FOO_H
#define FOO_H

// the things that you want to include

#endif
```

Now the file will effectively be included only once: the second time it is included its content is skipped.

Many compilers support the pragma `#once` that has the same effect:

```
// this is foo.h
#pragma once

// the things you want to include only once
```

However, this is not standardized, and the precise meaning is unclear (what if this is placed halfway a file) so the *Core Guidelines* recommend the explicit guards.

## 21.4 Other pragmas

- Packing data structure without padding bytes by `#pack`

```
#pragma pack(push, 1)
// data structures
#pragma pack(pop)
```

If you have too many `#ifdef` cases, you may get combinations that don't work. There is a convenient pragma to exit compilations that don't make sense: `#error`.

```
#ifdef __vax__
#error "Won't work on VAXen."
#endif
```

# Chapter 22

# Templates

You have seen in this course how objects of type `vector<string>` and `vector<float>` are very similar: have methods with the same names, and these methods behave largely the same. This angle-bracket notation is called 'templating' and the `string` or `float` is called the *template parameter*.

If you go digging into the source of the C++ library, you will find that, somewhere, there is just a single definition of the `vector` class, but with a new notation it gets the type `string` or `float` as template parameter.

To be precise, the templated class (or function) is preceded by a line

```
template< typename T >
class vector {
  ...
};
```

If you have multiple routines that do 'the same' for multiple types, you want the type name to be a variable. Syntax:

```
template <typename yourtypevariable>
// ... stuff with yourtypevariable ...
```

Historically `typename` was `class` but that's confusing.

## 22.1 Templated functions

We will start by taking a brief look at templated functions.

Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

**Exercise 22.1.** Machine precision, or 'machine epsilon', is sometimes defined as the smallest number $\epsilon$ so that $1 + \epsilon > 1$ in computer arithmetic.

Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

**Code:**

```
float float_eps;
epsilon(float_eps);
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << '\n';

double double_eps;
epsilon(double_eps);
cout << "Epsilon double: "
     << setw(10) << setprecision(4)
     << double_eps << '\n';
```

**Output**

```
[template] eps:

Epsilon float: 1.0000e-07
Epsilon double: 1.0000e-15
```

## 22.2 Templated classes

The most common use of templates is probably to define templated classes. You have in fact seen this mechanism in action:

The STL contains in effect

```
template<typename T>
class vector {
private:
  T *vectordata; // internal data
public:
  T at(int i) { return vectordata[i]  };
  int size() { /* return size of data */ };
  // much more
}
```

Let's consider a worked out example. We write a class `Store` that stored a single element of the type of the template parameter:

**Code:**

```
Store<int> i5(5);
cout << i5.value() << '\n';
```

**Output**

```
[template] example1i5:

5
```

The class definition looks pretty normal, except that the type (`int` in the above example) is parametrized:

```
template< typename T >
class Store {
private:
  T stored;
```

```
public:
  Store(T v) : stored(v) {};
  T value() { return stored;};
```

If we write methods that refer the templated type, things get a little more complicated. Let's say we want two methods `copy` and `negative` that return objects of the same templated type:

**Code:**

```
Store<float> also314 = f314.copy();
cout << also314.value() << '\n';
Store<float> min314 = f314.negative();
cout << min314.value() << '\n';
```

**Output**
**[template] example1f314:**

```
3.14
-3.14
```

The method definitions are fairly straightforward; if you leave out the template parameter, the *class name injection* mechanism uses the same template value as for the class being defined:

```
Store copy() { return Store(stored); };
Store<T> negative() { return Store<T>(-stored); };
```

### 22.2.1 Out-of-class method definitions

If we separate the class signature and the method definitions things get trickier. The class signature is easy:

```
template< typename T >
class Store {
private:
  T stored;
public:
  Store(T v);
  T value();
  Store copy();
  Store<T> negative();
};
```

The method definitions are more tricky. Now the template parameter needs to be specified every single time you mention the templated class, except for the name of the constructor:

```
template< typename T >
Store<T>::Store(T v) : stored(v) {};

template< typename T >
T Store<T>::value() { return stored;};

template< typename T >
Store<T>  Store<T>::copy() { return Store<T>(stored); };

template< typename T >
Store<T> Store<T>::negative() { return Store<T>(-stored); };
```

### 22.2.2 Specific implementations

Sometimes the template code works for a number of types (or values), but not for all. In that case you can specify the instantiation for specific types with empty angle brackets:

```
template <typename T>
void f(T);
template <>
void f(char c) { /* code with c */ };
template <>
void f(double d) { /* code with d */ };
```

### 22.2.3 Templates and separate compilation

The use of templates often makes *separate compilation* hard: in order to compile the templated definitions the compiler needs to know with what types they will be used. For this reason, many libraries are *header-only*: they have to be included in each file where they are used, rather than compiled separately and linked.

In the common case where you can foresee with which types a templated class will be instantiated, there is a way out. Suppose you have a templated class and function:

```
template <typename T>
class foo<T> {
};

template<typename T>
double f( T x );
```

and they will only be used (instantiated) with `float` and `double`, then adding the following lines after the class definition makes the file separately compilable:

```
template class foo<float>;
template class foo<double>;

template double f(float);
template double f(double);
```

If the class is split into a header and implementation file, these lines go in the implementation file.

## 22.3 Example: polynomials over fields

Any numerical application can be templated to allow for computation in *single precision* `float`s, and *double precision* `double`s. However, we can often also generalize the computation to other *fields*. Consider as an example polynomials, in both scalars and (square) matrices.

Let's start with a simple class for polynomials:

```
class polynomial {
private:
  vector<double> coefficients;
public:
  polynomial( vector<double> c )
    : coefficients(c) {};
  // 5 x^2 + 4 x + 3 = 5 x + 4 x + 3
  double eval( double x ) const {
    double y{0.};
    for_each(coefficients.rbegin(),coefficients.rend(),
```

```
            [x,&y] (double c) { y *= x; y += c; } );
    return y;
  };
  double operator()(double x) const { return eval(x); };
```

We store the polynomial coefficients, with the zeroth-degree coefficient in location zero, et cetera. The routine for evaluating a polynomial for a given input $x$ is an implementation of *Horner's scheme*:

$$5x^2 + 4x + 3 \equiv ((5) \cdot x + 4) \cdot x + 3$$

(Note that the `eval` method above uses `rbegin`, `rend` to traverse the coefficients in the correct order.)

For instance, the coefficients `2,0,1` correspond to the polynomial $2 + 0 \cdot x + 1 \cdot x^2$:

```
polynomial x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
  auto y = x2p2(x);
  cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

If we generalize the above to the case of matrices, all polynomial coefficients, as well as the $x$ input and $y$ output, are matrices.

The above code for evaluating a polynomial for a certain input works just as well for matrices, as long as the multiplication and addition operator are defined. So let's say we have a class `mat` and we have

```
mat::mat operator+( const mat& other ) const;
void mat::operator+=( const mat& other );
mat::mat operator*( const mat& other ) const;
void mat::operator*=( const mat& other );
```

Now we redefine the `polynomial` class, templated over the scalar type:

```
template< typename Scalar >
class polynomial {
private:
  vector<Scalar> coefficients;
public:
  polynomial( vector<Scalar> c )
    : coefficients(c) {};
  int degree() const { return coefficients.size()-1; };
  // 5 x^2 + 4 x + 3 = 5 x + 4 x + 3
  Scalar eval( Scalar x ) const {
    Scalar y{0.};
    for_each(coefficients.rbegin(),coefficients.rend(),
             [x,&y] (Scalar c) { y *= x; y += c; } );
    return y;
  };
```

The code using polynomials stays the same, except that we have to supply the scalar type as template parameter whenever we create a polynomial object. The above example of $p(x) = x^2 + 2$ becomes for scalars:

```
polynomial<double> x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
  auto y = x2p2(x);
  cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

and for matrices:

```
polynomial<mat> x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
  auto y = x2p2(x);
  cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

You see that after the templated definition the polynomial object is used entirely identically.

## 22.4 Concepts

The C++20 standard added the notion of *concept*.

Templates can be too generic. For instance, one could write a templated `gcd` function

```
template <typename T>
T gcd( T a, T b ) {
  if (b==0) return a;
  else return gcd(b,a%b);
}
```

which will work for various integral types. To prevent it being applied to non-integral types, you can specify a *concept* to the type:

```
template<typename T>
concept bool Integral() {
  return std::is_integral<T>::value;
}
```

used as:

```
template <typename T>
requires Integral<T>{}
T gcd( T a, T b ) { /* ... */ }
```

or

```
template <Integral T>
T gcd( T a, T b ) { /* ... */ }
```

Abbreviated function templates:

```
Integral auto gcd
    ( Integral auto a, Integral auto b )
  { /* ... */ }
```

# Chapter 23

# Error handling

## 23.1 General discussion

When you're programming, making errors is close to inevitable. *Syntax errors*, violations of the grammar of the language, will be caught by the compiler, and prevent generation of an executable. In this section we will therefore talk about *runtime errors*: behavior at runtime that is other than intended.

Here are some sources of runtime errors

**Array indexing** Using an index outside the array bounds may give a runtime error:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
  a.at(i) = x; // runtime error
```

or undefined behavior:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
  a[i] = x;
```

See further section 10.3.

**Null pointers** Using an uninitialized pointer is likely to crash your program:

```
Object *x;
if (false) x = new Object;
x->method();
```

**Numerical errors** such as division by zero will not crash your program, so catching them takes some care.

Guarding against errors.

- Check preconditions.
- Catch results.
- Check postconditions.

Error reporting:

- Message
- Total abort
- Exception

## 23.2 Mechanisms to support error handling and debugging

### 23.2.1 Assertions

One way catch errors before they happen, is to sprinkle your code with assertions: statements about things that have to be true. For instance, if a function should mathematically always return a positive result, it may be a good idea to check for that anyway. You do this by using the `assert` command, which takes a boolean, and will stop your program if the boolean is false:

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x,y) {
  assert( x<y );
  return /* some result */;
}
```

Check on valid results:

```
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```

There is also `static_assert`, which checks compile-time conditions only.

Since checking an assertion is a minor computation, you may want to disable it when you run your program in production by defining the `NDEBUG` macro:

```
#define NDEBUG
```

One way of doing that is passing it as a compiler flag:

```
icpc -DNDEBUG yourprog.cxx
```

As an example of using assertions, we can consider the iteration function of the Collatz exercise 6.13.

```
int collatz_next( int current ) {
  assert( current>0 );
  int next{-1};
  if (current%2==0) {
    next = current/2;
    assert(next<current);
  } else {
    next = 3*current+1;
    assert(next>current);
  }
  return next;
}
```

**Remark 14** *If an assertion fails, your program will call* `std::abort`*. This is a less elegant exit than* `std::exit`*.*

### 23.2.2 Exception handling

Assertions are a little crude: they terminate your program, and the only thing you can do is debug, rewrite, and rerun. Some errors are of a type that you could possibly recover from them. In that case, exception are a better idea, since these can be handled inside the program.

Code with problem:

```
if ( /* some problem */ )
  throw(5);
  /* or: throw("error"); */
```

```
try {
  /* code that can go wrong */
} catch (...) { // literally three
    dots!
  /* code to deal with the problem
    */
}
```

#### *23.2.2.1 Exception catching*

During the run of your program, an error condition may occur, such as accessing a vector elements outside the bounds of that vector, that will make your program stop. You may see text on your screen

```
terminating with uncaught exception
```

The operative word here is *exception*: an exceptional situation that caused the normal program flow to have been interrupted. We say that your program *throws* an *exception*.

**Code:**

```
vector<float> x(5);
x.at(5) = 3.14;
```

**Output**
**[except] boundthrow:**

```
libc++abi.dylib: terminating
    with uncaught exception
    of type
    std::out_of_range: vector
```

Now you know that there is an error in your program, but you don't know where it occurs. You can find out, but trying to *catch the exception*.

**Code:**

```
vector<float> x(5);
for (int i=0; i<10; i++) {
  try {
    x.at(i) = 3.14;
  } catch (...) {
    cout << "Exception indexing at: "
         << i << '\n';
    break;
  }
}
```

**Output**
**[except] catchbounds:**

```
Exception indexing at: 5
```

#### 23.2.2.2 Popular exceptions

- `std::out_of_range`: usually caused by using `at` with an invalid index.

#### 23.2.2.3 Throw your own exceptions

*Throwing* an *exception* is one way of signaling an error or unexpected behavior:

```
void do_something() {
  if ( oops )
    throw(5);
}
```

It now becomes possible to detect this unexpected behavior by *catching* the exception:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

If you're doing the prime numbers project, you can now do exercise 46.11.

You can throw integers to indicate an error code, a string with an actual error message. You could even make an error class:

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

You can use exception inheritance!

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std::string c ) {
  // handle string exception
```

```
}
```

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

**Exercise 23.1.** Define the function

$$f(x) = x^3 - 19x^2 + 79x + 100$$

and evaluate $\sqrt{f(i)}$ for the integers $i = 0 \ldots 20$.

- First write the program naively, and print out the root. Where is $f(i)$ negative? What does your program print?
- You see that floating point errors such as the root of a negative number do not make your program crash or something like that. Alter your program to throw an exception if $f(i)$ is negative, catch the exception, and print an error message.
- Alter your program to test the output of the `sqrt` call, rather than its input. Use the function *isnan*

  ```
  #include <cfenv>
  using std::isnan;
  ```

  and again throw an exception.

A *function try block* will catch exceptions, including in *member initializer* lists of constructors.

```
f::f( int i )
  try : fbase(i) {
    // constructor body
  }
  catch (...) { // handle exception
  }
```

- Functions can define what exceptions they throw:

  ```
  void func() throw( MyError, std::string );
  void funk() throw();
  ```

- Predefined exceptions: *bad_alloc*, *bad_exception*, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section 9.4.3.
- There is an implicit `try/except` block around your main. You can replace the handler for that. See the *exception* header file.
- Keyword *noexcept*:

  ```
  void f() noexcept { ... };
  ```

- There is no exception thrown when dereferencing a `nullptr`.

### 23.2.3 'Where does this error come from'

The CPP defines two macros, *__FILE__* and *__LINE__* that give you respectively the current file name and the current line number. You can use these to generate pretty error messages such as

```
Overflow occurred in line 25 of file numerics.cxx
```

The C++20 standard will offer `std::source_location` as a mechanism instead.

### 23.2.4 Legacy mechanisms

The traditional approach to error checking is for each routine to return an integer parameter that indicates success or absence thereof. Problems with this approach arise if it's used inconsistently, for instance by a user forgetting to heed the return codes of a library. Also, it requires that every level of the function calling hierarchy needs to check return codes.

The *PETSc* library uses this mechanism consistently throughout, and to great effect.

Exceptions are a better mechanism, since

- they can not be ignored, and
- they do not require handling on the levels of the calling hierarchy between where the exception is thrown and where it is caught.

And then there is the fact that memory management is automatic with exceptions.

### 23.2.5 Legacy C mechanisms

The *errno* variable and the *setjmp* / *longjmp* functions should not be used. These functions for instance do not the memory management advantages of exceptions.

## 23.3 Tools

Despite all your careful programming, your code may still compute the wrong result or crash with strange errors. There are two tools that may then be of assistance:

- *gdb* is the GNU interactive *debugger*. With it, you can run your code step-by-step, inspecting variables along way, and detecting various conditions. It also allows you to inspect variables after your code throws an error.
- *valgrind* is a memory testing tool. It can detect memory leaks (see section 16.3), as well as the use of uninitialized data.

# Chapter 24

# Standard Template Library

The C++ language has a *Standard Template Library* (STL), which contains functionality that is considered standard, but that is actually implemented in terms of already existing language mechanisms. The STL is enormous, so we just highlight a couple of parts.

You have already seen

- arrays (chapter 10),
- strings (chapter 11),
- streams (chapter 12).

Using a template class typically involves

```
#include <something>
using std::function;
```

see section 20.1.

## 24.1 Complex numbers

*Complex numbers* require the `complex` header. The `complex` type uses templating to set the precision.

```
#include <complex>
complex<float> f;
f.re = 1.; f.im = 2.;

complex<double> d(1.,3.);
```

Math operator like `+, *` are defined, as are math functions such as `exp`. Expressions involving a complex number and a simple scalar are well-defined if the scalar is of the underlying type of the complex number:

```
complex<float> x;
x + 1.f; // Yes
x + x.;  // No, because '1.' is double
```

Imaginary unit number $i$ through literals `i`, `if` (float), `il` (long):

```
using namespace std::complex_literals;
   std::complex<double> c = 1.0 + 1i;
```

Beware: `1+1i` does not compile.

**Code:**

```
vector< complex<double> > vec1(N, 1.+2.5i
    );
auto vec2( vec1 );
/* ... */
for ( int i=0; i<vec1.size(); i++ ) {
  vec2[i] = vec1[i] * ( 1.+1.i );
}
/* ... */
auto sum = accumulate
  ( vec2.begin(),vec2.end(),
    complex<double>(0.) );
cout << "result: " << sum << '\n';
```

**Output**

```
[complex] vec:
result: (-1.5e+06,3.5e+06)
```

Support:

```
std::complex<T> conj( const std::complex<T>& z );
std::complex<T> exp( const std::complex<T>& z );
```

### 24.1.1 Complex support in C

The C language has had complex number support since C99 with the types

```
float _Complex
double _Complex
long double _Complex
```

The header `complex.h` gives synonyms

```
float complex
double complex
long double complex
```

for these.

See for instance https://en.cppreference.com/w/c/numeric/complex for details.

## 24.2 Containers

C++ has several types of *containers*. You have already seen `std::vector` (section 10.3) and `std::array` (section 10.4) and strings (chapter 11). Many containers have methods such as `push_back` and `insert` in common.

In this section we will look at a couple more containers.

### 24.2.1 Maps: associative arrays

Arrays use an integer-valued index. Sometimes you may wish to use an index that is not ordered, or for which the ordering is not relevant. A common example is looking up information by string, such as

finding the age of a person, given their name. This is sometimes called 'indexing by content', and the data structure that supports this is formally known as an *associative array*.

In C++ this is implemented through a `map`:

```
#include <map>
using std::map;
map<string,int> ages;
```

is set of pairs where the first item (which is used for indexing) is of type `string`, and the second item (which is found) is of type `int`.

A map is made by inserting the elements one-by-one:

```
#include <map>
using std::make_pair;
ages.insert(make_pair("Alice",29));
ages["Bob"] = 32;
```

You can range over a map:

```
for ( auto person : ages )
  cout << person.first << " has age " << person.second << endl;
```

A more elegant solution uses *structured binding* (section 24.4):

```
for ( auto [person,age] : ages )
  cout << person << " has age " << age << endl;
```

(It is possible to use const-references here.)

Searching for a key gives either the iterator of the key/value pair, or the `end` iterator if not found:

```
for ( auto k : {4,5} ) {
  auto wherek = intcount.find(k);
  if (wherek==intcount.end())
    cout << "could not find key" << k << '\n';
  else {
    auto [kk,vk] = *wherek;
    cout << "found key: " << kk << " has value " << vk << '\n';
  }
}
```

**Exercise 24.1.** If you're doing the prime number project, you can now do the exercises in section 46.6.2.

### 24.2.2 Sets

The `set` container is like a `map<SomeType,bool>`, that is, it only says'this element is present'. Like a mathematical set, in other words.

```
#include <set>
std::set<int> my_ints;
my_ints.insert(5);
my_ints.empty(); // predicate
my_ints.size(); // obvious
```

You can iterate over a set:

```
for ( auto x : my_ints )
  // do something
```

This gives you the elements in no particular order.

You can search through a set with `find`, which results in an iterator. If no match is found the `end` iterator is returned.

```
auto itr = my_ints.find(6);
if ( itr==my_ints.end() )
  cout << "not found\n";
```

You can search on elements that satisfy a predicate with `find_if`:

```
auto res = find_i(my_ints.begin(),my_ints.end(),
        [] ( auto e ) { return e>37; } );
```

Sets are useful in many computer algorithms. You often encounter idioms such as

```
while (not done) {
  for ( x in unprocessed ) {
    if (something with x) {
      processed.add(x);
      unprocessed.remove(x);
    }
  }
  }
```

See for instance HPC book [9], section 9.1.1.

## 24.3 Regular expression

The header `regex` gives C++ the functionality for *regular expression* matching. For instance, `regex_match` returns whether or not a string matches an expression exactly:

**Code:**

```
auto cap = regex("[A-Z][a-z]+");
for ( auto n :
      {"Victor", "aDam", "DoD"}
      ) {
  auto match =
    regex_match( n, cap );
  cout << n;
  if (match) cout << ": yes";
  else       cout << ": no" ;
  cout << '\n';
}
```

**Output**

```
[regexp] regexp:
Looks like a name:
Victor: yes
aDam: no
DoD: no
```

(Note that the regex matches substrings, but `regex_match` only returns true for a match on the whole string.

For finding substrings, use `regex_search`:

- the function itself evaluates to a `bool`;
- there is an optional return parameter of type *smatch* ('string match') with information about the match.

The *smatch* object has these methods:

- `smatch::position` states where the expression was matched,
- while `smatch::str` returns the string that was matched.
- `smatch::prefix` has the string preceding the match; with `smatch::prefix().size()` you get the number of characters preceding the match, that is, the location of the match.

**Code:**

```
{
  auto findthe = regex("the");
  auto found = regex_search
    ( sentence,findthe );
  assert( found==true );
  cout << "Found <<the>>" << '\n';
}
{
  smatch match;
  auto findthx = regex("o[^o]+o");
  auto found = regex_search
    ( sentence, match ,findthx );
  assert( found==true );
  cout << "Found <<o[^o]+o>>"
       << " at " << match.position(0)
       << " as <<" << match.str(0) <<
  ">>"
       << " preceeded by <<" <<
  match.prefix() << ">>"
       << '\n';
}
```

**Output**
**[regexp] search:**

```
Found <<the>>
Found <<o[^o]+o>> at 12 as
    <<own fo>> preceeded by
    <<The quick br>>
```

### 24.3.1 Regular expression syntax

C++ uses a variant of the International regular expression syntax. http://ecma-international.org/ecma-262/5.1/#sec-15.10. Consult that document for escape characters and more.

If your regular expression is getting too complicated with escape characters and such, consider using the *raw string literal* construct.

## 24.4 Tuples and structured binding

Remember how in section 7.5.2 we said that if you wanted to return more than one value, you could not do that through a return value, and had to use an *output parameter*? Well, using the STL there is a different solution.

You can make a *tuple* with `tuple`: an entity that comprises several components, possibly of different type, and which unlike a `struct` you do not need to define beforehand. (For tuples with exactly two elements, use `pair`.)

In C++11 you would use the *get* method to extract elements from a pair or tuple:

```
#include <tuple>

std::tuple<int,double,char> id = \
    std::make_tuple<int,double,char>( 3, 5.12, 'f' );
    // or:
    std::make_tuple( 3, 5.12, 'f' );
double result = std::get<1>(id);
std::get<0>(id) += 1;

// also:
std::pair<int,char> ic = make_pair( 24, 'd' );
```

Annoyance: all that 'get'ting.

This does not look terribly elegant. Fortunately, C++17 can use denotations and the **auto** keyword to make this considerably shorter. Consider the case of a function that returns a tuple. You could use **auto** to deduce the return type:

```
auto maybe_root1(float x) {
  if (x<0)
    return make_tuple
      <bool,float>(false,-1);
  else
    return make_tuple
      <bool,float>
        (true,sqrt(x));
};
```

but more interestingly, you can use a *tuple denotation*:

```
tuple<bool,float>
    maybe_root2(float x) {
  if (x<0)
    return {false,-1};
  else
    return {true,sqrt(x)};
};
```

The calling code is particularly elegant:

**Code:**

```
1 auto [succeed,y] = maybe_root2(x);
2 if (succeed)
3   cout << "Root of " << x
4        << " is " << y << '\n';
5 else
6   cout << "Sorry, " << x
7        << " is negative" << '\n';
```

**Output**
**[stl] tuple:**

```
Root of 2 is 1.41421
Sorry, -2 is negative
```

This is known as *structured binding*.

An interesting use of structured binding is iterating over a map (section 24.2.1):

```
for ( const auto &[key,value] : mymap ) ....
```

## 24.5 Union-like stuff: tuples, optionals, variants

There are cases where you need a value that is one type or another, for instance, a number if a computation succeeded, and an error indicator if not.

The simplest solution is to have a function that returns both a bool and a number:

```
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = sqrt(x);
  return true;
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
      cout << "Root is " << x << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

We will now consider some more idiomatically C++17 solutions to this.

### 24.5.1 Tuples

Using tuples (section 24.4) the solution to the above 'a number or an error' now becomes:

```
#include <tuple>
using std::tuple, std::pair;
  /* ... */
pair<bool,float> RootAndValid(float x) {
  if (x<0)
    return {false,x};
  else
    return {true,sqrt(x)};
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto [ok,root] = RootAndValid(x) ; ok )
      cout << "Root is " << root << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

### 24.5.2 Optional

The most elegant solution to 'a number or an error' is to have a single quantity that you can query whether it's valid. For this, the C++17 standard introduced the concept of a *nullable type*: a type that can somehow convey that it's empty.

Here we discuss `std::optional`.

```
#include <optional>
using std::optional;
```

#### 24.5.2.1 Creating an optional

You can create an optional quantity with a function that returns either a value of the indicated type, or `{}`, which is a synonym for `std::nullopt`.

```
#include <optional>
using std::optional;

optional<float> f {
  if (something) return 3.14;
  else return {};
}
```

#### 24.5.2.2 Optional value

You can test whether the optional quantity actually has a quantity with the method `has_value`, in which case you can extract the quantity with the method `value`:

```
auto maybe_x = f();
if (f.has_value())
// do something with f.value();
```

Trying to take the `value` for something that doesn't have one leads to a `bad_optional_access` exception:

**Code:**

```
optional<float> maybe_number = {};
try {
  cout << maybe_number.value() << '\n';
} catch (std::bad_optional_access) {
  cout << "failed to get value\n";
}
```

**Output**

```
[union] optional:
failed to get value
```

There is a function `value_or` that gives the value, or a default if the optional did not have a value.

**Exercise 24.2.** If you are doing the prime number project, you can now do exercise 46.14.

**Exercise 24.3.** The *eight queens* problem (chapter 49) can be elegantly solved using `std::optional`. See also section 49.3 for a Test-Driven Development (TDD) approach.

**Remark 15** *If you have an optional class object, you can assign that object with the* `emplace` *method:*

```
class WithInt {
public:
  WithInt( int i ) {};
  void foo() {};
};
```

```
/* ... */
optional<WithInt> withint;
{ withint.emplace(5); }
cout << withint.has_value() << '\n';
withint.foo();
```

### 24.5.3 Expected

The `std::optional` of the previous section is great for cases, such as the square root, where it is clear what it means if the value does not exist. However, if the non-existence comes from some sort of error, you may want to ask what the reason for non-existence is.

The C++23 addition of `std::expected` allows you to return a value, or provide more information on why that error is not there.

Expect double, return info string if not:

```
std::expected<double,string>
      square_root( double x ) {
  auto result = sqrt(x);
  if (x<0)
  return
    std::unexpected("negative");
  else if (x<limits<double>::min())
  return
    std::unexpected("underflow");
  else return result;
}
```

```
auto root = square_root(x);
if (x)
cout << "Root=" << root.value() <<
    '\n';
else if (root.error()==/* et cetera
    */ )
/* handle the problem */
```

### 24.5.4 Variant

In C, a `union` is an entity that can be one of a number of types. Just like that C arrays do not know their size, a `union` does not know what type it is. The C++ `variant` (C++17) does not suffer from these limitations. The function `get_if` can retrieve a value by type.

For a first example we consider the square root example.

```
#include <variant>
using std::variant, std::get_if;
  /* ... */
variant<bool,float>
    RootVariant(float x) {
  if (x<0)
    return false;
  else
    return sqrt(x);
};
```

```
for ( auto x : {2.f,-2.f} ) {
  auto okroot = RootVariant(x);
  auto root =
    get_if<float>(&okroot);
  if ( root )
    cout << "Root is " << *root <<
    '\n';
  auto nope = get_if<bool>(&okroot);
  if (nope)
    cout << "could not take root of
    " << x << '\n';
}
```

Showing some more possibilities with a variant of int, double, string:

```
variant<int,double,string> union_ids;
```

We can use the `index` function to see what variant is used (0,1,2 in this case) and `get` the value accordingly:

```
union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
  cout << "Double case: " << std::get<double>(union_ids) << '\n';
}
```

Getting the wrong variant leads to a `bad_variant_access` exception:

```
union_ids = 17;
cout << "Using option " << union_ids.index() << ": " << get<int>(union_ids) <<
    '\n';
try {
  cout << "Get as double: " << get<double>(union_ids) << '\n';
} catch ( bad_variant_access b ) {
  cout << "Exception getting as double while index=" << union_ids.index() <<
    '\n';
}
```

It is safer to use `get_if` which takes a pointer to a variant, and return a pointer if successful, and a null pointer if not:

```
union_ids = "Hello world";
if ( auto union_int = get_if<int>(&union_ids) ; union_int )
  cout << "Int: " << *union_int << '\n';
else if ( auto union_string = get_if<string>(&union_ids) ; union_string )
  cout << "String: " << *union_string << '\n';
```

Note that this needs the address of the variant, and returns something that you need to dereference.

**Exercise 24.4.** Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

The routine should return two roots, or one root, or an indication that the equation has no solutions.

**Code:**

```
for ( auto coefficients :
      { make_tuple(2.0, 1.5, 2.5),
        make_tuple(1.0, 4.0, 4.0),
        make_tuple(2.2, 5.1, 2.5)
      } ) {
  auto result =
    compute_roots(coefficients);
```

**Output**
**[union] quadratic:**

```
With a=2 b=1.5 c=2.5
No root
With a=2.2 b=5.1 c=2.5
Root1: -0.703978 root2: -1.6142
With a=1 b=4 c=4
Single root: -2
```

In this exercise you can return a boolean to indicate 'no roots', but a boolean can have two values, and only one has meaning. For such cases there is `std::monostate`.

#### *24.5.4.1 The same function on all variants*

Suppose you have a variant of some classes, which all support an identically prototyped method:

```
class x_type {
public: r_type method() { ... };
};
class y_type {
public: r_type method() { ... };
```

It is not directly possible to call this method on a variant:

```
variant< x_type,y_type> xy;
// WRONG xy.method();
```

For a specific example:

```
variant<double_object,string_object>
  union_is_double{ double_object(2.5) },
  union_is_string{ string{"two-point-five"} };
```

where we have methods `stringer` that gives a string representation, and `sizer` that gives the 'size' of the object.

The solution for this is `visit`, coming from the `variant` header. This is used to apply an object (defined below) to the variant object:

**Code:**

```
cout << "Size of <<"
     << visit(
    stringer{},union_is_double )
     << ">> is "
     << visit( sizer{},union_is_double )
     << '\n';
cout << "Size of <<"
     << visit(
    stringer{},union_is_string )
     << ">> is "
     << visit( sizer{},union_is_string )
     << '\n';
```

**Output**
**[union] visit:**

```
Size of <<2.5>> is 2
Size of <<two-point-five>>
    is 14
```

The mechanism to realize this is to have an object (here *stringer* and *sizer*) with an overloaded **operator()**. One implementation:

```
class sizer {
public:
  int operator()( double_object d ) {
    return static_cast<int>( d.value() ); };
  int operator()( string_object s ) {
    return s.value().size(); };
};
```

### 24.5.5 Any

While **variant** can be any of a number of pre specified types, **std**::*any* can contain really any type. Thus it is the equivalent of **void*** in C.

An *any* object can be cast with *any_cast*:

```
std::any a{12};
std::any_cast<int>(a); // succeeds
std::any_cast<string>(a); // fails
```

## 24.6 Random numbers

The STL has a *random number generator* that is more general and more flexible than the C version (section 24.6.4), discussed below.

- There are several generators that give uniformly distributed numbers;
- then there are distributions that translate this to non-uniform or discrete distributions.

First you declare an engine; later this will be transformed into a distribution:

```
std::default_random_engine generator;
```

This generator will start at the same value every time. You can seed it:

```
std::random_device r;
std::default_random_engine generator{ r() };
```

### 24.6.1 Distributions

The most common mode of generating random number is to pick them from a distribution. For instance, a uniform distribution between given bounds:

```
std::uniform_real_distribution<float> distribution(0.,1.);
```

A roll of the dice would result from:

```
std::uniform_int_distribution<int> distribution(1,6);
```

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// set the default random number generator
std::default_random_engine generator{r()};

// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

cout << "first rand: " << distribution(generator) << '\n';
```

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

Another distribution is the *Poisson distribution*:

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

**Exercise 24.5.** Chapter 65 has a case study of using random numbers for simulating a *random walk*.

### 24.6.2 Usage scenarios

Let's look at the following scenario where you need a whole bunch of random numbers, from a whole lot of places.

```
class Thing {
public:
  void do_something_random() {
    rnd = some_distribution( some_generator );
    f(rnd);
  };
};
int main() {
  vector<Thing> things(many);
  for ( auto& t : things )
    t.something_random();
};
```

You might be tempted not include not only the invocation of a Random Number Generator (RNG) in a routine that needs it, but also the definition. This is not going to work, because the generator will be initialized every time you call the function. You can fix this by making the generator variable `static`.

Wrong approach:

**Code:**

```
int nonrandom_int(int max) {
  std::default_random_engine engine;
  std::uniform_int_distribution<>
    ints(1,max);
  return ints(engine);
};
```

**Output**

```
[rand] nonrandom:
Three ints: 15, 15, 15.
```

Good approach:

**Code:**

```
int realrandom_int(int max) {
  static  std::default_random_engine
    static_engine;
  std::uniform_int_distribution<>
    ints(1,max);
  return ints(static_engine);
};
```

**Output**

```
[rand] truerandom:
Three ints: 15, 98, 70.
```

**Remark 16** *It would be Very! wrong to include the RNG itself in the function:*

```
class Thing {
private:
  random_device r;
  default_random_engine generator{ r() };
public:
  void do_something_random() {
    rnd = some_distribution( generator );
    f(rnd);
  };
};
```

*Some RNGs have a large amount of internal state, so this makes the `Thing` objects unnecessarily big.*

What if you have multiple routines that use random numbers? To keep it all statistically justified you could move the RNG into a separate routine:

```
int realrandom_int(int max) {
  static  std::default_random_engine static_engine;
  std::uniform_int_distribution<> ints(1,max);
  return ints(static_engine);
};
```

To clean up this design, you could then even put this in an object with online static class methods:

Note the use of **static**:

```
class generate {
private:
  static inline std::default_random_engine engine;
public:
  static int random_int(int max) {
    std::uniform_int_distribution<> ints(1,max);
    return ints(generate::engine);
  };
};
```

Usage:

```
auto nonzero_pcnt = generate::random_int(100)
```

### 24.6.3 Permutations

The function `shuffle` shuffles an array. Coupled with the `iota` function (from the `numeric` header) this easily gives a permutation:

**Code:**

```
1  std::vector<int> idxs(20);
2  iota(idxs.begin(),idxs.end(),0);
3 /* ... */
4  std::shuffle(idxs.begin(),
     idxs.end(), g);
```

**Output**
**[rand] shuffle:**

```
Iota:
  0   1   2   3   4   5   6
    7   8   9
 10  11  12  13  14  15  16
   17  18  19
Permute:
  6   9   4   3  16  15  18
    5   2  11
 14  19  17   1   0  13   7
   10  12   8
```

(where `g` is a random generator.)

### 24.6.4 C random function

There is an easy (but not terribly great) *random number generator* that works the same as in C.

```
#include <random>
using std::rand;
float random_fraction =
    (float)rand()/(float)RAND_MAX;
```

The function `rand` yields an **int** – a different one every time you call it – in the range from zero to `RAND_MAX`. Using scaling and casting you can then produce a fraction between zero and one with the above code.

This generator has some problems.

- The C random number generator has a period of $2^{15}$, which may be small.


Figure 24.1: Low number bias of a random number generator taken module

- There is only one generator algorithm, which is implementation-dependent, and has no guarantees on its quality.
- There are no mechanisms for transforming the sequence to a range. The common idiom

  ```
  int under100 = rand() % 100
  ```

  is biased to small numbers. Figure 24.1 shows this for a generator with period 7 taken modulo 3.

If you run your program twice, you will twice get the same sequence of random numbers. That is great for debugging your program but not if you were hoping to do some statistical analysis. Therefore you can set the *random number seed* from which the random sequence starts by the `srand` function. Example:

```
srand(time(NULL));
```

seeds the random number generator from the current time. This call should happen only once, typically somewhere high up in your main.

## 24.7 Time

Header

```
#include <chrono>
```

Convenient:

```
using namespace std::chrono
```

but here we spell it all out.

### 24.7.1 Time durations

You can define durations with `seconds`:

```
seconds s{3};
auto t = 4s;
```

**Remark 17** *Use one of the following:*

```
std::chrono::seconds two{2};

using std::chrono;
seconds two{2};
```

```
using namespace std::chrono;
seconds two{2};
```

You can do arithmetic and comparisons on this type:

**Code:**

```
cout << "This lasts "
     << s.count() << "s" << '\n';
cout << "This lasts ";
print_seconds( s+5s );
auto nine = 3.14*3s;
cout << nine.count()
     << "s is under 10 sec: "
     << boolalpha << (nine<10s)
     << '\n';
```

**Output**
**[chrono] basicsecond:**

```
This lasts 3s
This lasts 8s
9.42s is under 10 sec: true
```

Note that while `seconds` takes an integer argument, you can then multiply or divide it to get fractional values.

There is a duration `millisecond`, and you can convert seconds implicitly to milli, but the other way around you need `duration_count`:

```
print_milliseconds( 5s );
// DOES NOT COMPILE print_seconds( 6ms );
print_seconds( duration_cast<seconds>(6ms) );
```

The full list of durations (with suffixes) is: `hours` (`1h`), `minutes` (`1min`), `seconds` (`1s`), `milliseconds` (`1ms`), `microseconds` (`1us`), `nanoseconds` (`1ns`).

### 24.7.2 Time points

A *time point* can be considered as a duration from some starting point, such as the start of the Unix *epoch*: the start of the year 1970.

```
time_point<system_clock,seconds> tp{10'000s};
```

is `2h+46min+40s` into 1970.

You make this explicit by calling the `time_since_epoch` method on a time point, giving a duration.

### 24.7.3 Clocks

There are several clocks. The common supplied clocks are

- `system_clock` for time points that have a relation to the calendar; and
- `steady_clock` for precise measurements.

Usually, `high_resolution_clock` is a synonym for either of these.

A clock has properties:

- `duration`
- `rep`
- `period`
- `time_point`
- `is_steady`
- and a method `now()`.

As you saw above, a `time_point` is associated with a clock, and time points of different clocks can not be compared or converted to each other.

#### 24.7.3.1 Duration measurement

To time a segment of execution, use the `now` method of the clock, before and after the segment. Subtracting the time points gives a duration in nanoseconds, which you can cast to anything else:

**Code:**

```
using clock = system_clock;
clock::time_point  before =
    clock::now();
std::this_thread::sleep_for( 1.5s );
auto after = clock::now();
cout << "Slept for "
     <<
    duration_cast<milliseconds>(after-before).count()
     << "ms\n";
```

**Output**

```
[chrono] clock:
Slept for 1503ms
```

(The `sleep` function is not a `chrono` function, but comes from the `thread` header; see section 26.1.4.)

#### 24.7.3.2 Clock resolution

The *clock resolution* can be found from the `period` property:

```
auto
  num = myclock::period::num,
  den = myclock::period::den;
auto tick = static_cast<double>(num)/static_cast<double>(den);
```

Timing:

```
auto start_time = myclock::now();
auto duration = myclock::now()-start_time;
auto microsec_duration =
    std::chrono::duration_cast<std::chrono::microseconds>(duration);
cout << "This took " << microsec_duration.count() << "usec" << endl;
```

Computing new time points:

```
auto deadline = myclock.now() + std::chrono::seconds(10);
```

### 24.7.4 C mechanisms not to use anymore

Letting your process sleep: `sleep`

Time measurement: `getrusage`

## 24.8 File system

As of the C++17 standard,there is a file system header, `filesystem`, which includes things like a directory walker.

```
#include <filesystem>
```

> *TACC note:* The filesystem header seems to be included and working only in the *Intel OneAPI compiler*.

## 24.9 Regular expressions

**Code:**

```
auto cap = regex("[A-Z][a-z]+");
for ( auto n :
      {"Victor", "aDam", "DoD"}
      ) {
  auto match =
    regex_match( n, cap );
  cout << n;
  if (match) cout << ": yes";
  else       cout << ": no" ;
  cout << '\n';
}
```

**Output**

```
[regexp] regexp:
Looks like a name:
Victor: yes
aDam: no
DoD: no
```

## 24.10 Enum classes

The C-style `enum` keyword introduced global names, so

```
enum colors { red,yellow,green };
cout << red << "," << yellow << "," << green << '\n';
enum flag { red,white,blue }; // Collision!
```

does not work.

In C++ the `enum class` (or `enum struct`) was introduced, which makes the names into class members:

```
enum class colors { red,yellow,green };
cout << static_cast<int>( colors::red ) << ","
     << static_cast<int>( colors::yellow ) << ","
     << static_cast<int>( colors::green ) << '\n';
```

If such a class inherits from an integral type, this does not mean it behaves like an integer; for instance, you can not immediately ask if one is less than another. Instead, it only determines the amount of space taken for an enum item.

To let it behave like an integral type,you need to cast it:

```
enum class flag : unsigned int { red,white,blue };
// but we still need to cast them
cout << static_cast<int>( flag::red ) << ","
     << static_cast<int>( flag::white ) << ","
     << static_cast<int>( flag::blue ) << '\n';
```

If you only want a namespace-d enum:

```
class Field {
public:
  enum color { invalid=-1,white=0,red=1,blue=2,green=3 };
private:
  color mycolor;
public:
  void set_color( color c ) { mycolor = c; };
  /* ... */
  Field onefield;
  onefield.set_color( Field::color::blue );
```

# Chapter 25

# Fine points of scalar types

## 25.1 Integers

There are several integer types. First of all they can differ by how many bytes they take up; section 25.2.1. Next, there are signed and unsigned types;

## 25.2

25.2.3.

### 25.2.1 Integer precision

In addition to `int`, there are also `short` and `long` integers.

```
short int ishort = 1024;
short ishort2;
int normal = 2000111222;
long int much = 1234567890123;
long much2;
long long int whole_lot;
long long whole_lot2;
```

- A short int is at least 16 bits;
- An integer is at least 16 bits, which was the case in the old days of the *DEC PDP-11*, but nowadays they are commonly 32 bits;
- A long integer is at least 32 bits, but often 64;
- A 'long long' integer is at least 64 bits.
- If you need only one byte for your integer, you can use a `char`; see section 11.1.

There are a number of generally accepted *data models* for the definition of these types; see HPC book [9], section 3.7.1.

If you want to determine precisely what the range of integers or real numbers is that is stored in an `int` or `float` variable, you can use `limits`; see section 25.4.

If you want to dictate how many bits to use, there is the `cstdint` header, which defines such types as `int16_t`, `int32_t`, `int64_t`.

### 25.2.2 Integer overflow

From the limited space that an integer takes, it is clear that there have to be a smallest and largest integer. Querying such limits on integers is discussed in section 25.4.

Computations that exceed those limits have an undefined result; however, since C++20 integers are guaranteed to be stored as 'two's complement'; see HPC book [9], section 3.2.

### 25.2.3 Unsigned types

For the integer types `int`, `short`, `long` there are unsigned types

```
unsigned int i;
unsigned short s;
unsigned long l;
```

which contain nonnegative values. Consequently they have twice the range:

**Code:**

```
cout << "max int      : "
     << numeric_limits<int>::max() <<
    '\n';
cout << "max unsigned: "
     << numeric_limits<unsigned
    int>::max() << '\n';
```

**Output**
**[int] limit:**

```
max int      : 2147483647
max unsigned: 4294967295
```

(For the mechanism used here, see section 25.4.)

Unsigned values are fraught with danger. For instance, comparing them to integers gives counter-intuitive results:

**Code:**

```
unsigned int one{1};
int mone{-1};
cout << "less: " << boolalpha <<
    (mone<one) << '\n';
```

**Output**
**[int] cmp:**

```
less: false
```

For this reason, C++20 has introduced utility functions `cmp_equal` and such (in the `utility` header) that do these comparisons correctly.

## 25.3 Floating point types

Truncation and precision are tricky things. As a small illustration, let's do the same computation in single and double precision. While the results show the same with the default `cout` formatting, if we subtract them we see a non-zero difference.

**Code:**

```
double point3d = .3/.7;
float
  point3f = .3f/.7f,
  point3t = point3d;
cout << "double precision: "
     << point3d << '\n'
     << "single precision: "
     << point3f << '\n'
     << "difference with truncation:"
     << point3t - point3f
     << '\n';
```

**Output**
**[basic] point3:**

```
double precision: 0.428571
single precision: 0.428571
difference with
    truncation:-2.98023e-08
```

You can actually explain the size of this difference, however, we defer discussion of the details of floating point arithmetic to HPC book [9], chapter 3.

## 25.4 Limits

There used to be a header file `limits.h` that contained macros such as `MAX_INT` and `MIN_INT`. While this is still available, the STL offers a better solution in the `numeric_limits` function of the `numeric` header.

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;

cout << numeric_limits<long>::max();
```

- The largest number is given by `max`; use `lowest` for 'most negative'.
- The smallest denormal number is given by `denorm_min`.
- `min` is the smallest positive number that is not a denormal;
- There is an `epsilon` function for machine precision:

**Code:**

```
cout << "Single lowest "
     << numeric_limits<float>::lowest()
     << " and epsilon "
     << numeric_limits<float>::epsilon()
     << '\n';
cout << "Double lowest "
     << numeric_limits<double>::lowest()
     << " and epsilon "
     << numeric_limits<double>::epsilon()
     << '\n';
```

**Output**
**[stl] eps:**

```
Single lowest -3.40282e+38 and
    epsilon 1.19209e-07
Double lowest -1.79769e+308 and
    epsilon 2.22045e-16
```

**Code:**

```
cout << "Signed int: "
     << numeric_limits<int>::min() << " "
     << numeric_limits<int>::max()
     << '\n';
cout << "Unsigned    "
     << numeric_limits<unsigned int>::min() <<
    " "
     << numeric_limits<unsigned int>::max()
     << '\n';
cout << "Single      "
     << numeric_limits<float>::denorm_min() <<
    " "
     << numeric_limits<float>::min() << " "
     << numeric_limits<float>::max()
     << '\n';
cout << "Double      "
     << numeric_limits<double>::denorm_min()
    << " "
     << numeric_limits<double>::min() << " "
     << numeric_limits<double>::max()
     << '\n';
```

**Output**
**[stl] limits:**

```
Signed int: -2147483648
    2147483647
Unsigned    0 4294967295
Single      1.4013e-45
    1.17549e-38 3.40282e+38
Double      4.94066e-324
    2.22507e-308 1.79769e+308
```

**Exercise 25.1.** Write a program to discover what the maximal $n$ is so that $n!$, that is, $n$-factorial, can be represented in an `int`, `long`, or `long long`. Can you write this as a templated function?

Operations such as dividing by zero lead to floating point numbers that do not have a valid value. For efficiency of computation, the processor will compute with these as if they are any other floating point number.

### 25.4.1 Not-a-number

The *IEEE 754* standard for floating point numbers states that certain bit patterns correspond to the value *NaN*: 'not a number'. This is the result of such computations as the square root of a negative number, or zero divided by zero; you can also explicitly generate it with *quiet_NaN* or *signalling_NaN*.

- `NaN` is only defined for floating point types: the test *has_quiet_NaN* is false for other types such as **bool** or **int**.
- Even through *complex* is built on top of floating point types, there is no `NaN` for it.

**Code:**

```
cout << "Double NaNs: "
     <<
   std::numeric_limits<double>::quiet_NaN(
      << ' ' // nan
     <<
   std::numeric_limits<double>::signaling_NaN()
   << ' ' // nan
     << '\n'
     << "zero divided by zero: "
     << 0 / 0.0          << '\n';
cout << boolalpha
     << "Int has NaN: "
     <<
   std::numeric_limits<int>::has_quiet_NaN
     << '\n';
```

**Output**
**[limits] nan:**

```
Double NaNs: nan nan
zero divided by zero: nan
Int has NaN: false
```

### 25.4.2 Tests

There are tests for detecting whether a number is `Inf` or `NaN`. However, using these may slow a computation down.

The functions `isinf` and `isnan` are defined for the floating point types (`float`, `double`, `long double`), returning a `bool`.

```
#include <math.h>
isnan(-1.0/0.0);   // false
isnan(sqrt(-1.0)); // true
isinf(-1.0/0.0);   // true
isinf(sqrt(-1.0)); // false
```

## 25.5 Common numbers

```
#include <numbers>
static constexpr float pi = std::numbers::pi;
```

# Chapter 26

# Concurrency

Concurrency is a difficult subject. For a good introduction watch https://www.youtube.com/watch?v=F6Ipn7gCOsY from which a lot of this is taken.

## 26.1 Thread creation

Use header

```
#include <thread>
```

A thread is an object of class `std::thread`, and creating it you immediately begin its execution. The thread constructor takes at least one argument, a *callable* (a function pointer or a lambda), and optionally the arguments of that function-like object.

The environment that calls the thread needs to call `join` to ensure the completion of the thread.

**Code:**

```
auto start_time = Clock::now();
auto waiting_thread =
  std::thread( [] () {
    sleep(1);
  }
    );
waiting_thread.join();
auto duration = Clock::now()-start_time;
```

**Output**
**[thread] block:**

```
This took 1.00136 sec
```

An example with a function that takes arguments:

```
#include <thread>

auto somefunc = [] (int i,int j) { /* stuff */ };
std::thread mythread( somefunc, arg1,arg2 );
mythread.join()
```

### 26.1.1 Multiple threads

Creating a single thread is not very useful. Often you will create multiple threads that will subdivide work, and then wait until they all finish.

```
vector<thread> mythreads;
for ( i=/* stuff */ )
  mythreads.push_back( thread( somefunc,someargs ) );
for ( i=/* stuff */ )
  mythreads[i].join();
```

Here is a simple hello world example. Because there is no guarantee on the ordering of when threads start or end, the output can look messy:

**Code:**

```
vector< std::thread > threads;
for ( int i=0; i<NTHREADS-1; i++ ) {
  threads.push_back
    ( std::thread(hello_n,i) );
}
threads.emplace_back
  ( hello_n,NTHREADS-1 );
for ( auto& t : threads )
  t.join();
```

**Output**
**[thread] hellomess:**

```
Hello Hello 01

Hello 2
Hello 3
Hello 4
```

(Note the call to `emplace_back`: because of *perfect forwarding* it can invoke the constructor on the `thread` arguments.)

We bring order in this message by including a wait. Note that the thread now executes a function given by a lambda expression:

**Code:**

```
  threads.push_back
    ( std::thread
      ( /* function: */
       [] (int i ) {
         std::chrono::seconds
wait(i);

std::this_thread::sleep_for(wait);
          hello_n(i); },
       /* argument: */ i
       )
     );
```

**Output**
**[thread] hellonice:**

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
```

Of course, in practice you don't synchronize threads through waits. Read on.

### 26.1.2 Asynchronous tasks

One problem with threads is how to return data. You could solve that with capturing a variable by reference, but that is not very elegant. A better solution would be if you could ask a thread 'what is the thing you just calculated'.

For this we have the `std::future`, templated with the return type. Thus, a

```
std::future<int>
```

will be an int, somewhere in the future. You retrieve the value with `get`:

```
std::future<string> fut_str = std::async
  ( [] () -> string { return "Hello world"; } );
auto result_str = fut_str.get();
cout << result_str << '\n';
```

An example with multiple futures:

```
    vector< std::future<string> > futures;
    for ( int ithread=0; ithread<NTHREADS; ithread++ ) {
      futures.push_back
        ( std::async
          ( [ithread] () ->string {
            stringstream ss;
            ss << "Hello world " << ithread;
            return ss.str();
          } ) );
    }
    for ( int ithread=0; ithread<NTHREADS; ithread++ ) {
      cout << futures.at(ithread).get() << '\n';
    }
```

One problem with `async` is that the task need not execute on a new thread: the runtime can decide to execute it on the calling thread, only when the `get` call is made. To force a new thread to be spawned immediately, use

```
auto fut = std::async
    ( std::launch:async, fn, arg1, arg2 );
```

Lazy evaluation on the calling thread can be explicitly specified with `std::launch:async`.

### 26.1.3 Return results: futures and promises

Explicit use of promises and futures is on a lower level than `async`.

Requires header `future`.

```
auto promise = std::promise<std::string>();
auto producer = std::thread
  ( [&promise] { promise.set_value("Hello World"); } );

auto future = promise.get_future();

auto consumer = std::thread
  ( [&future] { std::cout << future.get() << '\n'; } );

producer.join(); consumer.join();
```

```
vector< std::thread >
    producers,consumers;
vector< std::promise<string> >
    promises;

for ( int i=0; i<4; i++ ) {

  promises.push_back(
    std::promise<string>() );
  producers.push_back
    ( std::thread
      ( [ i,&promises ] {
        stringstream ss;
        ss << "Hello world " << i <<
    ".";
    promises.at(i).set_value(ss.str());
      } ) );

  consumers.push_back
    ( std::thread
      ( [ i,&promises ] {
        std::cout <<
    promises.at(i).get_future().get()
    << '\n';
      } ) );
}

for ( auto& p : producers ) p.join();
for ( auto& c : consumers ) c.join();
```

### 26.1.4 The current thread

```
std::this_thread::get_id();
```

This is a unique ID, but not like an MPI rank.

There is also a `sleep_for` function for a thread.

### 26.1.5 More thread stuff

The C++20 `jthread` launches a thread which will join when its destructor is called. With the creation loop:

```
{
  vector<thread> mythreads;
  for ( i=/* stuff */ )
    mythreads.push_back( thread( somefunc,someargs ) );
}
```

the joins happen when the vector goes out of scope.

## 26.2 Data races

An important topic in concurrency is that of *data races*: the phenomenon that multiple accesses to a single data item are not temporally or causally ordered, for instance because the accesses are from threads that are simultaneously active.

```
std::mutex alock;
alock.lock();
/* critical section */
alock.unlock();
```

This has a bunch of problems, for instance if the critical section can throw an exception.

One solution is `std::lock_guard`:

```
std::mutex alock;
thread( [] () {
    std::lock_guard<std::mutex> myguard(alock);
    /* critical stuff */
    } );
```

The lock guard locks the mutex when it is created, and unlocks it when it goes out of scope.

For C++17, `std::scoped_lock` can do this with multiple mutexes.

Atomic variables exist:

```
std::atomic<int> shared_int;
shared_int++;
```

Communication between threads:

```
std::condition_variable somecondition;
// thread 1:
std::mutex alock;
std::unique_lock<std::mutex> ulock(alock);
somecondition.wait(ulock)
// thread 2:
somecondition.notify_one();
```

Similar but different:

```
#include <future>
std::future<int> future_computation =
  std::async( [] (int x) { return f(x); },
              100 );
future_computation.get();
```

```
std::future_status comp_status;
comp_status = future_computation.wait_for( /* chrono duration */ );
if (comp_status==std::future_status::ready)
  /* computation has finished */
```

## 26.3 Synchronization

Threads run concurrent with the environment (or thread) that created them. That means that there is no temporal ordering between actions prior to synchronization with `std::thread::join`.

In this example, the concurrent updates of `counter` form a *data race*:

**Code:**

```
auto start_time = myclock::now();
auto deadline = myclock::now() +
  std::chrono::seconds(1);
int counter{0};
auto add_thread =
  std::thread( [&counter,deadline]() {
    while (myclock::now()<deadline)
      printf("Thread:
  %d\n",++counter);
  }
    );
while (myclock::now()<deadline)
  printf("Main: %d\n",++counter);
add_thread.join();
cout << "Final value: " << counter <<
  '\n';
```

**Output**
**[thread] race:**

```
Three runs of <<race>>;
printing first lines only:
----
Main: 1
Thread: 51
Final value: 526851
Runtime: 1.00048 sec
----
Main: 1
Thread: 47
Final value: 617669
Runtime: 1.00243 sec
----
Main: 1
Thread: 47
Final value: 509073
Runtime: 1.00215 sec
```

Formally, this program has Undefined Behavior (UB), which you see reflected in the different final values.

The final value can be fixed by declaring the counter as `std::atomic`:

**Code:**

```
auto start_time = myclock::now();
auto deadline = myclock::now() +
  std::chrono::seconds(1);
std::atomic<int> counter{0};
auto add_thread =
  std::thread( [&counter,deadline]() {
    while (myclock::now()<deadline)
      printf("Thread:
  %d\n",++counter);
  }
    );
while (myclock::now()<deadline)
  printf("Main: %d\n",++counter);
add_thread.join();
cout << "Final value: " << counter <<
  '\n';
```

**Output**
**[thread] atomic:**

```
Three runs of <<atomic>>;
printing first lines only:
----
Main: 1
Thread: 54
Final value: 495120
Runtime: 1.00282 sec
----
Thread: 1
Main: 353
Final value: 474618
Runtime: 1.00312 sec
----
Main: 1
Thread: 59
Final value: 339453
Runtime: 1.00212 sec
```

Note that the accesses by the main and the thread are still not predictable, but that is a feature, not a bug, and definitely not UB

# Chapter 27

# Obscure stuff

## 27.1 Auto

This is not actually obscure, but it intersects many other topics, so we put it here for now.

### 27.1.1 Declarations

Sometimes the type of a variable is obvious:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
                ( 20, new myclass(1.3) );
```

(Pointer to vector of 20 shared pointers to `myclass`, initialized with unique instances.) You can write this as:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
            ( 20, new myclass(1.3) );
```

Return type can be deduced in C++17:

```
auto equal(int i,int j) {
  return i==j;
};
```

Return type of methods can be deduced in C++17:

```
class A {
private: float data;
public:
  A(float i) : data(i) {};
  auto &access() {
    return data; };
  void print() {
    cout << "data: " << data << '\n'; };
};
```

`auto` discards references and such:

**Code:**

```
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

**Output**
**[auto] plainget:**

```
data: 5.7
```

Combine `auto` and references:

**Code:**

```
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

**Output**
**[auto] refget:**

```
data: 6.7
```

For good measure:

```
A my_a(5.7);
const auto &get_data = my_a.access();
get_data += 1; // WRONG does not compile
my_a.print();
```

### 27.1.2 Auto and function definitions

The return type of a function can be given with a *trailing return type* definition:

```
auto f(int i) -> double { /* stuff */ };
```

This notations is more common for lambdas, chapter 13.

### 27.1.3 `decltype`: declared type

There are places where you want the compiler to deduce the type of a variable, but where this is not immediately possible. Suppose that in

```
auto v = some_object.get_stuff();
f(v);
```

you want to put a `try ... catch` block around just the creation of $v$. This does not work:

```
try { auto v = some_object.get_stuff();
} catch (...) {}
f(v);
```

because the `try` block is a scope. It also doesn't work to write

```
auto v;
try { v = some_object.get_stuff();
} catch (...) {}
f(v);
```

because there is no indication what type `v` is created with.

Instead, it is possible to query the type of the expression that creates `v` with `decltype`:

```
decltype(some_object.get_stuff()) v;
try { auto v = some_objects.get_stuff();
} catch (...) {}
f(v);
```

10.9.3

## 27.2 Casts

In C++, constants and variables have clear types. For cases where you want to force the type to be something else, there is the *cast* mechanism. With a cast you tell the compiler: treat this thing as such-and-such a type, no matter how it was defined.

In C, there was only one casting mechanism:

```
sometype x;
othertype y = (othertype)x;
```

This mechanism is still available as the **reinterpret_cast**, which does 'take this byte and pretend it is the following type':

```
sometype x;
auto y = reinterpret_cast<othertype>(x);
```

The inheritance mechanism necessitates another casting mechanism. An object from a derived class contains in it all the information of the base class. It is easy enough to take a pointer to the derived class, the bigger object, and cast it to a pointer to the base object. The other way is harder.

Consider:

```
class Base {};
class Derived : public Base {};
Base *dobject = new Derived;
```

Can we now cast dobject to a pointer-to-derived ?

- **static_cast** assumes that you know what you are doing, and it moves the pointer regardless.
- **dynamic_cast** checks whether `dobject` was actually of class `Derived` before it moves the pointer, and returns **nullptr** otherwise.

**Remark 18** *One further problem with the C-style casts is that their syntax is hard to spot, for instance by searching in an editor. Because C++ casts have a unique keyword, they are easier to recognize in a text editor.*

Further reading https://www.quora.com/How-do-you-explain-the-differences-among-static_cast-reinterpret_cast-const_cast-and-dynamic_cast-to-a-new-C++-programmer/answer/Brian-Bi

### 27.2.1 Static cast

One use of casting is to convert to constants to a 'larger' type. For instance, allocation does not use integers but *size_t*.

```
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
cout << "overflow: " << overflow << '\n';
size_t bignumber = static_cast<size_t>(hundredk)*hundredk;
cout << "bignumber: " << bignumber << '\n';
```

However, if the conversion is possible, the result may still not be 'correct'.

**Code:**

```
long int hundredg = 100000000000;
cout << "long number:      "
     << hundredg << '\n';
int overflow;
overflow = static_cast<int>(hundredg);
cout << "assigned to int: "
     << overflow << '\n';
```

**Output**

```
[cast] intlong:
```

There are no runtime tests on static casting.

Static casts are a good way of casting back void pointers to what they were originally.

### 27.2.2 Dynamic cast

Consider the case where we have a base class and derived classes.

```
class Base {
public:
  virtual void print() = 0;
};
class Derived : public Base {
public:
  virtual void print() {
    cout << "Construct derived!"
         << '\n'; };
};
class Erived : public Base {
public:
  virtual void print() {
    cout << "Construct erived!"
         << '\n'; };
};
```

Also suppose that we have a function that takes a pointer to the base class:

```
void f( Base *obj ) {
  Derived *der =
    dynamic_cast<Derived*>(obj);
  if (der==nullptr)
    cout << "Could not be cast to Derived"
```

```
          << '\n';
  else
    der->print();
};
```

The function can discover what derived class the base pointer refers to:

```
Base *object = new Derived();
f(object);
Base *nobject = new Erived();
f(nobject);
```

If we have a pointer to a derived object, stored in a pointer to a base class object, it's possible to turn it safely into a derived pointer again:

**Code:**

```
Base *object = new Derived();
f(object);
Base *nobject = new Erived();
f(nobject);
```

**Output**
**[cast] deriveright:**

```
make[1]: Nothing to be done
    for `deriveright'.
```

On the other hand, a `static_cast` would not do the job:

**Code:**

```
void g( Base *obj ) {
  Derived *der =
    static_cast<Derived*>(obj);
  der->print();
};
  /* ... */
    Base *object = new Derived();
    g(object);
    Base *nobject = new Erived();
    g(nobject);
```

**Output**
**[cast] derivewrong:**

```
make[1]: Nothing to be done
    for `derivewrong'.
```

Note: the base class needs to be polymorphic, meaning that that pure virtual method is needed. This is not the case with a static cast, but, as said, this does not work correctly in this case.

### 27.2.3 Const cast

With `const_cast` you can add or remove const from a variable. This is the only cast that can do this.

### 27.2.4 Reinterpret cast

The `reinterpret_cast` is the crudest cast, and it corresponds to the C mechanism of 'take this byte and pretend it of type whatever'. There is a legitimate use for this:

```
void *ptr;
ptr = malloc( how_much );
auto address = reinterpret_cast<long int>(ptr);
```

so that you can do arithmetic on the address. For this particular case, `intptr_t` is actually better.

### 27.2.5 A word about void pointers

A traditional use for casts in C was the treatment of *void pointer*s. The need for this is not as severe in C++ as it was before.

A typical use of void pointers appears in the PETSc [3, 4] library. Normally when you call a library routine, you have no further access to what happens inside that routine. However, PETSc has the functionality for you to specify a monitor so that you can print out internal quantities.

```
int KSPSetMonitor(KSP ksp,
  int (*monitor)(KSP,int,PetscReal,void*),
  void *context,
  // one parameter omitted
  );
```

Here you can declare your own monitor routine that will be called internally: the library makes a *callback* to your code. Since the library can not predict whether your monitor routine may need further information in order to function, there is the `context` argument, where you can pass a structure as void pointer.

This mechanism is no longer needed in C++ where you would use a *lambda* (chapter 13):

```
KSPSetMonitor( ksp,
  [mycontext] (KSP k,int ,PetscReal r) -> int {
    my_monitor_function(k,r,mycontext); } );
```

## 27.3 lvalue vs rvalue

The terms 'lvalue' and 'rvalue' sometimes appear in compiler error messages.

```
int foo() {return 2;}

int main()
{
    foo() = 2;

    return 0;
}

# gives:
test.c: In function 'main':
test.c:8:5: error: lvalue required as left operand of assignment
```

See the 'lvalue' and 'left operand'? To first order of approximation you're forgiven for thinking that an *lvalue* is something on the left side of an assignment. The name actually means 'locator value': something that's associated with a specific location in memory. Thus an lvalue is, also loosely, something that can be modified.

An *rvalue* is then something that appears on the right side of an assignment, but is really defined as everything that's not an lvalue. Typically, rvalues can not be modified.

The assignment `x=1` is legal because a variable `x` is at some specific location in memory, so it can be assigned to. On the other hand, `x+1=1` is not legal, since `x+1` is at best a temporary, therefore not at a specific memory location, and thus not an lvalue.

Less trivial examples:

```
int foo() { x = 1; return x; }
int main() {
  foo() = 2;
}
```

is not legal because `foo` does not return an lvalue. However,

```
class foo {
private:
  int x;
public:
  int &xfoo() { return x; };
};
int main() {
  foo x;
  x.xfoo() = 2;
```

is legal because the function `xfoo` returns a reference to the non-temporary variable `x` of the `foo` object.

Not every lvalue can be assigned to: in

```
const int a = 2;
```

the variable `a` is an lvalue, but can not appear on the left hand side of an assignment.

### 27.3.1 Conversion

Most lvalues can quickly be converted to rvalues:

```
int a = 1;
int b = a+1;
```

Here `a` first functions as lvalue, but becomes an rvalue in the second line.

The ampersand operator takes an lvalue and gives an rvalue:

```
int i;
int *a = &i;
&i = 5; // wrong
```

### 27.3.2 References

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```
std::string &s = std::string(); // wrong
```

is illegal. The type of `s` is an 'lvalue reference' and it can not be assigned from an rvalue.

On the other hand

```
const std::string &s = std::string();
```

works, since `s` can not be modified any further.

### 27.3.3 Rvalue references

A new feature of C++ is intended to minimize the amount of data copying through *move semantics*.

Consider a copy assignment operator

```
BigThing& operator=( const BigThing &other ) {
  BigThing tmp(other); // standard copy
  std::swap( /* tmp data into my data */ );
  return *this;
};
```

This calls a copy constructor and a destructor on `tmp`. (The use of a temporary makes this safe under exceptions. The `swap` method never throws an exception, so there is no danger of half-copied memory.)

However, if you assign

```
thing = BigThing(stuff);
```

Now a constructor and destructor is called for the temporary rvalue object on the right-hand side.

Using a syntax that is new in *C++*, we create an *rvalue reference*:

```
BigThing& operator=( BigThing &&other ) {
  swap( /* other into me */ );
  return *this;
}
```

## 27.4 Move semantics

With an *overloaded operator*, such as addition, on matrices (or any other big object):

```
Matrix operator+(Matrix &a,Matrix &b);
```

the actual addition will involve a copy:

```
Matrix c = a+b;
```

Use a move constructor:

```
class Matrix {
private:
  Representation rep;
public:
  Matrix(Matrix &&a) {
    rep = a.rep;
    a.rep = {};
  }
};
```

## 27.5 Graphics

C++ has no built-in graphics facilities, so you have to use external libraries such as *OpenFrameworks*, https://openframeworks.cc.

## 27.6 Standards timeline

Each standard has many changes over the previous.

If you want to detect what language standard you are compiling with, use the `__cpluscplus` macro:

**Code:**

```
cout << "C++ version: " << __cplusplus
    << '\n';
```

**Output**

```
[basic] version:
C++ version: 201703
```

This returns a `long int` with possible values `199711`, `201103`, `201402`, `201703`, `202002`.

Here are some of the highlights of the various standards.

### 27.6.1 C++98/C++03

Of the C++03 standard we only highlight deprecated features.

- `auto_ptr` was an early attempt at smart pointers. It is deprecated, and C++17 compilers will actually issue an error on it. For current smart pointers see chapter 16.

### 27.6.2 C++11

- `auto`

  ```
  const auto count = std::count
      (begin(vec),end(vec),value);
  ```

  The `count` variable now gets the type of whatever `vec` contained.
- Range-based for. We have been treating this as the base case, for instance in section 10.2. The C++11 mechanism, using an *iterator* (section 14.1.2) is largely obviated.
- Lambdas. See chapter 13.
- Chrono.
- Variadic templates.
- Smart pointers.

  ```
  unique_ptr<int> iptr( new int(5) );
  ```

  This fixes problems with `auto_ptr`.
- `constexpr`

  ```
  constexpr int get_value() {
    return 5*3;
  }
  ```

### 27.6.3 C++14

C++14 can be considered a bug fix on C++11. It simplifies a number of things and makes them more elegant.

- Auto return type deduction:

```
auto f() {
  SomeType something;
  return something;
}
```

- Generic lambdas (section 13.3.2)

```
const auto count = std::count(begin(vec),end(vec),
    [] ( const auto i ) { return i<3; }
);
```

  Also more sophisticated capture expressions.

- *constexpr*

```
constexpr int get_value() {
  int val = 5;
  int val2 = 3;
  return val*val2
}
```

### 27.6.4 C++17

- Optional; section 24.5.2.
- Structured binding declarations as an easier way of dissecting tuples; section 24.4.
- Init statement in conditionals; section 5.5.3.

### 27.6.5 C++20

- *modules*: these offer a better interface specification than using *header files*.
- *coroutines*, another form of parallelism.
- *concepts* including in the standard library via ranges; section 22.4.
- *spaceship operator* including in the standard library
- broad use of normal C++ for direct compile-time programming, without resorting to template meta programming (see last trip reports)
- *ranges*
- *calendars* and *time zones*
- *text formatting*
- **span**. See section 10.9.5.
- *numbers*. Section 25.5.
- Safe integer/unsigned comparison; section 25.2.3; integers are guaranteed two's complement.

Here is a summary with examples: https://oleksandrkvl.github.io/2021/04/02/cpp-20-overview.html.

# Chapter 28

# Graphics

The C++ language and standard library do not have graphics components. However, the following projects exist.

https://www.sfml-dev.org/

# Chapter 29

# C++ for C programmers

## 29.1 I/O

There is little employ for `printf` and `scanf`. Use **`cout`** (and `cerr`) and **`cin`** instead. There is also the *fmtlib* library.

Chapter 12.

## 29.2 Arrays

Arrays through square bracket notation are unsafe. They are basically a pointer, which means they carry no information beyond the memory location.

It is much better to use `vector`. Use range-based loops, even if you use bracket notation.

Chapter 10.

Vectors own their data exclusively, so having multiple C style pointers into the same data act like so many arrays does not work. For that, use the **`span`**; section 10.9.5.

### 29.2.1 Vectors from C arrays

Suppose you have to interface to a C code that uses `malloc`. Vectors have advantages, such as that they know their size, you may want to wrap these C style arrays in a vector object. This can be done using a *range constructor*:

```
vector<double> x( pointer_to_first, pointer_after_last );
```

Such vectors can still be used dynamically, but this may give a memory leak and other possibly unwanted behavior:

**Code:**

```
    float *x;
    x =
    (float*)malloc(length*sizeof(float));
  /* ... */
    vector<float> xvector(x,x+length);
    cout << "xvector has size: " <<
    xvector.size() << '\n';
    xvector.push_back(5);
    cout
      << "Push back was successful" <<
    '\n';
    cout << "pushed element: "
         << xvector.at(length) << '\n';
    cout << "original array: "
         << x[length] << '\n';
```

**Output**
**[array] cvector:**

```
xvector has size: 53
Push back was successful
pushed element: 5
original array: 0
```

## 29.3 Dynamic storage

Another advantage of vectors and other containers is the *RAII* mechanism, which implies that dynamic storage automatically gets deallocated when leaving a scope. Section 10.9.3. (For safe dynamic storage that transcends scope, see smart pointers discussed below.)

RAII stands for 'Resource Allocation Is Initialization'. This means that it is no longer possible to write

```
double *x;
if (something1) x = malloc(10);
if (something2) x[0];
```

which may give a memory error. Instead, declaration of the name and allocation of the storage are one indivisible action.

On the other hand:

```
vector<double> x(10);
```

declares the variable `x`, allocates the dynamic storage, and initializes it.

## 29.4 Strings

A *C string* is a character array with a *null terminator*. On the other hand, a `string` is an object with operations defined on it.

Chapter 11.

## 29.5 Pointers

Many of the uses for *C pointer*s, which are really addresses, have gone away.

- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than *`malloc`*; see above.
- Traversing arrays and vectors can be done with ranges; section 10.2.
- To pass an argument *by reference*, use a *reference*. Section 7.5.
- Anything that obeys a scope should be created through a *constructor*, rather than using *`malloc`*.

There are some legitimate needs for pointers, such as Objects on the heap. In that case, use **`shared_ptr`** or **`unique_ptr`**; section 16.2. The C pointers are now called *bare pointer*s, and they can still be used for 'non-owning' occurrences of pointers.

### 29.5.1 Parameter passing

No longer by address: now true references! Section 7.5.

## 29.6 Objects

Objects are structures with functions attached to them. Chapter 9.

## 29.7 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter 20.

## 29.8 Templates

If you find yourself writing the same function for a number of types, you'll love templates. Chapter 22.

## 29.9 Obscure stuff

### 29.9.1 Lambda

Function expressions. Chapter 13.

### 29.9.2 Const

Functions and arguments can be declared const. This helps the compiler. Section 18.1.

### 29.9.3 Lvalue and rvalue

Section 27.3.

# Chapter 30

# C++ review questions

## 30.1 Arithmetic

1. Given
   ```
   int n;
   ```
   write code that uses elementary mathematical operators to compute n-cubed: $n^3$.
   Do you get the correct result for all $n$? Explain.
2. What is the output of:
   ```
   int m=32, n=17;
   cout << n%m << endl;
   ```

## 30.2 Looping

1. Suppose a function
   ```
   bool f(int);
   ```
   is given, which is true for some positive input value. Write a main program that finds the smallest positive input value for which `f` is true.
2. Suppose a function
   ```
   bool f(int);
   ```
   is given, which is true for some negative input value. Write a main program that finds the (negative) input with smallest absolute value for which `f` is true.

## 30.3 Functions

**Exercise 30.1.** The following code snippet computes in a loop the recurrence

$$v_{i+1} = av_i + b, \qquad v_0 \text{ given.}$$

Write a recursive function

```
float v = value_n(n,a,b,v0);
```

that computes the value $v_n$ for $n \geq 0$.

## 30.4 Vectors

**Exercise 30.2.** The following program has several syntax and logical errors. The intended purpose is to read an integer $N$, and sort the integers $1, \ldots, N$ into two vectors, one for the odds and one for the evens. The odds should then be multiplied by two.

Your assignment is to debug this program. For 10 points of credit, find 10 errors and correct them. Extra errors found will count as bonus points. For logic errors, that is, places that are syntactically correct, but still 'do the wrong thing', indicate in a few words the problem with the program logic.

```
#include <iostream>
using std::cout; using std:cin;
using std::vector;

int main() {
  vector<int> evens,odd;
  cout << "Enter an integer value " << endl;
  cin << N;
  for (i=0; i<N; i++) {
    if (i%2=0) {
      odds.push_back(i);
    else
      evens.push_back(i);
  }
  for ( auto o : odds )
    o /= 2
  return 1
}
```

## 30.5 Vectors

**Exercise 30.3.** Take another look at exercise 30.1. Now assume that you want to save the values $v_i$ in an array `vector<float> values`. Write code that does that, using first the iterative, then the recursive computation. Which do you prefer?

## 30.6 Objects

**Exercise 30.4.** Let a class `Point` class be given. How would you design a class `SetOfPoints` (which models a set of points) so that you could write

```
Point p1,p2,p3;
SetOfPoints pointset;
// add points to the set:
pointset.add(p1); pointset.add(p2);
```

Give the relevant data members and methods of the class.

**Exercise 30.5.** You are programming a video game. There are moving elements, and you want to have an object for each. Moving elements need to have a method `move` with an argument that indicates a time duration, and this method updates the position of the element, using the speed of that object and the duration.

Supply the missing bits of code.

```
class position {
  /* ... */
public:
  position() {};
  position(int initial) { /* ... */ };
  void move(int distance) { /* ... */ };
};
class actor {
protected:
  int speed;
  position current;

public:
  actor() { current = position(0); };
  void move(int duration) {
    /* THIS IS THE EXERCISE: */
    /* write the body of this function */
  };
};
class human : public actor {
public:
  human() // EXERCISE: write the constructor
};
class airplane : public actor {
public:
  airplane() // EXERCISE: write the constructor
};

int main() {
  human Alice;
  airplane Seven47;
  Alice.move( 5 );
  Seven47.move( 5 );
```

# PART III

# FORTRAN

# Chapter 31

# Basics of Fortran

Fortran is an old programming language, dating back to the 1950s, and the first 'high level programming language' that was widely used. In a way, the fields of programming language design and compiler writing started with Fortran, rather than this language being based on established fields. Thus, the design of Fortran has some idiosyncrasies that later designed languages have not adopted. Many of these are now 'deprecated' or simply inadvisable. Fortunately, it is possible to write Fortran in a way that is every bit as modern and sophisticated as other current languages.

In this part of the book, you will learn safe practices for writing Fortran. Occasionally we will not mention practices that you will come across in old Fortran codes, but that we would not advise you taking up. While this exposition of Fortran can stand on its own, we will in places point out explicitly differences with C++.

## 31.1 Source format

Fortran started in the era when programs were stored on *punch card*s. Those had 80 columns, so a line of Fortran source code could not have more than 80 characters. Also, the first 6 characters had special meaning. This is referred to as *fixed format*. However, starting with *Fortran 90* it became possible to have *free format*, which allowed longer lines without special meaning for the initial columns.

There are further differences between the two formats (notably continuation lines) but we will only discuss free format in this course.

Many compilers have a convention for indicating the source format by the file name extension:

- `f` and `F` are the extensions for old-style fixed format; and
- `f90` and `F90` are the extensions for new free format.

Capital letters indicate that the *C preprocessor* is applied to the file. For this course we will use the `F90` extension.

## 31.2 Compiling Fortran

The minimal Fortran program is:

```
Program SomeProgram
  ! stuff goes here
End Program SomeProgram
```

You would compile this:

```
yourfortrancompiler -o myprogram myprogram.F90
```

and then execute with

```
./myprogram
```

For Fortran programs, the compiler is *gfortran* for the GNU compiler, and *ifort* for Intel.

**Exercise 31.1.** Add the line

```
print *,"Hello world!"
```

to the empty program, and compile and run it.

Fortran ignores *case* in both keywords and identifiers. Keywords such as **Program** in the above program can thus just as well be written as PrOgRaM.

A program optionally has a **stop** statement, which can return a message to the OS.

**Code:**

```
Program SomeProgram
  stop 'the code stops here'
End Program SomeProgram
```

**Output**
**[basicf] stop:**

```
STOP the code stops here
```

Additionally, a numeric code returned by **stop**

```
stop 1
```

can be queried with the $? shell parameter:

**Code:**

```
Program SomeProgram
  stop 17
End Program SomeProgram
```

**Output**
**[basicf] return:**

```
./stopreturn || code=$? \
         && echo Return code
    is $code
STOP 17
Return code is 17
```

## 31.3 Main program

Fortran does not use curly brackets to delineate blocks, instead you will find **end** statements. The very first one appears right when you start writing your program: a Fortran program needs to start with a **Program** line, and end with **End Program**. The program needs to have a name on both lines:

```
Program SomeProgram
  ! stuff goes here
End Program SomeProgram
```

and you can not use that name for any entities in the program.

**Remark 19** *The* emacs *editor will supply the block type and name if you supply the 'end' and hit the TAB or RETURN key; see section 2.1.1.*

### 31.3.1 Program structure

Unlike C++, Fortran can not mix variable declarations and executable statements, so both the main program and any subprograms have roughly a structure:

```
Program foo
  < declarations >
  < statements >
End Program foo
```

Another thing to note is that there are no include directives. Fortran does not have a 'standard library' such as C++ that needs to be explicitly included. Or you could say that the Fortran standard library is always by default included.

### 31.3.2 Statements

Let's say a word about layout. Fortran has a 'one line, one statement' principle, stemming from its *punch card* days.

- As long as a statement fits on one line, you don't have to terminate it explicitly with something like a semicolon:

  ```
  x = 1
  y = 2
  ```

- If you want to put two statements on one line, you have to terminate the first one:

  ```
  x = 1; y = 2
  ```

  But watch out for the line length: this is often limited to 132 characters.
- If a statement spans more than one line, all but the first line need to have an explicit *continuation character*, the ampersand:

  ```
  x = very &
    long &
    expression
  ```

### 31.3.3 Comments

Fortran knows only single-line *comments*, indicated by an exclamation point:

```
x = 1 ! set x to one
```

Everything from the exclamation point to the end of the line is ignored.

Maybe not entirely obvious: you can have a comment after a continuation character:

```
x = f(a) & ! term1
  + g(b)   ! term2
```

**Remark 20** *In Fortran77, 19 continuation lines were allowed. In Fortran95 this number was 40. As of the Fortran2003 standard, a line can be continued 256 times.*

## 31.4 Variables

Unlike in C++, where you can declare a variable right before you need it, Fortran wants its variables declared near the top of the program or subprogram:

```
Program YourProgram
  implicit none
  ! variable declaration
  ! executable code
End Program YourProgram
```

The `implicit none` should always be included; see section 31.4.1.1 for an explanation.

A variable declaration looks like:

```
type [ , attributes ] :: name1 [ , name2, .... ]
```

where

- we use the common grammar shorthand that `[ something ]` stands for an optional 'something';
- *type* is most commonly `integer`, `real(4)`, `real(8)`, `logical`. See below; section 31.4.1.
- the optional *attributes* are things such as `dimension`, `allocatable`, `intent`, `parameter` et cetera.
- *name* is something you come up with. This has to start with a letter. Unusually, variable names are case-insensitive. Thus,

  ```
  Integer :: MYVAR
  MyVar = 2
  print *,myvar
  ```

  is perfectly legal.

**Remark 21** *In Fortran66 there was a limit of six characters to the* length *of a* variable name, *though many compilers had extensions to this. As of the Fortran2003 standard, a variable name can be 63 characters long.*

The built-in data types of Fortran:

- Numeric: `Integer`, `Real`, `Complex`
- precision control:

  ```
  Integer :: i
  Integer(4) :: i4
  Integer(8) :: i8
  ```

  This usually corresponds to number of bytes; see textbook for full story.
- Logical: `Logical`.
- Character: `Character`. Strings are realized as arrays of characters.
- Derived types (like C++ structures or classes): `Type`

Some variables are not intended ever to change, such as if you introduce a variable `pi` with value 3.14159. You can mark this name as being a synonym for the value, rather than a variable you can assign to, with the `parameter` keyword.

```
real,parameter :: pi = 3.141592
```

In chapter 40 you will see that **parameter**s are often used for defining the size of an array.

Further specifications for numerical precision are discussed in section 31.4.1.2. Strings are discussed in chapter 36.

### 31.4.1 Declarations

#### *31.4.1.1 Implicit declarations*

Fortran has a somewhat unusual treatment of variable types: if you don't specify what data type a variable is, Fortran will deduce it from a simple rule, based on the first character of the name. This is a very dangerous practice, so we advocate putting a line

```
implicit none
```

immediately after any program or subprogram header. Now every variable needs to be given a type explicitly in a declaration.

#### *31.4.1.2 Variable 'kind's*

Fortran has several mechanisms for indicating the precision of a numerical type.

```
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
complex*32 :: c32
```

This often corresponds to the number of bytes used, **but not always**. It is technically a numerical *kind selector*, and it is nothing more than an identifier for a specific type.

### 31.4.2 Initialization

Variables can be initialized in their declaration:

```
integer :: i=2
real(4) :: x = 1.5
```

That this is done at compile time, leading to a common error:

```
subroutine foo()
  implicit none
  integer :: i=2
  print *,i
  i = 3
end subroutine foo
```

On the first subroutine call `i` is printed with its initialized value, but on the second call this initialization is not repeated, and the previous value of `3` is remembered.

## 31.5 Complex numbers

A complex number is a pair of real numbers. Complex constants can be written with a parenthesis notation, but to form a complex number from two real variables requires the **CMPLX** function. You can also use this function to force a real number to complex, so that subsequent computations are done in the complex realm.

Real and imaginary parts can be extracted with the function **real** and **aimag**.

Complex constants are written as a pair of reals in parentheses.
There are some basic operations.

**Code:**

```
Complex :: &
      fourtyfivedegrees = (1.,1.), &
      number,rotated
Real :: x,y
print *,"45 degrees:",fourtyfivedegrees
x = 3. ; y = 1.; number = cmplx(x,y)
rotated = number * fourtyfivedegrees
print '("Rotated number has Re=",f5.2,"
    Im=",f5.2)',&
      real(rotated),aimag(rotated)
```

**Output**
**[basicf] complex:**

```
45 degrees:
    (1.00000000,1.00000000)
Rotated number has Re= 2.00 Im=
    4.00
```

The imaginary root $i$ is not predefined. Use

```
Complex,parameter :: i = (0,1)
```

In Fortran2008, **Complex** is a derived type, and the real/imaginary parts can be extracted as

```
print *,rotated%re,rotated%im
```

**Code:**

```
print *,"45 degrees:",fourtyfivedegrees
x = 3. ; y = 1.; number = cmplx(x,y)
rotated = number * fourtyfivedegrees
print '("Rotated number has Re=",f5.2,"
    Im=",f5.2)',&
      rotated%re,rotated%im
```

**Output**
**[basicf] complexf08:**

```
45 degrees:
    (1.00000000,1.00000000)
Rotated number has Re= 2.00
    Im= 4.00
```

**Exercise 31.2.** Write a program to compute the complex roots of the quadratic equation

$$ax^2 + bx + c = 0$$

The variables `a,b,c` have to be real, but use the **cmplx** function to force the computation of the roots to happen in the complex domain.

## 31.6 Expressions

Fortran has arithmetic, logical, and string expressions.

- Arithmetic expressions look more or less the way you would expect them to. The only unusual operator is the power operator: `x**.5` is the square root of variable `x`.
- For boolean expressions there are constants `.true.` and `.false.`; operators are likewise enclosed in dots: `.and.` and such. Boolean variables are of type `Logical`. See section 31.19 for more.
- String handling will be discussed in chapter 36.

## 31.7 Bit operations

As of Fortran95 there are functions for bitwise operations:

- `btest(word,pos)` returns a `logical` if bit `pos` in `word` is set.
- `ibits(word,pos,len)` returns an integer (of the same kind as `word` with bits $p, \cdots p+\ell-1$ (extending leftward) right-adjusted.
- `ibset(i,pos)` takes an integer and returns the integer resulting from setting bit `pos` to 1. Likewise, `ibclr` clears that bit.
- `iand`, `ior`, `ieor` all operate on two integers, returning the bitwise and/or/xor result.
- `mvbits(from,frompos,len,to,tpos)` copies a range of bits between two integers.

## 31.8 Commandline arguments

Modern Fortran has functions for querying *commandline arguments*. First of all `command_argument_count` queries the number of arguments. This does not include the command itself, so this is one less than the C/C++ `argc` argument to main.

```
if (command_argument_count()==0) then
   print *,"This program needs an argument"
   stop 1
end if
```

The command can be retrieved with `get_command`.

The commandline arguments are retrieved with `get_command_argument`. These are strings as in C/C++, but you have to specify their length in advance:

```
character(len=10) :: size_string
integer :: size_num
```

Converting this string to an integer or so takes a little format trickery:

```
call get_command_argument(number=1,value=size_string)
read(size_string,'(i3)') size_num
```

(see section 42.3.)

## 31.9 Fortran type kinds

### 31.9.1 Kind selection

Kinds can be used to ask for a type with specified precision.

- For integers you can specify the number of decimal digits with **selected_int_kind**(*n*).
- For floating point numbers can specify the number of significant digits, and optionally the decimal exponent range with **selected_real_kind**(*p*[,*r*]). of significant digits.

Conversely, the properties of such types can be retrieved again with the functions **precision** (not for integers), **range**, *storage_size*.

Declaration of precision and/or range:

**Code:**

```
integer,parameter :: &
     i12 = selected_int_kind(12), &
     p6 = selected_real_kind(6), &
     p10r100 =
    selected_real_kind(10,100), &
     r400 = selected_real_kind(r=400), &
     p20 = selected_real_kind(20), &
     p40 = selected_real_kind(40)
integer(kind=i12) :: i
real(kind=p6) :: x
real(kind=p10r100) :: y
real(kind=r400) :: z
real(kind=p20) :: p
```

**Output**
**[basicf] kind:**

```
 Kinds:    8    4    8   16
   16   -1
Precision / range / bits:
 integer 12 :    0   18   64
 precision 6:    6   37   32
 p=10 r=100 :   15  307   64
 range=400  :   33 4931  128
 p=20       :   33 4931  128
```

Likewise, you can specify the precision of a constant. Writing `3.14` will usually be a single precision real.

Adding single/double precision constants, print as double:

**Code:**

```
real(8) :: x,y,z
x = 1.
y = .1
z = x+y
print *,z
x = 1.d0
y = .1d0
z = x+y
print *,z
```

**Output**
**[basicf] e0:**

```
   1.1000000014901161
   1.1000000000000001
```

You can query how many bytes a data type takes with **kind**.

Number of bytes determines numerical precision:

- Computations in 4-byte have relative error $\approx 10^{-6}$
- Computations in 8-byte have relative error $\approx 10^{-15}$

Also different exponent range: max $10^{\pm 50}$ and $10^{\pm 300}$ respectively.

F08: *storage_size* reports number of bits.

F95: **bit_size** works on integers only.

*c_sizeof* reports number of bytes, requires *iso_c_binding* module.

**Code:**

```
use iso_c_binding
implicit none
integer,parameter :: &
     p6 = selected_real_kind(6), &
     p12 = selected_real_kind(12)
real(kind=p6) :: x4
real(kind=p12) :: x8
10 format(i2" digits takes",i3," bytes")
print 10,6,c_sizeof(x4)
print 10,12,c_sizeof(x8)
```

**Output**
**[basicf] binding:**

```
 6 digits takes  4 bytes
12 digits takes  8 bytes
```

Force a constant to be **real**(8):

```
real(8) :: x,y
x = 3.14d0
y = 6.022e-23
```

- Use a compiler flag such as `-r8` to force all reals to be 8-byte.
- Write `3.14d0`
- `x = real(3.14, kind=8)`

### 31.9.2 Range

You can use the function **huge** to query the maximum value of a type.

**Code:**

```
Integer :: idef
Real :: rdef
Real(8) :: rdouble

print 10,"integer is kind",kind(idef)
print 10,"integer max is",huge(idef)
print 10,"real is kind",kind(rdef)
print 15,"real max is",huge(rdef)
print 10,"real8 is kind",kind(rdouble)
print 15,"real8 max is",huge(rdouble)
```

**Output**
**[typef] def:**

```
 integer is kind          4
  integer max is 2147483647
    real is kind           4
     real max is 0.3403E+39
   real8 is kind           8
    real8 max is 0.1798+309
```

With *ISO bindings* there is a more systematic approach.

Integers:

```
Integer(kind=Int8) :: i8
Integer(kind=Int16) :: i16
Integer(kind=Int32) :: i32
```

```
Integer(kind=Int64) :: i64
```

**Code:**

```
print 10,"Checking on supported types:"
print 10,"number of defined int
    types:",size(INTEGER_KINDS)
print 10,"these are the supported
    types:",INTEGER_KINDS
print 15,"Pre-defined types
    INT8,INT16,INT32,INT64:",&
     INT8,INT16,INT32,INT64
print *
print 20,"kind Int8 max is",huge(i8)
print 20,"kind Int16 max is",huge(i16)
print 20,"kind Int32 max is",huge(i32)
print 20,"kind Int64 max is",huge(i64)
```

**Output**

```
[typef] int:
  Checking on supported
    types:
  number of defined int
    types:  5
these are the supported
    types:  1   2   4   8
 16
Pre-defined types
    INT8,INT16,INT32,INT64

       1   2   4   8

    kind Int8 max is
             127
   kind Int16 max is
           32767
   kind Int32 max is
     2147483647
   kind Int64 max is
    9223372036854775807
```

Floating point numbers:

```
use iso_fortran_env
implicit none
real(kind=real32) :: x32
real(kind=real64) :: x64
print *,"32 bit max float:",huge(x32)
print *,"64 bit max float:",huge(x64)
```

## 31.10 Quick comparison Fortran vs C++

### 31.10.1 Statements

Some of it is much like C++:

- Assignments:

```
x = y
x = 2*y / (a+b)
z1 = 5; z2 = 6
```

  (Note the lack of semicolons at the end of statements.)
- I/O
- conditionals and loops

Different:

- function definition and calls

- array syntax
- object oriented programming
- modules

### 31.10.2 Input/Output, or I/O as we say

- Input:

```
READ *,n
```

- Output:

```
PRINT *,n
```

There is also `Write`.

The 'star' indicates that default formatting is used.
Other syntax for read/write with files and formats.

### 31.10.3 Expressions

- Pretty much as in C++
- Exception: `r**a` for power $r^a$.
- Modulus (the `%` operator in C++) is a function: `MOD(7,3)`.

- Long form:

```
.and. .not. .or.
.lt. .le. .eq. .ne. .ge. .gt.
.true. .false.
```

- Short form:

```
< <= == /= > >=
```

Conversion is done through functions.

- **INT**: truncation; **NINT** rounding
- **REAL**, *FLOAT*, *SNGL*, **DBLE**
- **CMPLX**, **CONJG**, *AIMIG*

http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-conv.htm

Complex numbers exist; section 31.5.

Strings are delimited by single or double quotes.

For more, see chapter 36.

## 31.11 Review questions

**Exercise 31.3.** What is the output for this fragment, assuming `i, j` are integers?

```
integer :: idiv
!! ...
i = 3 ; j = 2 ; idiv = i/j
print *,idiv
```

**Exercise 31.4.** What is the output for this fragment, assuming `i, j` are integers?

```
real    :: fdiv
!! ...
i = 3 ; j = 2 ; fdiv = i/j
print *,fdiv
```

**Exercise 31.5.** In declarations

```
real(4) :: x
real(8) :: y
```

what do the `4` and `8` stand for?

What is the practical implication of using the one or the other?

**Exercise 31.6.** Write a program that :

- displays the message `Type a number`,
- accepts an integer number from you (use `Read`),
- makes another variable that is three times that integer plus one,
- and then prints out the second variable.

**Exercise 31.7.** In the following code, if `value` is nonzero, what do expect about the output?

```
real(8) :: value8,should_be_value
real(4) :: value4
!! ...
print *,".. original value was:",value8
value4 = value8
print *,".. copied to single:",value4
should_be_value = value4
print *,".. copied back to double:",should_be_value
print *,"Difference:",value8-should_be_value
```

# Chapter 32

# Conditionals

## 32.1 Forms of the conditional statement

The Fortran conditional statement uses the `if` keyword:

Single line conditional:

```
if ( test ) statement
```

The full if-statement is:

```
if ( something ) then
  !! something_doing
else
  !! otherwise_else
end if
```

The 'else' part is optional; you can nest conditionals.

You can label conditionals, which is good for readability but adds no functionality:

```
checkx: if ( ... some test on x ... ) then
checky:   if ( ... some test on y ... ) then
                ... code ...
            end if checky
          else checkx
               ... code ...
          end if checkx
```

## 32.2 Operators

| Operator | old style | meaning | example |
|---|---|---|---|
| == | .eq. | equals | x==y-1 |
| /= | .ne. | not equals | x*x/=5 |
| > | .gt. | greater | y>x-1 |
| >= | .ge. | greater or equal | sqrt(y)>=7 |
| < | .lt. | less than | |
| <= | .le. | less or equal | |
| | .and. .or. | and, or | x<1 .and. x>0 |
| | .not. | not | .not.( x>1 .and. x<2 ) |
| | .eqv. | equiv (iff, not XOR) | |
| | .neqv. | not equiv (XOR) | |

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

**Exercise 32.1.** Read in three grades: Algebra, Biology, Chemistry, each on a scale $1 \cdots 10$. Compute the average grade, with the conditions:

- Algebra is always included.
- Biology is only included if it increases the average.
- Chemistry is only included if it is 6 or more.

## 32.3 Select statement

The Fortran equivalent of the C++ `case` statement is `select`. It takes single values or ranges; works for integers and character strings.

Test single values or ranges, integers or characters:

```
Select Case (i)
Case (:-1) ! range one and less
   print *,"Negative"
Case (5)
   print *,"Five!"
Case (0)
   print *,"Zero."
Case (1:4,6:) ! other cases, can not have (1:)
   print *,"Positive"
end Select
```

Compiler does checking on overlapping cases!

Case values need to be constant expressions.

The default case is covered with a `case default` case.

## 32.4 Boolean variables

The Fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`

**Exercise 32.2.** Print a boolean variable. What does the output look like in the true and false case?

## 32.5 Obsolete conditionals

Old versions of Fortran had other forms of the `if` statement, which you may still encounter in codes. The `if, arithmetic` was declared obsolescent in Fortran90 and was deleted in Fortran2018.

## 32.6 Review questions

**Exercise 32.3.** What is a conceptual difference between the C++ `switch` and the Fortran `Select` statement?

# Chapter 33

# Loop constructs

## 33.1 Loop types

Fortran has the usual indexed and 'while' loops. There are variants of the basic loop, and both use the `do` keyword. The simplest loop has a loop variable, an upper bound, and a lower bound.

```
integer :: i

do i=1,10
  ! code with i
end do
```

You can include a step size (which can be negative) as a third parameter:

By steps of 3:

```
do i=1,10,3
  ! code with i
end do
```

Counting down:

```
do i=10,1,-1
  ! code with i
end do
```

The loop variable is defined outside the loop, so it will have a value after the loop terminates.

- Fortran loops determine the iteration count before execution; a loop will run that many iterations, unless you `Exit`.
- You are not allowed to alter the iteration variable.
- Non-integer loop variables used to be allowed, no longer.

The while loop has a pre-test:

```
do while (i<1000)
  print *,i
  i = i*2
end do
```

## 33.2 Interruptions of the control flow

For indeterminate looping, you can use the **while** test, or leave out the loop parameter altogether. In that case you need the **exit** statement to stop the iteration.

Loop without counter or while test:

```
do
  call random_number(x)
  if (x>.9) exit
  print *,"Nine out of ten exes agree"
end do
```

Compare to `break` in C++.

Skip rest of current iteration:

```
do i=1,100
  if (isprime(i)) cycle
  ! do something with non-prime
end do
```

Compare to `continue` in C++.

You can label loops
useful with **exit** statement:

```
outer: do i=1,10
   inner: do j=1,10
      test: if (i*j>42) then
         print *,i,j
         exit outer
      end if test
   end do inner
end do outer
```

The label needs to be on the same line as the **do**, and if you use a label, you need to mention it on the **end do** line.

Cycle and exit can apply to multiple levels, if the do-statements are labeled.

```
outer : do i = 1,10
inner : do j = 1,10
    if (i+j>15) exit outer
    if (i==j) cycle inner
end do inner
end do outer
```

## 33.3 Implied do-loops

There are do loops that you can write in a single line by an expression and a loop header. In effect, such an *implied do loop* becomes the sum of the indexed expressions. This is useful for I/O. For instance, iterate a simple expression:

If you loop over a print statement, each print statement is on a new line;
use an implied loop to print on one line.

```
Print *,(2*i,i=1,20)
```

You can iterate multiple expressions:

```
Print *,(2*i,2*i+1,i=1,20)
```

These loops can be nested:

```
Print *,( (i*j,i=1,20), j=1,20 )
```

Also useful for **Read**.

This construct is especially useful for printing arrays.

**Exercise 33.1.** Use the implied do-loop mechanism to print a triangle:

```
1
2 2
3 3 3
4 4 4 4
```

up to a number that is input.

## 33.4 Obsolete loop statements

Old versions of Fortran had other forms of the $\circ$ statement, which you may still encounter in codes. As of Fortran2018, **do** loops have to end in **end do** or **continue**. Shared termination is likewise a deleted feature.

Fortran has a **goto** statement. While this was needed in the 1950 and 60s, nowadays it is considered bad programming practice. Most of its traditional uses can be covered with the **cycle** and **exit** statements. The **continue** statement, usually used as the target of a **goto**, is similarly rarely used anymore.

## 33.5 Review questions

**Exercise 33.2.** What is the output of:

```
do i=1,11,3
  print *,i
end do
```

What is the output of:

```
do i=1,3,11
  print *,i
end do
```

# Chapter 34

# Procedures

Programs can have subprograms: parts of code that for some reason you want to separate from the main program. The term for these is *procedure*. While this is actually a keyword, you will not see it until section 38.5; in this chapter we consider only **subroutine** and **function**.

If you structure your code in a single file, this is the recommended structure:

Simplest way of defining procedures:
in **Contains** part of main program.

```
Program foo
  < declarations>
  < executable statements >
  Contains
    < procedure definitions >
End Program foo
```

Two types of procedures: functions and subroutines. More later.

That is, procedures are placed after the main program statements, separated by a **Contains** clause.

In general, these are the placements of procedures:

- Internal: after the **Contains** clause of a program:

  ```
  Program foo
    ... stuff ...
  Contains
    Subroutine bar()
    End Subroutine bar
  End Program foo
  ```

  This is the mode that we focus on in this chapter.
- In a **Module**; see section 38.2.
- Externally: the procedure is not internal to a **Program** or **Module**. This can happen in the case of 3rd party libraries, or code linked in from another language. In this case it's safest to declare the procedure through an **Interface** specification; section 43.1.

## 34.1 Subroutines and functions

Fortran has two types of procedures:

- Subroutines, which are somewhat like `void` functions in C++: they can be used to structure the code, and they can only return information to the calling environment through their parameters.
- Functions, which are like C++ functions with a return value.

Both types have the same structure, which is roughly the same as of the main program. For subroutines:

```
subroutine foo( <parameters> )
<variable declarations>
<executable statements>
end subroutine foo
```

and for functions:

```
returntype function foo( <parameters> )
<variable declarations>
<executable statements>
end function foo
```

There is another syntax for declaring functions, see section 34.2.1.

Exit from a procedure can happen two ways:

1. the flow of control reaches the end of the procedure body:

   ```
   subroutine foo()
     statement1
     ..
     statementn
   end subroutine foo
   ```

   or
2. execution is finished by an explicit **`return`** statement.

   ```
   subroutine foo()
     print *,"foo"
     if (something) return
     print *,"bar"
   end subroutine foo
   ```

The **`return`** statement is optional in the first case. The **`return`** statement is different from C++ in that it does not indicate a returned result value of a function.

**Exercise 34.1.** Rewrite the above subroutine `foo` without a **`return`** statement.

A subroutine is invoked with a **`call`** statement:

```
call foo()
```

**Code:**

```
program printone
  implicit none
  call printint(5)
contains
  subroutine printint(invalue)
    implicit none
    integer :: invalue
    print *,invalue
  end subroutine printint
end program printone
```

**Output**
**[funcf] printone:**

```
           5
```

Arguments types are defined in the body, not the header

**Code:**

```
program addone
  implicit none
  integer :: i=5
  call addint(i,4)
  print *,i
contains
  subroutine addint(inoutvar,addendum)
    implicit none
    integer :: inoutvar,addendum
    inoutvar = inoutvar + addendum
  end subroutine addint
end program addone
```

**Output**
**[funcf] addone:**

```
           9
```

Parameters are always ‘by reference’!

Recursive functions in Fortran need to be explicitly declared as such, with the **recursive** keyword.

Declare function as **Recursive Function**

**Code:**

```
recursive integer function fact(invalue) &
     result (val)
  implicit none
  integer,intent(in) :: invalue
  if (invalue==0) then
     val = 1
  else
     val = invalue * fact(invalue-1)
  end if
end function fact
```

**Output**
**[funcf] fact:**

```
echo 7 | ./fact
           7 factorial is
     5040
```

Note the **result** clause. This prevents ambiguity.

## 34.2 Return results

While a **subroutine** can only return information through its parameters, a *function* procedure returns an explicit result:

```
logical function test(x)
  implicit none
  real :: x

  test = some_test_on(x)
  return ! optional, see above
end function test
```

You see that the result is not returned in the **return** statement, but rather through assignment to the function name. The **return** statement, as before, is optional and only indicates where the flow of control ends.

A *function* in Fortran is a procedure that return a result to its calling program, much like a non-void function in C++

- **subroutine** vs **function**:
  compare *void* functions vs non-void in C++.
- Function header:
  Return type, keyword **function**, name, parameters
- Function body has statements
- Result is returned by assigning to the function name
- Use: `y = f(x)`

**Code:**

```
1  program plussing
2    implicit none
3    integer :: i
4    i = plusone(5)
5    print *,i
6  contains
7    integer function plusone(invalue)
8      implicit none
9      integer :: invalue
10     plusone = invalue+1 ! note!
11   end function plusone
12 end program plussing
```

**Output**

```
[funcf] plusone:
              6
```

- The function name is a variable
- ... that you assign to.

A function is not invoked with **call**, but rather through being used in an expression:

```
if (test(3.0) .and. something_else) ...
```

You now have the following cases to make the function known in the main program:

- If the function is in a **contains** section, its type is known in the main program.

- If the function is in a module (see section 38.2 below), it becomes known through a `use` statement.

  *F77 note:* Without modules and `contains` sections, you need to declare the function type explicitly in the calling program. The safe way is through using an `interface` specification.

**Exercise 34.2.** Write a program that asks the user for a positive number; non-positive input should be rejected. Fill in the missing lines in this code fragment:

**Code:**

```
program readpos
  implicit none
  real(4) :: userinput
  print *,"Type a positive number:"
  userinput = read_positive()
  print *,"Thank you for",userinput
contains
  real(4) function read_positive()
    implicit none
  !! ...
  end function read_positive
end program readpos
```

**Output**

**[funcf] readpos:**

```
 Type a positive number:
 No, not  -5.00000000
 No, not   0.00000000
 No, not  -3.14000010
 Thank you for   2.48000002
```

### 34.2.1 The 'result' keyword

Apart from assigning to the function name, there is a second mechanism for returning a function result, namely through the `Result` keyword.

```
function some_function() result(x)
  implicit none
  real :: x
  !! stuff
  x = ! some computation
end function
```

You see that here

- the assignment to the name is missing,
- the function name is not typed; but
- instead there is a typed local variable that is marked to be the result.

### 34.2.2 The 'contains' clause

```
Program NoContains
  implicit none
  call DoWhat()
end Program NoContains

subroutine DoWhat(i)
  implicit none
  integer :: i
  i = 5
end subroutine DoWhat
```

Warning only, crashes.

```
Program ContainsScope
  implicit none
  call DoWhat()
contains
  subroutine DoWhat(i)
    implicit none
    integer :: i
    i = 5
  end subroutine DoWhat
end Program ContainsScope
```

Error, does not compile

**Code:**

```
1  Program NoContainTwo
2    implicit none
3    integer :: i=5
4    call DoWhat(i)
5  end Program NoContainTwo
6
7  subroutine DoWhat(x)
8    implicit none
9    real :: x
10   print *,x
11 end subroutine DoWhat
```

**Output**

**[funcf] nocontaintype:**

```
nocontain2.F90:15:16:

   15 |    call DoWhat(i)
      |                1
Warning: Type mismatch in
    argument 'x' at (1); passed
    INTEGER(4) to REAL(4)
    [-Wargument-mismatch]
   7.00649232E-45
```

At best compiler warning if all in the same file

## 34.3 Arguments

Arguments are declared in procedure body:

```
subroutine f(x,y,i)
  implicit none
  integer,intent(in) :: i
  real(4),intent(out) :: x
  real(8),intent(inout) :: y
  x = 5; y = y+6
end subroutine f
! and in the main program
call f(x,y,5)
```

declaring the 'intent' is optional, but highly advisable.

- Everything is passed by reference.
  Don't worry about large objects being copied.

- Optional intent declarations:
  Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.

The term *dummy argument* is what Fortran calls the parameters in the procedure definition:

```
subroutine f(x) ! 'x' is dummy argument
```

The arguments in the procedure call are the *actual argument*s:

```
call f(x) ! 'x' is actual argument
```

Compiler checks your intent against your implementation. This code is not legal:

```
subroutine ArgIn(x)
  implicit none
  real,intent(in) :: x
  x = 5 ! compiler complains
end subroutine ArgIn
```

Self-protection: if you state the intended behavior of a routine, the compiler can detect programming mistakes.

Allow compiler optimizations:

```
x = f()
call ArgOut(x)
print *,x
```

Call to `f` removed

```
do i=1,1000
  x = ! something
  y1 = .... x ....
  call ArgIn(x)
  y2 = ! same expression as y1
```

`y2` is same as `y1` because `x` not changed

(May need further specifications, so this is not the prime justification.)

**Exercise 34.3.** Write a subroutine `trig` that takes a number $\alpha$ as input and passes $\sin\alpha$ and $\cos\alpha$ back to the calling environment.

### 34.3.1 Keyword and optional arguments

The arguments in a procedure call can always be given with their corresponding parameter name. This is called a *keyword argument*, and it is sometimes useful to prevent confusion.

```
! confusing:
call two_point( 1.1, 2.2, 3.3, 4.4 )
! better:
call two_point( x1=1.1, x2=2.2, y1=3.3, y2=4.4 )
```

Arguments not given with a keyword are called *positional argument*s. You can mix positional and keyword arguments, but if you give one argument by keyword, all subsequent ones also need their keyword.

- Use the name of the *formal parameter* as keyword.
- Keyword arguments have to come last.

**Code:**

```
1   call say_xy(1,2)
2   call say_xy(x=1,y=2)
3   call say_xy(y=2,x=1)
4   call say_xy(1,y=2)
5   ! call say_xy(y=2,1) ! ILLEGAL
6 contains
7   subroutine say_xy(x,y)
8     implicit none
9     integer,intent(in) :: x,y
10    print *,"x=",x,", y=",y
11  end subroutine say_xy
```

**Output**

```
[funcf] keyword:
 x=           1 , y=           2
 x=           1 , y=           2
 x=           1 , y=           2
 x=           1 , y=           2
```

A relation notion is that of *optional argument*s. A parameter can be marked `optional`, after which it can be omitted from a procedure call.

- Optional parameters can be anywhere in the parameter list;
- If you omit one optional parameter in the argument list, all subsequent arguments need to be given by keyword.
- The procedure can test whether or not an optional argument was supplied with the function `Present`

- Extra specifier: `Optional`
- Presence of argument can be tested with `Present`

## 34.4 Types of procedures

Procedures that are in the main program (or another type of program unit), separated by a `contains` clause, are known as *internal procedure*s. This is as opposed to *module procedure*s.

There are also *statement functions*, which are single-statement functions, usually to identify commonly used complicated expressions in a program unit. Presumably the compiler will *inline* them for efficiency.

The standard library functions, such as `sqrt`, can be declared as such in an `intrinsic` statement

```
Intrinsic :: sqrt,cmplx
```

but this is not necessary.

The `entry` statement is so bizarre that I refuse to discuss it.

## 34.5 Local variable `save`-ing

Normally, local variables in a procedure act as if they get created when the procedure is invoked, and disappear again when its execution ends. It is possible to retain the value of a variable between invocations by giving it an attribute of `save`.

```
subroutine whatever()
integer,save :: i
```

(This corresponds roughly to a *static* variable in C++.)

Here is a major pitfall. If you give a local variable an initialization value:

```
subroutine whatever()
integer :: i = 5
```

then the variable implicitly gets a *save* attribute, whether this is specified or not. The initialization is only executed once, probably at compile time, and at the second procedure invocation the saved value is used.

This may trip you up as the following example shows:

Local variable is initialized only once,
second time it uses its retained value.

**Code:**

```
1 integer function maxof2(i,j)
2   implicit none
3   integer,intent(in) :: i,j
4   integer :: max=0
5   if (i>max) max = i
6   if (j>max) max = j
7   maxof2 = max
8 end function maxof2
```

**Output**

**[funcf] save:**

```
Comparing:  1   3
            3
Comparing: -2  -4
            3
```

# Chapter 35

# Scope

## 35.1 Scope

Fortran 'has no curly brackets': you not easily create nested scopes with local variables as in C++. For instance, the range between `do` and `end do` is not a scope. This means that all variables have to be declared at the top of a program or subprogram.

### 35.1.1 Variables local to a program unit

Variables declared in a subprogram have similar scope rules as in C++:

- Their visibility is controlled by their textual scope:

```
Subroutine Foo()
  integer :: i
  ! 'i' can now be used
  call Bar()
  ! 'i' still exists
End Subroutine Foo
Subroutine Bar() ! no parameters
  ! The 'i' of Foo is unknown here
End Subroutine Bar
```

- Their dynamic scope is the lifetime of the program unit in which they are declared:

```
Subroutine Foo()
  call Bar()
  call Bar()
End Subroutine Foo
Subroutine Bar()
  Integer :: i
  ! 'i' is created every time Bar is called
End Subroutine Bar
```

  (That last example has a little subtlety; see section 34.5 for the `Save` attribute on procedure variables.)

#### *35.1.1.1 Variables in a module*

Variables in a module (section 38.2) have a lifetime that is independent of the calling hierarchy of program units: they are *static variables*.

#### *35.1.1.2 Other mechanisms for making static variables*

Before Fortran gained the facility for recursive functions, the data of each function was placed in a statically determined location. This meant that the second time you call a function, all variables still have the value that they had last time. To force this behavior in modern Fortran, you can add the `Save` specification to a variable declaration.

Another mechanism for creating static data was the **`Common`** block. This should not be used, since a **`Module`** is a more elegant solution to the same problem.

### 35.1.2 Variables in an internal procedure

An *internal procedure* (that is, one placed in the **`Contains`** part of a program unit) can receive arguments from the containing program unit. It can also access directly any variable declared in the containing program unit, through a process called *host association*.

The rules for this are messy, especially when considering implicit declaration of variables, so we advise against relying on it.

# Chapter 36

# String handling

## 36.1 String denotations

A string can be enclosed in single or double quotes. That makes it easier to have the other type in the string.

```
print *,'This string was in single quotes'
print *,'This string in single quotes contains a single '' quote'
print *,"This string was in double quotes"
print *,"This string in double quotes contains a double "" quote"
```

## 36.2 Characters

The datatype `Character` is used both for characters and strings. Therefore, see next section.

## 36.3 Strings

The length of a Fortran string is specified with the `len` keyword when the string is created:

```
character(len=50) :: mystring
mystring = "short string"
```

The `len` function also gives the length of the string, but note that that is the length with which it was allocated, not how much non-blank content you put in it.

String length, with / without trimming.

**Code:**

```
1 character(len=12) :: strvar
2 !! ...
3 strvar = "word"
4 print *,len(strvar),len(trim(strvar))
```

**Output**

```
[stringf] strlen:
           12           4
```

To get the more intuitive length of a string, that is, the location of the last non-blank character, you need to `trim` the string.

Concatenation is done with a double slash:

**Code:**

```
character(len=10) :: firstname,lastname
character(len=15) :: shortname,fullname
!! ...
firstname = "Victor"; lastname =
    "Eijkhout"
shortname = firstname // lastname
print *,"without trimming: ",shortname
fullname = trim(firstname) // " " //
    trim(lastname)
print *,"with trimming: ",fullname
```

**Output**

**[stringf] concat:**

```
without trimming: Victor
   Eijkh
with trimming: Victor Eijkhout
```

## 36.4 Conversions

Sometimes we want to convert between the string `123` and the number 123. Let's start easy, by looking at characters and their *ascii* codes.

### 36.4.1 Character conversions

Given an integer, `Char` gives the character with that ascii code. That can be a printable or an unprintable character:

**Code:**

```
print *,"97 is a:",char(97)
print *,"84 is T:",char(84)
print *,"53 is 5:",char(53)
print *,"11 is VT :",char(11),"."
```

**Output**

**[stringf] ascii:**

```
97 is a:a
84 is T:T
53 is 5:5
11 is VT :
          .
```

Note the last one!

In the other direction, `Iachar` gives the ascii code of a character.

```
character :: char
integer :: code

char = "x"
code = iachar(char)
print *,char," has code",code
```

**Remark 22** *There is also a function* `Ichar`*, but it returns the code in the native character set. In rare cases this can be something other than ascii.*

**Exercise 36.1.** Write a test to see if a character is lowercase:

**Code:**

```
print *,"lower t",islower("t")
print *,"lower T",islower("T")
print *,"lower 3",islower("3")
```

**Output**
**[stringf] lower:**

```
 lower t T
 lower T F
 lower 3 F
```

Similarly, write a test `isdigit`.

### 36.4.2 String conversions

Converting between a number and a string relies on concept from the I/O chapter (chapter 42); see section 42.5.

## 36.5 Further notes

In addition to the character definition with the `len` specification, there is

```
character*80 :: str
character,dimension(80) :: str
character :: str(80
```

These should not be used.

# Chapter 37

# Structures, eh, types

Fortran has structures for bundling up data, but there is no `struct` keyword: instead you declare both the structure type and variables of that *derived type* with the **`type`** keyword.

## 37.1 Derived type basics

Now you need to

- Define the type to describe what's in it;
- Declare variables of that type; and
- use those variables, but setting the type members or using their values.

**`Type name`** / **`End Type name`** block.
Member declarations inside the block:

```
type mytype
  integer :: number
  character :: name
  real(4) :: value
end type mytype
```

Type definitions go before executable statements.

Creating type variables is a little different from objects in a C++ class. In C++ the class name could be used by itself as the datatype; in Fortran you need to write **`Type`**`(mytype)`. Otherwise, it looks like any other variable declaration.

Declare type variables in the main program:

```
Type(mytype) :: struct1,struct2
```

Initialize with type name:

```
struct1 = mytype( 1, 'my_name', 3.7 )
```

Copying:

```
struct2 = struct1
```

If you need access to a single field in a type, there is a notation analogous to the 'dot' notation in C++: in Fortran you use the percent sign `%`.

Access structure members with `%`
(compare C++ dot-notation)

```
Type(mytype) :: typed_struct
typed_struct%member = ....
```

As an example, we use the 'point' structure from the geometry project.

```
type point
   real :: x,y
end type point
```

```
type(point) :: p1,p2
p1 = point(2.5, 3.7)

p2 = p1
print *,p1
print *,p2%x,p2%y
```

Note that printing a type by itself is equivalent to printing its components in sequence.

You can have arrays of types:

```
type(my_struct) :: data
type(my_struct),dimension(10) :: data_array
```

## 37.2 Derived types and procedures

Structures can be passed as procedure argument, just like any other datatype. In this example the function *length*:

- Takes a structure of **type**(*point*) as argument; and
- returns a **real**(4) result.
- The structure is declared as **intent**(**in**).

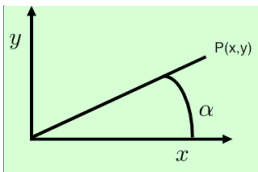
Function with structure argument:

```
real(4) function length(p)
  implicit none
  type(point),intent(in) :: p
  length = sqrt( &
       p%x**2 + p%y**2 )
end function length
```

Function call

```
print *,"Length:",length(p2)
```

**Exercise 37.1.** Add a function *angle* that takes a *Point* argument and returns the angle of the $x$-axis and the line from the origin to that point.




Your program should read in the $x, y$ values of the point and print out the angle in radians.

Bonus: can you print the angle as a fraction of $\pi$? So

$$(1, 1) \Rightarrow 0.25$$

*You can base this off the file `point.F90` in the repository*

**Exercise 37.2.** Write a program that has the following:

- A type `Point` that contains real numbers `x,y`;
- a type `Rectangle` that contains two `Point`s, corresponding to the lower left and upper right point;
- a function `area` that has one argument: a `Rectangle`.

Your program should

- Accept two real numbers on one line, for the bottom left point;
- similarly, again on one line, the coordinates of the top right point; then
- print out the area of the (axi-parallel) rectangle defined by these two points.

**Exercise 37.3.** In the previous exercise 37.2:
Bonus points for using a module,
double bonus points for using an object-oriented solution.

## 37.3 Parameterized types

If a derived type contains an array, you may want to have the length of that array to be variable, without making the array dynamically allocatable. For this, Fortran has *parametrized* types: you can define a type with some combination of:

- a parameter with attribute **`len`**, used as the length of an array member; or
- a parameter with attribute **`kind`**, used as the kind of some variable; section 31.4.1.2.

Example:

```
type point(dim)
   integer,len :: dim
   real,dimension(dim) :: x
end type point
```

I haven't figured out how to set variables:

```
type(point(3)) :: p1,p2
p1%x = [1.,2.,3.]

p2 = p1
print *,p2%x
```

These types can be passed normally:

```
real(4) function length(p)
  implicit none
  type(point(3)),intent(in) :: p
  length = sqrt( &
       p%x(1)**2 + p%x(2)**2 + p%x(3)**2 )
end function length
```

# Chapter 38

# Modules

Fortran has a clean mechanism for importing data (including numeric constants such as $\pi$), functions, types that are defined in another file. This is done through modules, defined with the `module` keyword.

Modules look like a program, but without main
(only 'stuff to be used elsewhere'):

```
Module geometry
  type point
     real :: x,y
  end type point
  real(8),parameter :: pi = 3.14159265359
contains
  real(4) function length(p)
    implicit none
    type(point),intent(in) :: p
    length = sqrt( p%x**2 + p%y**2 )
  end function length
end Module geometry
```

Note also the numeric constant.

Module imported through `use` statement;
placed before `implicit none`

**Code:**

```
1  Program size
2    use geometry
3    implicit none
4
5    type(point) :: p1,p2
6    p1 = point(2.5, 3.7)
7
8    p2 = p1
9    print *,p2%x,p2%y
10   print *,"length:",length(p2)
11   print *,2*pi
12
13 end Program size
```

**Output**

```
[structf] typemod:
   2.50000000       3.70000005
 length:   4.46542263
   6.2831854820251465
```

**Exercise 38.1.** Take exercise 37.2 and put all type definitions and all functions in a module.

## 38.1 Modules for program modularization

Modules are Fortran's mechanism for supporting *separate compilation*: you can put your module in one file, your main program in another, and compile them separately.

A module is a container for definitions of subprograms and types, and for data such as constants and variables. A module is not a structure or object: there is only one instance.

**Remark 23** *The* `use` *statement is somewhat similar to an* `#include "stuff.h"` *line in C++. However, note that C++20 has also adopted modules, as cleaner than preprocessor-based solutions.*

## 38.2 Module definition

What do you use a module for?

- Type definitions: it is legal to have the same type definition in multiple program units, but this is not a good idea. Write the definition just once in a module and make it available that way.
- Function definitions: this makes the functions available in multiple sources files of the same program, or in multiple programs.
- Define constants: for physics simulations, put all constants in one module and use that, rather than spelling out the constants each time.
- Global variables: put variables in a module if they do not fit an obvious scope.

Any routines come after the `contains` clause.

**Remark 24** *Modules were introduced in Fortran90. In earlier standards, information could be made globally available through* `common` *blocks. Since modules are much cleaner than common blocks, do not use those anymore.*

A module is made available with the `use` keyword, which needs to go before the `implicit none`.

`Use` statement placed before `Implicit`

```
Program ModProgram
  use FunctionsAndValues
  implicit none

  print *,"Pi is:",pi
  call SayHi()

End Program ModProgram
```

Also possible:

```
Use mymodule, Only: func1,func2
Use mymodule, func1 => new_name1
```

If you compile a module, you will find a `.mod` file in your directory. (This is little like a `.h` file in C++.) If this file is not present, you can not **use** the module in another program unit, so you need to compile the file containing the module first.

**Exercise 38.2.** Write a module *PointMod* that defines a type *Point* and a function *distance* to make this code work:

```
use pointmod
implicit none
type(Point) :: p1,p2
real(8) :: p1x,p1y,p2x,p2y
read *,p1x,p1y,p2x,p2y
p1 = point(p1x,p1y)
p2 = point(p2x,p2y)
print *,"Distance:",distance(p1,p2)
```

Put the program and module in two separate files and compile thusly:

```
ifort -g -c pointmod.F90
ifort -g -c pointmain.F90
ifort -g  -o pointmain pointmod.o pointmain.o
```

## 38.3 Separate compilation

The exercises in this course are simple enough that you can include any modules in the same file as your main program. However, in realistic applications you will have a separate files for modules, maybe even using one file per module.

Suppose program is split over two files:
`theprogram.F90` and `themodule.F90`.

- Compile the module: `ifort -c themodule.F90`; this gives
- an *object file* (extension: `.o`) that will be linked later, and
- a module file `modulename.mod`.
- Compile the main program:
  `ifort -c theprogram.F90` will read the `.mod` file; and finally
- Link the object files into an *executable*:
  `ifort -o myprogram theprogram.o themodule.o`
  The compiler is used as *linker*: there is no compiling in this step.

Important: the module needs to be compiled before any (sub)program that uses it.

The Fortran2008 standard introduced *sub module*s, which can even further facilitate separate compilation.

## 38.4 Access

By default, all the contents of a module is usable by a subprogram that uses it. However, a keyword **private** make module contents available only inside the module. You can make the default behavior

explicit by using the **public** keyword. Both **public** and **private** can be used as attributes on definitions in the module. There is a keyword **protected** for data members that are public, but can not be altered by code outside the module.

```
Module settings
  implicit none
  logical,protected ::
    has_been_initialized = .FALSE.
contains
  subroutine init()
    has_been_initialized = .TRUE.
  end subroutine init
End Module settings
```

```
if ( .not. has_been_initialized )
    then
   call init()
end if
!! WRONG does not compile:
! has_been_initialized = .FALSE.
```

## 38.5 Polymorphism

```
module somemodule

INTERFACE swap
MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
END INTERFACE

  contains
  subroutine swapreal
  ...
  end subroutine swapreal
  subroutine swapint
  ...
  end subroutine swapint
```

## 38.6 Operator overloading

You can define operations such as + or * on types.

```
Module Typedef
  Type inttype
     integer :: value
  end type inttype
  Interface operator(+)
     module procedure addtypes
  end Interface operator(+)
contains
```

```
  function addtypes(i1,i2)
    result(isum)
    implicit none
    Type(inttype),intent(in) :: i1,i2
    Type(inttype) :: isum
    isum%value = i1%value+i2%value
  end function addtypes
end Module Typedef
```

You can now make the code look nice and simple:

**Code:**

```
1 Type(inttype) :: i1,i2,i3
2 i1 = inttype(1); i2 = inttype(2)
3 i3 = i1+i2
4 print *,"Sum:",i3%value
```

**Output**

```
[typef] plus:
 Sum:            3
```

Overloading includes the assignment operator:

```
INTERFACE ASSIGNMENT (=)
subroutine_interface_body
END INTERFACE
```

You can define new operators with a dot-notation:

```
INTERFACE OPERATOR (.DIST.)
MODULE PROCEDURE calcdist
END INTERFACE
```

# Chapter 39

# Classes and objects

## 39.1 Classes

Fortran classes are based on `type` objects. Some aspects are similar to C++. For instance, the same syntax is used for specifying data members and methods:

```
print *,myobject%xfield
myobject%set_xfield(5.1)
```

Other aspects are a little different: in C++ you can write in one class definition all data and function members; in Fortran data and functions are declared separately.

A big difference is in how function methods are defined: the object itself becomes an extra parameter. You will see the details later.

First about how Fortran classes are organized. A class is a type definition inside a module, with an extra clause indicating what function methods are available for the type.

You define a type as before, with its data members, but now the type has a `contains` for the methods:

```
Module multmod

  type Scalar
     real(4) :: value
   contains
     procedure,public :: &
          printme,scaled
  end type Scalar

contains ! methods
  !! ...
end Module multmod
```

As stated above, calling methods on an object uses the same syntax as accessing its data members.

Method call similar to C++

**Code:**

```
1  Program Multiply
2    use multmod
3    implicit none
4
5    type(Scalar) :: x
6    real(4) :: y
7    x = Scalar(-3.14)
8    call x%printme()
9    y = x%scaled(2.)
10   print '(f7.3)',y
11
12 end Program Multiply
```

**Output**

```
[objectf] mult1:

The value is -3.140
 -6.280
```

The method definition works slightly different from C++, but if you know python you'll see the similarity. If a method is called with one argument;

```
call obj%fun(arg)
```

the function has two parameters, the first one being the object, and the second one the parenthesized argument.

Additionally, the first parameter is of type `Type(obj)`, but in the method it is declared as `Class(obj)`.

Note the extra first parameter:
which is a **Type** but declared here as **Class**:

```
subroutine printme(me)
  implicit none
  class(Scalar) :: me
  print '("The value is",f7.3)',me%value
end subroutine printme
function scaled(me,factor)
  implicit none
  class(Scalar) :: me
  real(4) :: scaled,factor
  scaled = me%value * factor
end function scaled
```

In summary:

- A class is a **Type** with a **contains** clause
  followed by **procedure** declarations,
- ... contained in a module.
- Actual methods go in the **contains** part of the module
- First argument of method is the object itself.

```
Module PointClass
  Type,public :: Point
     real(8) :: x,y
   contains
     procedure, public :: &
        distance
  End type Point
contains
  !! ... distance function ...
  !! ...
End Module PointClass
```

```
Program PointTest
  use PointClass
  implicit none
  type(Point) :: p1,p2

  p1 = point(1.d0,1.d0)
  p2 = point(4.d0,5.d0)

  print *,"Distance:",&
      p1%distance(p2)

End Program PointTest
```

| | C++ | Fortran |
|---|---|---|
| Members | in the object | in the 'type' |
| Methods | in the object | interface: in the type<br>implementation: in the module |
| Constructor | default or explicit | none |
| object itself | 'this' | first argument |
| Class members | global variable | accessed through first arg |
| Object's methods | period | percent |

**Exercise 39.1.** Take the point example program and add a distance function:

```
  Type(Point) :: p1,p2
  ! ... initialize p1,p2
  dist = p1%distance(p2)
  ! ... print distance
```

*You can base this off the file* `pointexample.F90` *in the repository*

**Exercise 39.2.** Write a method `add` for the `Point` type:

```
  Type(Point) :: p1,p2,sum
  ! ... initialize p1,p2
  sum = p1%add(p2)
```

What is the return type of the function `add`?

### 39.1.1 Final procedures: destructors

The Fortran equivalent of *destructors* is a *final procedure*, designated by the *final* keyword.

```
 contains
   final :: &
        print_final
end type Scalar
```

A final procedure has a single argument of the type that it applies to:

```
subroutine print_final(me)
  implicit none
```

```
  type(Scalar) :: me
  print '("On exit: value is",f7.3)',me%value
end subroutine print_final
```

The final procedure is invoked when a derived type object is deleted, except at the conclusion of a program:

```
  call tmp_scalar()
contains
  subroutine tmp_scalar()
    type(Scalar) :: x
    real(4) :: y
    x = Scalar(-3.14)
  end subroutine tmp_scalar
```

## 39.2 Inheritance

Inheritance:

```
type,extends(baseclas) :: derived_class
```

Pure virtual:

```
type,abstract
```

http://fortranwiki.org/fortran/show/Object-oriented+programming

It is of course best to put the type definition and method definitions in a module, so that you can **use** it.

Mark methods as **private** so that they can only be used as part of the **type**:

```
Module PointClass
  !! ...
  private
contains
  subroutine setzero(p)
    implicit none
    class(point) :: p
    p%x = 0.d0 ; p%y = 0.d0
  end subroutine setzero
  !! ...
End Module PointClass
```

## 39.3 Operator overloading

For many physical quantities it makes sense to define an addition operator. This makes it possible to write

```
Type(X) :: x,y,z
! stuff
x = y+z
```

For purposes of exposition, let's make a very simple class:

```
Type,public :: ScalarField
   real(8) :: value
 contains
   procedure,public :: set,print
   procedure,public :: add
End type ScalarField
```

We define a couple of obvious methods:

```
subroutine set(v,x)
  implicit none
  class(ScalarField) :: v
  real(8),intent(in) :: x

  v%value = x
end subroutine set

subroutine print(v)
  implicit none
  class(ScalarField) :: v

  print '(f7.4)', v%value
end subroutine print
```

```
call u%set(2.d0)
call v%set(1.d0)
!  z = u%add(v)
z = u+v
```

Before we can define the addition operator, it is first necessary to define an addition function:

```
function add(in1,in2) result(out)
  implicit none
  class(ScalarField),intent(in) :: in1
  type(ScalarField),intent(in) :: in2
  type(ScalarField) :: out

  out%value = in1%value + in2%value
end function add
```

This function needs to satisfy some conditions:

- The function needs to have two input parameters. Obviously.
- The input parameters need to be declared `Intent(In)`. This is a little less obvious, but it makes sense, because the arguments to the addition parameter are not really passed the normal way.

Turning the function into an operator is then pretty simple.

Interface block:

```
interface operator(+)
   module procedure add
end interface operator(+)
```

**Exercise 39.3.** Extend the above example program so that the type stores an array instead of a scalar.

**Code:**

```
integer,parameter :: size = 12

Type(VectorField) :: u,v,z

call u%alloc(size)
call v%alloc(size)
call u%setlinear()
call v%setconstant(1.d0)
!  z = u%add(v)
z = u+v
call z%print()
```

**Output**

```
[geomf] field:
 2.0000  3.0000  4.0000  5.0000
    6.0000  7.0000  8.0000
   9.0000 10.0000 11.0000
   12.0000 13.0000
```

*You can base this off the file `scalar.F90` in the repository*

Similarly, we can redefine the assignment operator; see https://dannyvanpoucke.be/oop-fortran-tut5-en/. This comes with some complications regarding *shallow copy* and *deep copy*.

# Chapter 40

# Arrays

Array handling in Fortran is similar to C++ in some ways, but there are differences, such as that Fortran indexing starts at 1, rather than 0. More importantly, Fortran has better handling of multi-dimensional arrays, and it is easier to manipulate whole arrays.

## 40.1 Static arrays

The preferred way for specifying an array size is:

Creating arrays through `dimension` keyword:

```
real(8), dimension(100) :: x,y
```

One-dimensional arrays of size 100.

```
integer,dimension(10,20) :: iarr
```

Two-dimensional array of size $10 \times 20$.

These arrays are statically defined, and only live inside their program unit (subroutine, function, module).
Dynamic allocation later.

Such an array, with size explicitly indicated, is called a *static array* or *automatic array*. (See section 40.4 for dynamic arrays.)

Array indexing in Fortran is 1-based by default:

```
integer,parameter :: N=8
real(4),dimension(N) :: x
do i=1,N
  ... x(i) ...
```

Different from C/C++.

Note the use of `parameter`: *compile-time constant*
Size needs to be known to the compiler.

Unlike C++, Fortran can specify the lower bound explicitly:

```
real,dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Preferred: use `lbound` and `ubound`
(see also 40.2.1)

**Code:**

```
1 real,dimension(-1:7) :: array
2 integer :: idx
3 !! ...
4 do idx=lbound(array,1),ubound(array,1)
5    array(idx) = 1+idx/10.
6    print *,array(idx)
7 end do
```

**Output**
**[arrayf] lubound:**

```
 0.899999976
  1.00000000
  1.10000002
  1.20000005
  1.29999995
  1.39999998
  1.50000000
  1.60000002
  1.70000005
```

Such arrays, as in C++, obey the scope: they disappear at the end of the program or subprogram.

### 40.1.1 Initialization

There are various syntaxes for *array initialization*, including the use of *implicit do-loops*:

Different syntaxes:

- Explicit:

```
real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
```

- Implicit do-loop:

```
real5 = [ (1.01*i,i=1,size(real5,1)) ]
```

- Legacy syntax

```
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

(This is pre-Fortran2003. Slashes were also used for some other deprecated constructs.)

### 40.1.2 Array sections

Fortran is more sophisticated than C++ in how it can handle arrays as a whole. For starters, you can assign one array to another:

```
real*8, dimension(10) :: x,y
x = y
```

This obviously requires the arrays to have the same size. You can assign subarrays, or *array section*s, as long as they have the same shape. This uses a colon syntax.

- `A(:)` to get all indices,
- `A(:n)` to get indices up to `n`,
- `A(n:)` to get indices `n` and up.
- `A(m:n)` indices in range `m,...,n`.

Assignment from one section to another:

**Code:**

```
real(8),dimension(5) :: x = &
      [.1d0, .2d0, .3d0, .4d0, .5d0]
!! ...
x(2:5) = x(1:4)
print '(f5.3)',x
```

**Output**
**[arrayf] sectionassign:**

```
0.100
0.100
0.200
0.300
0.400
```

Note:
Format syntax will be discussed later:
float number, 5 positions, 3 after decimal point.

**Exercise 40.1.** Code out the array assignment

```
x(2:5) = x(1:4)
```

with an explicit indexed loop. Do you get the same output? Why? What conclusion do you draw about internal mechanisms used in array sections?

The above exercise illustrates a point about the *semantics of array operations*: an array statement behaves as if all inputs are gathered together before any results are stored. Conceptually, it is as if the right-hand side is assembled an copied to some temporary locations before being written to the left-hand side. In practice, this may require large temporary arrays (and negatively affect performance by lessening *locality*) so you hope that the compiler does something smarter. However, the exercise showed that an array assignment can not trivially be converted to a simple loop.

**Exercise 40.2.** Can you formalize the sort of array statement for which a simple translation to a loop changes the semantics? (In compiler terminology this is called a *dependence*.)

Array operations can be more sophistaced than assigning to a whole array or a section of it. For instance, you can use a stride:

`X(a:b:c)` : stride `c`

Analogous to: `do i=a,b,c`

Copy a contiguous array to a strided subset of another:

**Code:**

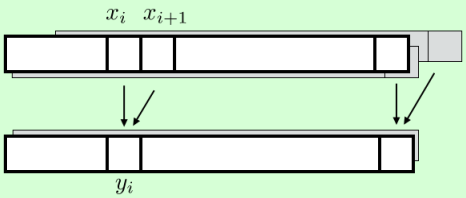
```
integer,dimension(5) :: &
      y = [0,0,0,0,0]
integer,dimension(3) :: &
      z = [3,3,3]
!! ...
y(1:5:2) = z(:)
print '(i3)',y
```

**Output**
**[arrayf] sectionmg:**

```
  3
  0
  3
  0
  3
```

You can even do arithmetic on array sections, for instance adding them together.

**Exercise 40.3.** Code $\forall_i$: $y_i = (x_i + x_{i+1})/2$:




- First with a do loop; then
- in a single array assignment statement by using sections.

Initialize the array `x` with values that allow you to check the correctness of your code.

### 40.1.3 Integer arrays as indices

It's even possible to use a set of indices, stored in an integer array, to access arbitrary locations in an array.

Indexed subset:

```
integer,dimension(4) :: i = [2,3,5,7]
real(4),dimension(10) :: x
print *,x(i)
```

## 40.2 Multi-dimensional

Arrays above had '*rank* one'. The rank is defined as the number of indices you need to address the elements. Mathematically this is not a rank but the dimension of the array, but that word is already taken. We will still use that word, for instance talking about the first and second dimension of an array.

A rank-two array, or matrix, is defined like this:

Declaration and use with parentheses and comma
(compare `a[i][j]` in C++):

```
real(8),dimension(20,30) :: array
array(i,j) = 5./2
```

A useful function is **reshape**.

Reshape: convert 2D array to 1D (or vv)
between arrays with the same number of elements.

Example:

- initialize as 1D,
- reshape to 2D

**Code:**

```
real,dimension(2,2) :: x
x = reshape( [ ( 1.*i,i=1,size(x) ) ],
    shape(x) )
print *,x
```

**Output**
**[arrayf] multi:**

```
   1.00000000        2.00000000
        3.00000000
    4.00000000
```

With multidimensional arrays we have to worry how they are stored in memory. Are they stored row-by-row, or column-by-column? In Fortran the latter choice, also known as *column-major* storage, is used; see figure 40.1.

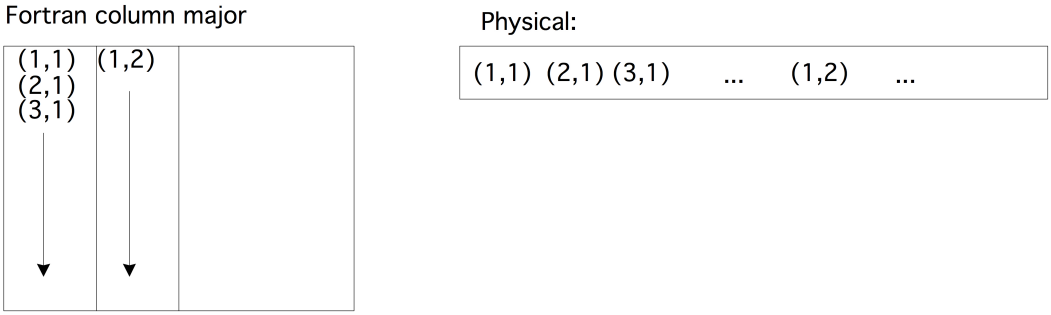



Figure 40.1: Column-major storage in Fortran

To traverse the elements as they are stored in memory, you would need the following code:

```
do col=1,size(A,2)
  do row=1,size(A,1)
    .... A(row,col) .....
  end do
end do
```

This is sometimes described as 'First index varies quickest'. There are various performance-related reasons why such traversal is better than with the loops exchanged.

**Exercise 40.4.** Can you describe in words how memory elements are access if you would write

```
do row=1,size(A,1)
  do col=1,size(A,2)
    .... A(row,col) .....
  end do
```

```
    end do

?
```

You can make sections in multi-dimensional arrays: you need to indicate a range in all dimensions.

```
real(8),dimension(10) :: a,b
a(1:9) = b(2:10)
```

or

```
logical,dimension(25,3) :: a
logical,dimension(25)   :: b
a(:,2) = b
```

You can also use strides.

Fill array by rows, printing is by column:

$$\begin{pmatrix} 1 & 2 & \dots & N \\ N+1 & \dots & & \\ & \dots & & \\ & & & MN \end{pmatrix}$$

**Code:**

```
integer,parameter :: M=4,N=5
real(4),dimension(M,N) :: rect

do i=1,M
   do j=1,N
      rect(i,j) = count
      count = count+1
   end do
end do
print *,rect
```

**Output**

```
[arrayf] printarray:

   1.00000000        6.00000000
        11.0000000
    16.0000000        2.00000000
         7.00000000
    12.0000000        17.0000000
         3.00000000
    8.00000000        13.0000000
         18.0000000
    4.00000000        9.00000000
         14.0000000
    19.0000000        5.00000000
         10.0000000
    15.0000000        20.0000000
```

### 40.2.1 Querying an array

We have the following properties of an array:

- The bounds are the lower and upper bound in each dimension. For instance, after

  ```
  integer,dimension(-1:1,-2:2) :: symm
  ```

  the array `symm` has a lower bound of `-1` in the first dimension and `-2` in the second. The functions **`Lbound`** and **`Ubound`** give these bounds as array or scalar:

  ```
  array_of_lower = Lbound(symm)
  upper_in_dim_2 = Ubound(symm,2)
  ```

**Code:**

```
real(8),dimension(2:N+1) :: Afrom2 = &
     [1,2,3,4,5]
!! ...
lo = lbound(Afrom1,1)
hi = ubound(Afrom1,1)
print *,lo,hi
print '(i3,":",f5.3)', &
     (i,Afrom1(i),i=lo,hi)
```

**Output**
**[arrayf] fsection2:**

```
           1           5
  1:1.000
  2:2.000
  3:3.000
  4:4.000
  5:5.000
```

- The *extent* is the number of elements in a certain dimension, and the **shape** is the array of extents.
- The **size** is the number of elements, either for the whole array, or for a specified dimension.

```
integer :: x(8), y(5,4)
size(x)
size(y,2)
```

### 40.2.2 Reshaping

**RESHAPE**

```
array = RESHAPE( list,shape )
```

Example:

```
square = reshape( (/(i,i=1,16)/),(/4,4/) )
```

**SPREAD**

```
array = SPREAD( old,dim,copies )
```

## 40.3 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:

Passing array as one symbol:

**Code:**

```
real(8),dimension(:) :: x(N) &
     = [ (i,i=1,N) ]
real(8),dimension(:) :: y(0:N-1) &
     = [ (i,i=1,N) ]

sx = arraysum(x)
sy = arraysum(y)
print '("Sum of one-based
    array:",/,4x,f6.3)', sx
print '("Sum of zero-based
    array:",/,4x,f6.3)', sy
```

**Output**

```
[arrayf] arraypass1d:
Sum of one-based array:
    55.000
Sum of zero-based array:
    55.000
```

Note declaration as `dimension(:)`
actual size is queried

```
real(8) function arraysum(x)
  implicit none
  real(8),intent(in),dimension(:) :: x
  real(8) :: tmp
  integer i

  tmp = 0.
  do i=1,size(x)
     tmp = tmp+x(i)
  end do
  arraysum = tmp
end function arraysum
```

The array inside the subroutine is known as a *assumed-shape array* or *automatic array*.

## 40.4 Allocatable arrays

Static arrays are fine at small sizes. However, there are two main arguments against using them at large sizes.

- Since the size is explicitly stated, it makes your program inflexible, requiring recompilation to run it with a different problem size.
- Since they are allocated on the so-called *stack*, making them too large can lead to *stack overflow*.

A better strategy is to indicate the shape of the array, and use `allocate` to specify the size later, presumably in terms of run-time program parameters.

```
! static:
integer,parameter :: s=100
real(8), dimension(s) :: xs,ys

! dynamic
integer :: n
real(8), dimension(:), allocatable :: xd,yd
```

```
read *,n
allocate(xd(n), yd(n))
```

You can **deallocate** the array when you don't need the space anymore.

If you are in danger of running out of memory, it can be a good idea to add a **stat**=*ierror* clause to the **allocate** statement:

```
integer :: ierr
allocate( x(n), stat=ierr )
if ( ierr/=0 ) ! report error
```

Has an array been allocated:

```
Allocated( x ) ! returns logical
```

Allocatable arrays are automatically deallocated when they go out of scope. This prevents the *memory leak* problems of C++.

Explicit deallocate:

```
deallocate(x)
```

### 40.4.1 Returning an allocated array

In an effort to keep the main program nice and abstract, you may want to delegate the **allocate** statement to a procedure. In case of a **Subroutine**, you can pass the (unallocated) array as a parameter. But can you return it from a **Function**?

This requires the **Result** keyword (section 34.2.1:

```
function create_array(n) result(v)
  implicit none
  integer,intent(in) :: n
  real,dimension(:),allocatable :: v
  integer  :: i
  allocate(v(n))
  v = [ (i+.5,i=1,n) ]
end function create_array
```

## 40.5 Array output

The simple statement

```
print *,A
```

will output the element of *A* in memory order; see section 40.2.

For a more sophisticated approach, the main thing to know is that each call to a format statement starts on a new line. Thus

```
print '(10f7.3)',A( ...stuff... )
```

will print 10 elements of the array on each line, before starting a new line of output.

The way to specify the elements of the array is to use implicit do-loops; section 33.3:

```
print '(10f7.3)', (A(i),i=1,size(A))
```

or for multiple dimensions:

```
do row=1,size(A,2)
  print '(10f7.3)', (A(i,row),i=1,size(A,1))
end do
```

What if, in this example, the rows are longer than 10 elements? You can not parametrize the format, but there is no harm in specifying more format than there are array elements:

## 40.6 Operating on an array

### 40.6.1 Arithmetic operations

Between arrays of the same shape:

```
A = B+C
D = D*E
```

(where the multiplication is by element).

### 40.6.2 Intrinsic functions

The following intrinsic functions are avaible for arrays:

- **Abs** creates the matrix of pointwise absolute values.
- **MaxLoc** returns the index of the maximum element.
- **MinLoc** returns the index of the minimum element.
- **MatMul** returns the matrix product of two matrices.
- **Dot_Product** returns the dot product of two arrays.
- **Transpose** returns the transpose of a matrix.
- **Cshift** rotates elements through an array.

Reduction operations on the array itself:

- **MaxVal** finds the maximum value in an array.
- **MinVal** finds the minimum value in an array.
- **Sum** returns the sum of all elements.
- **Product** return the product of all elements.

Reduction operations on a mask derived from the array:

- **All** finds if the mask is true for all elements
- **Any** finds if the mask is true for any element
- **Count** finds for how many elements the mask is true

- Functions such as **Sum** operate on a whole array by default.
- To restrict such a function to one subdimension add a keyword parameter **DIM**:

```
s = Sum(A,DIM=1)
```

  where the keyword is optional.
- Likewise, the operation can be restricted to a *MASK*:

```
s = Sum(A,MASK=B)
```

**Code:**

```
! Summing in I and J
sums = Sum( A,dim=1 )
print *,"Row sums:"
print 10,sums

sums = Sum( A,dim=2 )
print *,"Column sums:"
print 10,sums
10 format( 4(i3,1x) )
```

**Output**

```
[arrayf] rowcolsum:
 Matrix:
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15
 Row sums:
  6  22  38  54
 Column sums:
 24  28  32  36
```

**Exercise 40.5.** The 1-norm of a matrix is defined as the maximum of all sums of absolute values in any column:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

while the infinity-norm is defined as the maximum row sum:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Compute these norms using array functions as much as possible, that is, try to avoid using loops.

For bonus points, write Fortran **Function**s that compute these norms.

**Exercise 40.6.** Compare implementations of the matrix-matrix product.

1. Write the regular `i,j,k` implementation, and store it as reference.
2. Use the **DOT_PRODUCT** function, which eliminates the `k` index. How does the timing change? Print the maximum absolute distance between this and the reference result.
3. Use the **MATMUL** function. Same questions.
4. Bonus question: investigate the `j,k,i` and `i,k,j` variants. Write them both with array sections and individual array elements. Is there a difference in timing?

Does the optimization level make a difference in timing?

### 40.6.3 Restricting with `where`

If an array operation should not apply to all elements, you can specify the ones it applies to with a **`where`** statement.

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )
  sequence of array statements
ELSEWHERE
  sequence of array statements
END WHERE
```

### 40.6.4 Global condition tests

Reduction of a test on all array elements: **`all`**

```
REAL(8),dimension(N,N) :: A
LOGICAL :: positive,positive_row(N),positive_col(N)
positive = ALL( A>0)
positive_row = ALL( A>0,1 )
positive_col = ALL( A>0,2 )
```

**Exercise 40.7.** Use array statements (that is, no loops) to fill a two-dimensional array `A` with random numbers between zero and one. Then fill two arrays `Abig` and `Asmall` with the elements of `A` that are great than $0.5$, or less than $0.5$ respectively:

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i,j) = \begin{cases} 0 & \text{if } A(i,j) \geq 0.5 \\ A(i,j) & \text{otherwise} \end{cases}$$

Using more array statements, add `Abig` and `Asmall`, and test whether the sum is close enough to `A`.

Similar to **`all`**, there is a function **`any`** that tests if any array element satisfies the test.

```
if ( Any( Abs(A-B)>
```

## 40.7 Array operations

### 40.7.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

#### 40.7.1.1 Slicing

If your loop assigns to an array from another array, you can use section notation:

```
a(:) = b(:)
c(1:n) = d(2:n+1)
```

#### 40.7.1.2 'forall' keyword

The `forall` keyword also indicates an array assignment:

```
forall (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end forall
```

You can tell that this is for arrays only, because the loop index has to be part of the left-hand side of every assignment.

What happens if you apply `forall` to a statement with loop-carried dependencies? Consider first the traditional loops

```
A = [1,2,3,4,5]
do i=1,4
   A(i+1) = A(i)
end do
print '(5(i2x))',A
```

```
A = [1,2,3,4,5]
do i=4,1,-1
   A(i+1) = A(i)
end do
print '(5(i2x))',A
```

Can you predict the output? Now consider the following:

**Code:**

```
1 A = [1,2,3,4,5]
2 forall (i=1:4)
3    A(i+1) = A(i)
4 end forall
5 print '(5(i2x))',A
```

**Output**
**[arrayf] forallf:**

```
 1  1  2  3  4
```

**Code:**

```
1 A = [1,2,3,4,5]
2 do i=4,1,-1
3    A(i+1) = A(i)
4 end do
5 print '(5(i2x))',A
```

**Output**
**[arrayf] forallb:**

```
 1  1  2  3  4
```

What does this tell you about the execution?

In other words, this mechanism is prone to misunderstanding and therefore now deprecated. It is not a parallel loop! For that, the following mechanism is preferred.

#### 40.7.1.3 Do concurrent

The *do concurrent* is a true do-loop. With the `concurrent` keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
do concurrent (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end do
```

(Do not use `for all`)

### 40.7.2 Loops without dependencies

Here are some illustrations of simple array copying with the above mechanisms.

```
do i=2,n
   counted(i) = 2*counting(i-1)
end do
```

```
Original     1   2   3   4   5   6   7   8   9  10
Recursive    0   2   4   6   8  10  12  14  16  18
```

```
counted(2:n) = 2*counting(1:n-1)
```

```
Original     1   2   3   4   5   6   7   8   9  10
Section      0   2   4   6   8  10  12  14  16  18
```

```
forall (i=2:n)
   counted(i) = 2*counting(i-1)
end forall
```

```
Original     1   2   3   4   5   6   7   8   9  10
Forall       0   2   4   6   8  10  12  14  16  18
```

```
do concurrent (i=2:n)
   counted(i) = 2*counting(i-1)
end do
```

```
Original     1   2   3   4   5   6   7   8   9  10
Concurrent   0   2   4   6   8  10  12  14  16  18
```

**Exercise 40.8.** Create arrays `A,C` of length `2N`, and `B` of length `N`. Now implement

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \ldots N$$

and

$$C_i = A_{i/2}, \quad i = 1, \ldots 2N$$

using all four mechanisms. Make sure you get the same result every time.

### 40.7.3 Loops with dependencies

For parallel execution of a loop, all iterations have to be independent. This is not the case if the loop has a *recurrence*, and in this case, the 'do concurrent' mechanism is not appropriate. Here are the above four constructs, but applied to a loop with a dependence.

```
do i=2,n
   counting(i) = 2*counting(i-1)
end do
```

```
Original   1   2   3   4   5   6   7   8   9  10
Recursiv   1   2   4   8  16  32  64 128 256 512
```

The slicing version of this:

```
counting(2:n) = 2*counting(1:n-1)
```

```
Original   1   2   3   4   5   6   7   8   9  10
Section    1   2   4   6   8  10  12  14  16  18
```

acts as if the right-hand side is saved in a temporary array, and subsequently assigned to the left-hand side.

Using 'forall' is equivalent to slicing:

```
forall (i=2:n)
   counting(i) = 2*counting(i-1)
end forall
```

```
Original     1   2   3   4   5   6   7   8   9  10
Forall       1   2   4   6   8  10  12  14  16  18
```

On the other hand, 'do concurrent' does not use temporaries, so it is more like the sequential version:

```
do concurrent (i=2:n)
   counting(i) = 2*counting(i-1)
end do
```

```
Original     1   2   3   4   5   6   7   8   9  10
Concurrent   1   2   4   8  16  32  64 128 256 512
```

Note that the result does not have to be equal to the sequential execution: the compiler is free to rearrange the iterations any way it sees fit.

## 40.8 Review questions

**Exercise 40.9.** Let the following declarations be given, and assume that all arrays are properly initialized:

```
real                   :: x
real, dimension(10)    :: a, b
real, dimension(10,10) :: c, d
```

Comment on the following lines: are they legal, if so what do they do?

1. `a = b`

2. `a = x`
3. `a(1:10) = c(1:10)`

How would you:

1. Set the second row of `c` to `b`?
2. Set the second row of `c` to the elements of `b`, last-to-first?

# Chapter 41

# Pointers

Pointers in C/C++ are based on memory addresses; Fortran pointers on the other hand, are more abstract.

## 41.1 Basic pointer operations

Fortran pointers are a little like C or C++ pointers, and they are also different in many ways.

- Like C 'star' pointers, and unlike C++ 'smart' pointers, they can point at anything.
- Unlike C pointers, you have to declare that an object can be pointed at.
- Unlike any sort of pointer in C/C++, but like C++ references, they act as a sort of alias: there is no explicit dereferencing.

We will explore all this in detail.

Fortran pointers act like 'aliases': using a pointer variable is often the same as using the entity it points at. The difference with actually using the variable, is that you can decide what variable the pointer points at.

Fortran pointers are often automatically *dereferenced*: if you print a pointer you print the variable it references, not some representation of the pointer.

**Code:**

```
real,target :: x
real,pointer :: point_at_real

x = 1.2
point_at_real => x
print *,point_at_real
```

**Output**

```
[pointerf] basicp:

   1.20000005
```

Pointers are defined in a variable declaration that specifies the type, with the `pointer` attribute. Examples: the definition

```
real,pointer :: point_at_real
```

declares a pointer that can point at a real variable. Without further specification, this pointer does not point at anything yet, so using it is undefined.

- You have to declare that a variable is point-able:

```
real,target :: x
```

- Declare a pointer:

```
real,pointer :: point_at_real
```

- Set the pointer with => notation (New! Note!):

```
point_at_real => x
```

Now using `point_at_real` is the same as using `x`.

```
print *,point_at_real ! will print the value of x
```

Pointers can not just point at anything: the thing pointed at needs to be declared as **target**

```
real,target :: x
```

and you use the => operator to let a pointer point at a target:

```
point_at_real => x
```

If you use a pointer, for instance to print it

```
print *,point_at_real
```

it behaves as if you were using the value of what it points at.

**Code:**

```
1  real,target :: x,y
2  real,pointer :: that_real
3
4  x = 1.2
5  y = 2.4
6  that_real => x
7  print *,that_real
8  that_real => y
9  print *,that_real
10 y = x
11 print *,that_real
```

**Output**
**[pointerf] realp:**

```
1.20000005
2.40000010
1.20000005
```

1. *that_real* points at x, so the value of x is printed.
2. *that_real* is reset to point at y, so its value is printed.
3. The value of y is changed, and since *that_real* still points at y, this changed value is printed.

## 41.2 Combining pointers

What happens if you point a pointer at another pointer? The concept of pointer-to-pointer from C/C++ does not exist: instead, you two pointers pointing at the same thing.

If you have two pointers

```
real,pointer :: point_at_real,also_point
```

you can make the target of the one to also be the target of the other:

```
point_at_real => x
also_point => point_at_real
```

Note that the second pointer is also assigned with the => symbol. This is not a pointer to a pointer: it assigns the target of the right-hand side to be the target of the left-hand side.

What happens if you want to write `p2=>p1`
but you write `p2=p1`?
The second one is legal, but has different meaning:

Assign underlying variables:

```
real,target :: x,y
real,pointer :: p1,p2

x = 1.2
p1 => x
p2 => y
p2 = p1 ! same as y=x
print *,p2 ! same as print y
```

Crash because `p2` pointer unassociated:

```
real,target :: x
real,pointer :: p1,p2

x = 1.2
p1 => x
p2 = p1
print *,p2
```

**Exercise 41.1.** Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

**Code:**

```
real,dimension(10),target :: array &
     = [1.1, 2.2, 3.3, 4.4, 5.5, &
        9.9, 8.8, 7.7, 6.6, 0.0]
real,pointer :: biggest_element

print '(10f5.2)',array
call SetPointer(array,biggest_element)
print *,"Biggest element
    is",biggest_element
print *,"checking pointerhood:",&
     associated(biggest_element)
biggest_element = 0
print '(10f5.2)',array
```

**Output**

```
[pointerf] arpointf:
 1.10 2.20 3.30 4.40 5.50 9.90
    8.80 7.70 6.60 0.00
 Biggest element is   9.89999962
 checking pointerhood: T
 1.10 2.20 3.30 4.40 5.50 0.00
    8.80 7.70 6.60 0.00
```

*You can base this off the file `arpointf.F90` in the repository*

## 41.3 Pointer status

A pointer can be in three states:

1. a pointer is undefined when it is first created,
2. it can be null, if explicitly set so,
3. or it can be associated if it has been pointed at something.

As a common sense strategy, do not worry about the undefined state: in the example in section 41.5 pointer are quickly made null.

- **Nullify**: zero a pointer
- **Associated**: test whether assigned

**Code:**

```
real,target :: x
real,pointer :: realp

print *,"Pointer starts as not set"
if (.not.associated(realp)) &
   print *,"Pointer not associated"
x = 1.2
print *,"Set pointer"
realp => x
if (associated(realp)) &
   print *,"Pointer points"
print *,"Unset pointer"
nullify(realp)
if (.not.associated(realp)) &
   print *,"Pointer not associated"
```

**Output**
**[pointerf] statusp:**

```
 Pointer starts as not set
 Pointer not associated
 Set pointer
 Pointer points
 Unset pointer
 Pointer not associated
```

You can also specifically test

- **associated**(*p,x*): whether the pointer is associated with the variable, or
- **associated**(*p1,p2*): whether two pointers are associated with the same target.

If you want a pointer to point at something,
but you don't need a variable for that something:

**Code:**

```
Real,pointer :: x_ptr,y_ptr
allocate(x_ptr)
y_ptr => x_ptr
x_ptr = 6
print *,y_ptr
```

**Output**
**[pointerf] allocptr:**

```
   6.00000000
```

(Compare **make_shared** in C++)

## 41.4 Pointers and arrays

You can set a pointer to an array element or a whole array.

```
real(8),dimension(10),target :: array
real(8),pointer              :: element_ptr
real(8),pointer,dimension(:) :: array_ptr
```

```
element_ptr => array(2)
array_ptr   => array
```

More surprising, you can set pointers to array sections:

```
array_ptr => array(2:)
array_ptr => array(1:size(array):2)
```

In case you're wondering, this does not create temporary arrays, but the compiler adds descriptions to the pointers, to translate code automatically to strided indexing.

You can use the `allocate` statement for pointers to arrays:

```
Integer,Pointer,Dimension(:) :: array_point
Allocate( array_point(100) )
```

This is automatically deallocated when control leaves the scope. No memory leaks.

As an even more interesting example of pointers to array sections, let's consider the averaging operation

$$x_{i,j} = (x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1})/4.$$

We need pointers to the interior and its four offsets:

```
real(4),target,allocatable,dimension(:,:) :: grid
real(4),pointer,dimension(:,:) :: interior,left,right,up,down

Allocate( grid(N,N) )
!! ...
interior => grid(2:N-1,2:N-1)
up     => grid(1:N-2,2:N-1)
down   => grid(3:N,2:N-1)
left   => grid(2:N-1,1:N-3)
right  => grid(2:N-1,3:N)
```

The averaging operation is then an array statement:

```
interior = ( up+down+left+right )/4
```

## 41.5 Example: linked lists

For pictures of linked lists, see section 66.1.2.

- Linear data structure
- more flexible than array for insertion / deletion
- ... but slower in access

One of the standard examples of using pointers is the *linked list*. This is a dynamic one-dimensional structure that is more flexible than an array. Dynamically extending an array would require re-allocation, while in a list an element can be inserted.

**Exercise 41.2.** Using a linked list may be more flexible than using an array. On the other hand, accessing an element in a linked list is more expensive, both absolutely and as order-of-magnitude in the size of the list.

Make this argument precise.

### 41.5.1 Type definitions

A list is based on a simple data structure, a node, which contains a value field and a pointer to another node. The list data structure itself only contains a pointer to the first node in the list.

- Node: value field, and pointer to next node.
- List: pointer to head node.

```
type node
   integer :: value
   type(node),pointer :: next
end type node

type list
   type(node),pointer :: head
end type list
```

By way of example, we create a dynamic list of integers, sorted by size. To maintain the sortedness, we need to append or insert nodes, as required.

Our main program will create three nodes, and append them to the end of the list:

**Code:**

```
integer,parameter :: listsize=7
type(list) :: the_list
integer,dimension(listsize) :: inputs = &
     [ 62, 75, 51, 12, 14, 15, 16 ]
integer :: input,input_value

nullify(the_list%head)
do input=1,listsize
   input_value = inputs(input)
   call attach(the_list,input_value)
end do
```

**Output**

```
[pointerf] listappend:
List: [ 62,75,51,12,14,15,16, ]
```

### 41.5.2 Attach a node at the end

First we write a routine `attach` that takes a node pointer and attaches it to the end of a list, without any sorting.

```
subroutine attach( the_list,new_value )
  implicit none
  ! parameters
  type(list),intent(inout) :: the_list
  integer,intent(in) :: new_value
```

We distinguish two cases: when the list is empty, and when it is not. Initially, the list is empty, meaning that the 'head' pointer is un-associated. The first time we add an element to the list, we assign the node as the head of the list:

```
! if the list has no head node, attached the new node
if (.not.associated(the_list%head)) then
   allocate( the_list%head )
   the_list%head%value = new_value
else
   call node_attach( the_list%head,new_value )
end if
```

New element attached at the end.

```
recursive subroutine node_attach( the_node,new_value )
!! ...
  if ( .not. associated(the_node%next) ) then
     allocate( the_node%next )
     the_node%next%value = new_value
  else
     call node_attach( the_node%next,new_value )
  end if
```

**Exercise 41.3.** Take the recursive code for attaching an element, and turn it into an iterative version, that is, use a `while` loop that goes down the list till the end.

You may do the whole thing in the `attach` routine for the list head.

### 41.5.3 Insert a node in sort order

If we want to keep a list sorted, we need in many cases to insert the new node at a location short of the end of the list. This means that instead of iterating to the end, we iterate to the first node that the new one needs to be attached to.

Almost the same as before, but now keep the list sorted:

**Code:**

```
do in=1,listsize
   in_value = inputs(in)
   call insert(the_list,in_value)
   call print(the_list)
end do
```

**Output**

```
[pointerf] listinsert:
List: [  62  ]
List: [  62  75  ]
List: [  51  62  75  ]
List: [  12  51  62  75  ]
List: [  12  14  51  62  75  ]
List: [  12  14  15  51  62  75
    ]
List: [  12  14  15  16  51  62
    75  ]
```

**Exercise 41.4.** Copy the `attach` routine to `insert`, and modify it so that inserting a value will keep the list ordered.

*You can base this off the file `listfappendalloc.F90` in the repository*

**Exercise 41.5.** Modify your code from exercise 41.4 so that the new node is not allocated in the main program.
Instead, pass only the integer argument, and use **`allocate`** to create a new node when needed.

```
call insert(the_list,1)
call insert(the_list,5)
call insert(the_list,3)
```

**Exercise 41.6.** Write a **`print`** function for the linked list.
For the simplest solution, print each element on a line by itself.

More sophisticated: use the **`Write`** function and the **`advance`** keyword:

```
write(*,'(i1",")',advance="no") current%value
```

**Exercise 41.7.** Write a `length` function for the linked list.
Try it both with a loop, and recursively.

# Chapter 42

# Input/output

## 42.1 Types of I/O

Fortran can deal with input/output in ASCII format, which is called *formatted* I/O, and binary, or *unformatted* I/O. Formatted I/O can use default formatting, but you can specify detailed formatting instructions, in the I/O statement itself, or separately in a `Format` statement.

Fortran I/O can be described as *list-directed I/O*: both input and output commands take a list of item, possibly with formatting specified.

- `Print` simple output to terminal
- `Write` output to terminal or file ('unit')
- `Read` input from terminal or file
- `Open`, `Close` for files and streams
- `Format` format specification that can be used in multiple statements.

- Formatted: ASCII output. This is good for reporting, but not for numeric data storage.
- Unformatted: binary output. Great for further processing of output data.
- Beware: binary data is machine-dependent. Use *hdf5* for portable binary.

## 42.2 Print to terminal

The simplest command for outputting data is `print`.

```
print *,"The result is",result
```

In its easiest form, you use the star to indicate that you don't care about formatting; after the comma you can then have any number of comma-separated strings and variables.

### 42.2.1 Print on one line

The statement

```
print *,item1,item2,item3
```

will print the items on one line, as far as the line length allows.

Parameterized printing with an *implicit do loop*:

```
print *,( i*i,i=1,n)
```

All values will be printed on the same line.

### 42.2.2 Printing arrays

If you print a variable that is an array, all its elements will be printed, in *column-major* ordering if the array is multi-dimensional.

You can also control the printing of an array by using an *implicit do loop*:

```
print *,( A(i),i=1,n)
```

## 42.3 Formatted I/O

The default formatting uses quite a few positions for what can be small numbers. To indicate explicitly the formatting, for instance limiting the number of positions used for a number, or the whole and fractional part of a real number, you can use a format string.

For instance, you can use a letter followed by a digit to control the formatting width:

**Code:**

```
i = 56
print *,i
print '(i4)',i
print '(i2)',i
print '(i1)',i
i = i*i
print '("fit <",i0,"> ted")',i
```

**Output**
**[iof] i4:**

```
          56
  56
56
*
fit <3136> ted
```

(In the last line, the format specifier was not wide enough for the data, so an *asterisk* was used as output.)

Let's approach this semi-systematically.

### 42.3.1 Format letters

#### *42.3.1.1 Integers*

Integers can be set with `in`.

- If $n > 0$, that many positions are used with the number right aligned; except
- if the number does not fit in $n$ positions, it is rendered as asterisks.
- To use precisely the required number of positions, use `i0`.

**Code:**

```
i = 56
print *,i
print '(i4)',i
print '(i2)',i
print '(i1)',i
i = i*i
print '("fit <",i0,"> ted")',i
```

**Output**

```
[iof] i4:
          56
  56
56
*
fit <3136> ted
```

#### 42.3.1.2 Strings

Strings can be handled two ways:

1. They can be literally part of the format string:

   ```
   print '(i2,"--",i2)', m,n
   ```

2. The can be formatted with the $an$ specifier:

   ```
   print '(a5,2)',somestring,someint
   ```

3. To use precisely the required number of positions, use $a$

   ```
   print '(a,i0,a)', str1,int2,str3
   ```

#### 42.3.1.3 Floating point

- '$fm.n$' specifies a fixed point representation of a real number, with $m$ total positions (including the decimal point) and $n$ digits in the fractional part.
- '$em.n$' Exponent representation.

#### 42.3.1.4 Other

```
x : one space
x5 : five spaces

b : binary
o : octal
z : hex
```

### 42.3.2 Repeating and grouping

If you want to display items of the same type, you can use a repeat count:

**Code:**

```
i = 12; j = 34
print '(2i4)',i,j
print '(2i2)',i,j
```

**Output**

```
[iof] ii:
  12  34
1234
```

You can mix variables of different types, as well as literal string, by separating them with commas. And you can group them with parentheses, and put a repeat count on those groups again:

**Code:**

```
i = 23; j = 45; k = 67
print '(i2,1x,i2)',i,j
print '("Numbers:",3(1x,i2,"."))', i,j,k
```

**Output**
**[iof] ij:**

```
23 45
Numbers: 23. 45. 67.
```

Putting a number in front of a single specifier indicates that it is to be repeated.

If the data takes more positions than the format specifier allows, a string of asterisks is printed:

**Code:**

```
do ipower=1,5
   print '(i3)',number
   number = 10*number
end do
```

**Output**
**[fio] asterisk:**

```
  1
 10
100
***
***
```

If you find yourself using the same format a number of times, you can give it a *label*:

```
  print 10,"result:",x,y,z
10 format('(a6,3f5.3)')
```

https://www.obliquity.com/computer/fortran/format.html

```
print '( 3i4 )', i1,i2,i3
print '( 3(i2,":",f7.4) )', i1,r1,i2,r2,i3,r2
```

- If `abc` is a format string, then `10(abc)` gives 10 repetitions. There is no line break.
- If there is more data than specified in the format, the format is reused in a new print statement. This causes line breaks.
- The `/` (slash) specifier causes a line break.
- There may be a 80 character limit on output lines.

**Exercise 42.1.** Use formatted I/O to print the number $0 \cdots 99$ as follows:

```
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

## 42.4 File and stream I/O

If you want to send output anywhere else than the terminal screen, you need the `write` statement, which looks like:

```
write (unit,format) data
```

where `format` and `data` are as described above. The new element is the *unit*, which is a numerical indication of an output device, such as a file.

### 42.4.1 Units

For file I/O you write to a unit number, which is associated with a file through an `open` statement.

After you are done with the file, you `Close` it.

```
Open(11)
```

will result in a file with a name typically `fort11`. To give it a name of your choosing:

```
Open(11,FILE="filename")
```

Many other options for error handling, new vs old file, etc.

After this, a `Write` statement can refer to the unit:

```
Write (11,fmt) data
```

Again options for errors and such.

### 42.4.2 Other write options

By default, each `Write` statement, like a `Print` statement, writes to a new line (or 'record' in Fortran terminology). To prevent this,

```
write(unit,fmt,ADVANCE="no") data
```

will not issue a newline.

## 42.5 Conversion to/from string

Above, you saw the commands `Print`, `Write`, and `Read` in the context of output to/from terminal and file. However, there is a second type of use for `Write` and `Read` for string handling, namely conversion between strings and numerical quantities.

#### *42.5.0.1 String to numeric*

To convert a string to numerical quantities, perform a `Read` operation:

```
Read( some_string, some_format ) bunch, of, quantities
```

Example:

**Code:**

```
character(len=8) :: date
integer :: year,month,day
date = "20221027"
read( date,'( i4,i2,i2 )' ) &
     year,month,day
!! ...
print *,"Date:",date
print '( "Year=",i4,", mo=",i2,",
    day=",i2 )', &
     year,month,day
```

**Output**
**[stringf] date:**

```
 Date:20221027
Year=2022, mo=10, day=27
```

#### 42.5.0.2 Numeric to string

Conversely, to construct a string from some quantities you would perform a `Write` operation:

```
Write( some_string, some_format ) bunch, of, quantities
```

**Code:**

```
character(len=10) :: longdate
!! ...
write( longdate,&
     '( i4,"/",i2,"/",i2 )' &
     ) year,month,day
print *,"Long date:",longdate
```

**Output**
**[stringf] slash:**

```
 Long date:2022/10/27
```

## 42.6 Unformatted output

So far we have looked at ASCII output, which is nice to look at for a human , but is not the right medium to communicate data to another program.

- ASCII output requires time-consuming conversion.
- ASCII rendering leads to loss of precision.

Therefore, if you want to output data that is later to be read by a program, it is best to use *binary output* or *unformatted output*, sometimes also called *raw output*.

Indicated by lack of format specification:

```
write (*) data
```

Note: may not be portable between machines.

## 42.7 Print to printer

In Fortran standards before Fortran2003, column 1 of the output had a special meaning, corresponding to *line printer* tractor control. Notoriously, having a character here would move to a new page. While this feature has been removed from the standard, you may still see a black first column in your output, without specifying such.

# Chapter 43

# Leftover topics

## 43.1 Interfaces

If you want to use a procedure in your main program, the compiler needs to know the signature of the procedure: how many arguments, of what type, and with what intent. You have seen how the `contains` clause can be used for this purpose if the procedure resides in the same file as the main program.

If the procedure is in a separate file, the compiler does not see definition and usage in one go. To allowed the compiler to do checking on proper usage, we can use an `interface` block. This is placed at the calling site, declaring the signature of the procedure.

**Main program:**

```
interface
   function f(x,y)
     real*8 :: f
     real*8,intent(in) :: x,y
   end function f
end interface

real*8 :: in1=1.5, in2=2.6, result

result = f(in1,in2)
```

**Procedure:**

```
function f(x,y)
  implicit none
  real*8 :: f
  real*8,intent(in) :: x,y
```

The `interface` block is not required (an older `external` mechanism exists for functions), but is recommended. It is required if the function takes function arguments.

### 43.1.1 Polymorphism

The `interface` block can be used to define a generic function:

```
interface f
function f1( ..... )
function f2( ..... )
end interface f
```

where `f1`,`f2` are functions that can be distinguished by their argument types. The generic function `f` then becomes either `f1` or `f2` depending on what type of argument it is called with.

## 43.2 Random numbers

In this section we briefly discuss the Fortran *random number generator*. The basic mechanism is through the library subroutine `random_number`, which has a single argument of type `REAL` with `INTENT(OUT)`:

```
real(4) :: randomfraction
call random_number(randomfraction)
```

The result is a random number from the uniform distribution on $[0, 1)$.

Setting the *random seed* is slightly convoluted. The amount of storage needed to store the seed can be processor and implementation-dependent, so the routine `random_seed` can have three types of named argument, exactly one of which can be specified at any one time. The keyword can be:

- `SIZE` for querying the size of the seed;
- `PUT` for setting the seed; and
- `GET` for querying the seed.

A typical fragment for setting the seed would be:

```
integer :: seedsize
integer,dimension(:),allocatable :: seed

call random_seed(size=seedsize)
allocate(seed(seedsize))
seed(:) = ! your integer seed here
call random_seed(put=seed)
```

## 43.3 Timing

Timing is done with the `system_clock` routine.

- This call gives an integer, counting clock ticks.
- To convert to seconds, it can also tell you how many ticks per second it has: its *timer resolution*.

```
integer :: clockrate,clock_start,clock_end
call system_clock(count_rate=clockrate)
print *,"Ticks per second:",clockrate

call system_clock(clock_start)
! code
call system_clock(clock_end)
print *,"Time:",(clock_end-clock_start)/REAL(clockrate)
```

## 43.4 Fortran standards

- The first Fortran standard was just called 'Fortran'.
- Fortran4 was a popular next standard.
- Fortran66 was also common. It was very much based on `goto` statements, because there were no block structures.
- Fortran77 was a more structured language, containing the modern `do` and `if` block statements. However, memory management was still completely static and recursive procedures didn't exist.

- Fortran88 didn't happen in time to justify the name, and even calling it Fortran8X didn't help: it became Fortran90. This standard introduced
  - Modules, which made `common` blocks no longer needed.
  - the `implicit none` specification,
  - dynamic memory allocation.
  - recursion.

  Fortran95 was a clarification of Fortran90.
- Fortran2003 (and its refinement Fortran2008) introduced:
  - Object-orientation.
  - This is also when Co-array Fortran (CAF) became part of the language.
- Fortran2018 introduced sub-modules, more parallelism, more C-interoperability.

# Chapter 44

# Fortran review questions

## 44.1 Fortran versus C++

**Exercise 44.1.** For each of C++, Fortran, Python:

- Give an example of an application or application area that the language is suited for, and
- Give an example of an application or application area that the language is not so suited for.

## 44.2 Basics

**Exercise 44.2.**

- What does the `Parameter` keyword do? Give an example where you would use it.
- Why would you use a `Module`?
- What is the use of the `Intent` keyword?

## 44.3 Arrays

**Exercise 44.3.** You are looking at historical temperature data, say a table of the high and low temperature at January 1st of every year between 1920 and now, so that is 100 years.

Your program accepts data as follows:

```
Integer :: year, high, low

!! code omitted

read *,year,high,low
```

where the temperatures are rounded to the closest degree (Centigrade of Fahrenheit is up to you.)

Consider two scenarios. For both, give the lines of code for 1. the array in which you store the data, 2. the statement that inserts the values into the array.

- Store the raw temperature data.

- Suppose you are interested in knowing how often certain high/low temperatures occurred. For instance, 'how many years had a high temperature of 32F /0 C'.

## 44.4 Subprograms

**Exercise 44.4.** Write the missing procedure `pos_input` that

- reads a number from the user
- returns it
- and returns whether the number is positive

in such a way to make this code work:

**Code:**

```
program looppos
  implicit none
  real(4) :: userinput
  do while (pos_input(userinput))
     print &
      '("Positive input:",f7.3)',&
      userinput
  end do
  print &
      '("Nonpositive input:",f7.3)',&
      userinput
  !! ...
end program looppos
```

**Output**

**[funcf] looppos:**

```
Running with the following
    inputs:

5
1
-7.3

/bin/sh: ./looppos: No such
    file or directory
make[2]: *** [run_looppos]
    Error 127
```

Give the function parameter(s) the right `Intent` directive.

Hint: is `pos_input` a `SUBROUTINE` or `FUNCTION`? If the latter, what is the type of the function result? How many parameters does it have otherwise? Where does the variable `user_input` get its value? So what is the type of the parameter(s) of the function?

# PART IV

# EXERCISES AND PROJECTS

# Chapter 45

# Style guide for project submissions

*The purpose of computing is insight, not numbers. (Richard Hamming)*

Your project writeup is equally important as the code. Here are some common-sense guidelines for a good writeup. However, not all parts may apply to your project. Use your good judgment.

## 45.1 General approach

As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Turn in a writeup in pdf form (Word and text documents are not acceptable) that was generated from a text processing program such (preferably) LaTeX (for a tutorial, see Tutorials book [8], section-15).

## 45.2 Style

Your report should obey the rules of proper English.

- Observing correct spelling and grammar goes without saying.
- Use full sentences.
- Try to avoid verbiage that is disparaging or otherwise inadvisable. The *Google developer documentation style guide* [11] is a great resource.

## 45.3 Structure of your writeup

Consider this project writeup an opportunity to practice writing a scientific article.

Start with the obvious stuff.

- Your writeup should have a title. Not 'Project' or 'parallel programming', but something like 'Parallelization of Chronosynclastic Enfundibula in MPI'.
- Author and contact information. This differs per case. Here it is: your name, EID, TACC username, and email.
- Introductory section that is extremely high level: what is the problem, what did you do, what did you find.

- Conclusion: what do your findings mean, what are limitations, opportunities for future extensions.
- Bibliography.

### 45.3.1 Introduction

The reader of your document need not be familiar with the project description, or even the problem it addresses. Indicate what the problem is, give theoretical background if appropriate, possibly sketch a historic background, and describe in global terms how you set out to solve the problem, as well as a brief statement of your findings.

### 45.3.2 Detailed presentation

See section 45.5 below.

### 45.3.3 Discussion and summary

Separate the detailed presentation from a discussion at the end: you need to have a short final section that summarizes your work and findings. You can also discuss possible extensions of your work to cases not covered.

## 45.4 Experiments

You should not expect your program to run once and give you a final answer to your research question.

Ask yourself: what parameters can be varied, and then vary them! This allows you to generate graphs or multi-dimensional plots.

If you vary a parameter, think about what granularity you use. Do ten data points suffice, or do you get insight from using $10,000$?

Above all: computers are very fast, they do a billion operations per second. So don't be shy in using long program runs. Your program is not a calculator where a press on the button immediately gives the answer: you should expect program runs to take seconds, maybe minutes.

## 45.5 Detailed presentation of your work

The detailed presentation of your work is as combination of code fragments, tables, graphs, and a description of these.

### 45.5.1 Presentation of numerical results

You can present results as graphs/diagrams or tables. The choice depends on factors such as how many data points you have, and whether there is an obvious relation to be seen in a graph.

Graphs can be generated any number of ways. Kudos if you can figure out the LaTeX `tikz` package, but Matlab or Excel are acceptable too. No screenshots though.

Number your graphs/tables and refer to the numbering in the text. Give the graph a clear label and label the axes.

### 45.5.2 Code

Your report should describe in a global manner the algorithms you developed, and you should include relevant code snippets. If you want to include full listings, relegate that to an appendix: code snippets in the text should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a monospace font such as the `verbatim` environment, but using the `listings` package (used in this book) is very much recommended.

# Chapter 46

# Prime numbers

In this chapter you will do a number of exercises regarding prime numbers that build on each other. Each section lists the required prerequisites. Conversely, the exercises here are also referenced from the earlier chapters.

## 46.1 Arithmetic

*Before doing this section, make sure you study section 4.5Expressionssection.4.5.*

**Exercise 46.1.** Read two numbers and print out their modulus. The modulus operator is `x%y`.

- Can you also compute the modulus without the operator?
- What do you get for negative inputs, in both cases?
- Assign all your results to a variable before outputting them.

## 46.2 Conditionals

*Before doing this section, make sure you study section 5.1Conditionalssection.5.1.*

**Exercise 46.2.** Read two numbers and print a message stating whether the second is as divisor of the first:

**Code:**

```
int number,divisor;
bool is_a_divisor;
/* ... */
if (
/* ... */
    ) {
  cout << "Indeed, " << divisor
       << " is a divisor of "
       << number << '\n';
} else {
  cout << "No, " << divisor
       << " is not a divisor of "
       << number << '\n';
}
```

**Output**

**[primes] division:**

```
( echo 6 ; echo 2 ) |
    divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) |
    divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

## 46.3 Looping

*Before doing this section, make sure you study section 6.1The 'for' loopsection.6.1.*

**Exercise 46.3.** Read an integer and set a boolean variable to determine whether it is prime by testing for the smaller numbers if they divide that number.

Print a final message

```
Your number is prime
```

or

```
Your number is not prime: it is divisible by ....
```

where you report just one found factor.

Printing a message to the screen is hardly ever the point of a serious program. In the previous exercise, let's therefore assume that the fact of primeness (or non-primeness) of a number will be used in the rest of the program. So you want to store this conclusion.

**Exercise 46.4.** Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

**Exercise 46.5.** Read in an integer $r$. If it is prime, print a message saying so. If it is not prime, find integers $p \leq q$ so that $r = p \cdot q$ and so that $p$ and $q$ are as close together as possible. For instance, for $r = 30$ you should print out `5, 6`, rather than `3, 10`. You are allowed to use the function `sqrt`.

## 46.4 Functions

*Before doing this section, make sure you study section 7Functionschapter.7.*

Above you wrote several lines of code to test whether a number was prime.

**Exercise 46.6.** Write a function `is_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```
int main() {
  bool isprime;
  isprime = is_prime(13);
```

Read the number in, and print the value of the boolean.

Does your function have one or two `return` statements? Can you imagine what the other possibility looks like? Do you have an argument for or against it?

## 46.5 While loops

*Before doing this section, make sure you study section 6.3Looping untilsection.6.3.*

**Exercise 46.7.** Take your prime number testing function `is_prime`, and use it to write a program that prints multiple primes:

- Read an integer `how_many` from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable `number_of_primes_found` that is increased whenever a new prime is found.)

## 46.6 Classes and objects

*Before doing this section, make sure you study section 9.1What is an object?section.9.1.*

**Exercise 46.8.** Write a class `primegenerator` that contains:

- Methods `number_of_primes_found` and `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

In the previous exercise you defined the `primegenerator` class, and you made one object of that class:

```
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other.

**Exercise 46.9.** The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this for the even numbers up to a bound that you read in. Use the `primegenerator` class you developed in exercise 46.8Classes and objectsexcounter.46.8.

This is a great exercise for a top-down approach!

1. Make an outer loop over the even numbers $e$.
2. For each $e$, generate all primes $p$.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that $q$ is prime.

For each even number `e` then print `e,p,q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

An interesting corollary of the Goldbach conjecture is that each prime (start at 5) is equidistant between two other primes.

The *Goldbach conjecture* says that every even number $2n$ (starting at 4), is the sum of two primes $p + q$:

$$2n = p + q.$$

Equivalently, every number $n$ is equidistant from two primes:

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r \text{ prime}} \exists_{p,q \text{ prime}} : r = (p+q)/2 \text{ is prime.}$$

**Exercise 46.10.**

Write a program that tests this. You need at least one loop that tests all primes $r$; for each $r$ you then need to find the primes $p, q$ that are equidistant to it. Do you use two generators for this, or is one enough? Do you need three, for $p, q, r$?

For each $r$ value, when the program finds the $p, q$ values, print the $p, q, r$ triple and move on to the next $r$.

### 46.6.1 Exceptions

*Before doing this section, make sure you study section 23.2.2Exception handling-subsection.23.2.2.*

**Exercise 46.11.** Revisit the prime generator class (exercise 46.8Classes and objectsexcounter.46.8) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 25.4Limitssection.25.4.)

**Code:**

```
try {
  do {
    auto cur = primes.nextprime();
    cout << cur << '\n';
  } while (true);
} catch ( string s ) {
  cout << s << '\n';
}
```

**Output**
**[primes] genx:**

```
9931
9941
9949
9967
9973
Reached max int
```

### 46.6.2 Prime number decomposition

*Before doing this section, make sure you study section 24.2.1Maps: associative arrayssubsection.24.2.1.*

Design a class `Integer` which stores its value as its prime number decomposition. For instance,

$$180 = 2^2 \cdot 3^3 \cdot 5 \quad \Rightarrow \quad \texttt{[ 2:2, 3:2, 5:1 ]}$$

You can implement this decomposition itself as a *vector*, (the $i$-th location stores the exponent of the $i$-th prime) but let's use a `map` instead.

**Exercise 46.12.** Write a constructor of an `Integer` from an `int`, and methods `as_int` / `as_string` that convert the decomposition back to something classical. Start by assuming that each prime factor appears only once.

**Code:**

```
Integer i2(2);
cout << i2.as_string() << ": "
     << i2.as_int() << '\n';

Integer i6(6);
cout << i6.as_string() << ": "
     << i6.as_int() << '\n';
```

**Output**
**[primes] decomposition26:**

```
2^1 : 2
2^1 3^1 : 6
```

**Exercise 46.13.** Extend the previous exercise to having multiplicity $> 1$ for the prime factors.

**Code:**

```
Integer i180(180);
cout << i180.as_string() << ": "
     << i180.as_int() << '\n';
```

**Output**
**[primes] decomposition180:**

```
2^2 3^2 5^1 : 180
```

Implement addition and multiplication for `Integer`s.

Implement a class `Rational` for rational numbers, which are implemented as two `Integer` objects. This class should have methods for addition and multiplication. Write these through operator overloading if you've learned this.

Make sure you always divide out common factors in the numerator and denominator.

## 46.7 Other

The following exercise requires `std::optional`, which you can learn about in section 24.5.2Optional-subsection.24.5.2.

**Exercise 46.14.** Write a function `first_factor` that optionally returns the smallest factor of a given input.

```
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << '\n';
```

## 46.8 Eratosthenes sieve

The Eratosthenes sieve is an algorithm for prime numbers that step-by-step filters out the multiples of any prime it finds.

1. Start with the integers from 2: $2, 3, 4, 5, 6, \ldots$
2. The first number, 2, is a prime: record it and remove all multiples, giving

   $$3, 5, 7, 9.11, 13, 15, 17 \ldots$$

3. The first remaining number, 3, is a prime: record it and remove all multiples, giving

   $$5, 7, 11, 13, 17, 19, 23, 25, 29 \ldots$$

4. The first remaining number, 5, is a prime: record it and remove all multiples, giving

   $$7, 11, 13, 17, 19, 23, 29, \ldots$$

### 46.8.1 Arrays implementation

The sieve can be implemented with an array that stores all integers.

**Exercise 46.15.** Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers. Apply the sieve algorithm to find the prime numbers.

### 46.8.2 Streams implementation

The disadvantage of using an array is that we need to allocate an array. What's more, the size is determined by how many integers we are going to test, not how many prime numbers we want to generate. We are going to take the idea above of having a generator object, and apply that to the sieve algorithm: we will now have multiple generator objects, each taking the previous as input and erasing certain multiples from it.

**Exercise 46.16.** Write a `stream` class that generates integers and use it through a pointer.

**Code:**

```
for (int i=0; i<7; i++)
  cout << "Next int: "
       << the_ints->next() << '\n';
```

**Output**
**[sieve] ints:**

```
Next int: 2
Next int: 3
Next int: 4
Next int: 5
Next int: 6
Next int: 7
Next int: 8
```

Next, we need a stream that takes another stream as input, and filters out values from it.

**Exercise 46.17.** Write a class `filtered_stream` with a constructor

```
filtered_stream(int filter,shared_ptr<stream> input);
```

that

1. Implements `next`, giving filtered values,
2. by calling the `next` method of the input stream and filtering out values.

**Code:**

```
auto integers =
  make_shared<stream>();
auto odds =
  shared_ptr<stream>
  ( new filtered_stream(2,integers) );
for (int step=0; step<5; step++)
  cout << "next odd: "
       << odds->next() << '\n';
```

**Output**
**[sieve] odds:**

```
next odd: 3
next odd: 5
next odd: 7
next odd: 9
next odd: 11
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

**Exercise 46.18.** Write a program that generates prime numbers as follows.

- Maintain a `current` stream, that is initially the stream of prime numbers.
- Repeatedly:
  - Record the first item from the current stream, which is a new prime number;
  - and set `current` to a new stream that takes `current` as input, filtering out multiples of the prime number just found.

## 46.9 Range implementation

*Before doing this section, make sure you study section 14.1.5Iterating over classes-subsection.14.1.5.*

**Exercise 46.19.** Make a `primes` class that can be ranged:

**Code:**

```
primegenerator allprimes;
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout << '\n';
```

**Output**

```
[primes] range:
2, 3, 5, 7, 11, 13, 17, 19, 23,
    29, 31, 37, 41, 43, 47, 53,
    59, 61, 67, 71, 73, 79, 83,
    89, 97, 101,
```

## 46.10 User-friendliness

Use the `cxxopts` package (section 63.2Options processing: `cxxopts`section.63.2) to add commandline options to some primality programs.

**Exercise 46.20.** Take your old prime number testing program, and add commandline options:

- the `-h` option should print out usage information;
- specifying a single int `--test 1001` should print out all primes under that number;
- specifying a set of ints `--tests 57,125,1001` should test primeness for those.

# Chapter 47

# Geometry

In this set of exercises you will write a small 'geometry' package: code that manipulates points, lines, shapes. These exercises mostly use the material of section 9Classes and objectschapter.9.

## 47.1 Basic functions

**Exercise 47.1.** Write a function with (float or double) inputs $x, y$ that returns the distance of point $(x, y)$ to the origin.

Test the following pairs: $1, 0$; $0, 1$; $1, 1$; $3, 4$.

**Exercise 47.2.** Write a function with inputs $x, y, \theta$ that alters $x$ and $y$ corresponding to rotating the point $(x, y)$ over an angle $\theta$.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should behave like:

**Code:**

```
const float pi = 2*acos(0.0);
float x{1.}, y{0.};
rotate(x,y,pi/4);
cout << "Rotated halfway: ("
     << x << "," << y << ")" << '\n';
rotate(x,y,pi/4);
cout << "Rotated to the y-axis: ("
     << x << "," << y << ")" << '\n';
```

**Output**

```
[geom] rotate:

Rotated halfway:
    (0.707107,0.707107)
Rotated to the y-axis: (0,1)
```

## 47.2 Point class

*Before doing this section, make sure you study section 9.1What is an object?section.9.1.*

A class can contain elementary data. In this section you will make a `Point` class that models Cartesian coordinates and functions defined on coordinates.

**Exercise 47.3.** Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `angle` computes the angle of vector $(x, y)$ with the $x$-axis.

**Exercise 47.4.** Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

```
p.distance(q)
```

computes the distance between them.

**Exercise 47.5.** Write a method *halfway* that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p + q)/2$:

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions `Add` and `Scale` and combine these. (Later you will learn about operator overloading.)

How would you print out a *Point* to make sure you compute the halfway point correctly?

**Exercise 47.6.** Make a default constructor for the point class:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

but which gives an indication that it is undefined:

**Code:**

```
Point p3;
cout << "Uninitialized point:"
     << '\n';
p3.printout();
cout << "Using uninitialized point:"
     << '\n';
auto p4 = Point(4,5)+p3;
p4.printout();
```

**Output**
**[geom] linearnan:**

```
Uninitialized point:
Point: nan,nan
Using uninitialized point:
Point: nan,nan
```

Hint: see section 25.4.1Not-a-numbersubsection.25.4.1.

**Exercise 47.7.** Revisit exercise 47.2Basic functionsexcounter.47.2 using the `Point` class. Your code should now look like:

```
newpoint = point.rotate(alpha);
```

**Exercise 47.8.** Advanced. Can you make a `Point` class that can accommodate any number of space dimensions? Hint: use a `vector`; section 10.3Vector are a classsection.10.3. Can you make a constructor where you do not specify the space dimension explicitly?

## 47.3 Using one class in another

*Before doing this section, make sure you study section 9.2Inclusion relations between classessection.9.2.*

**Exercise 47.9.** Make a class `LinearFunction` with a constructor:

```
LinearFunction( Point input_p1,Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

**Exercise 47.10.** Make a class `LinearFunction` with two constructors:

```
LinearFunction( Point input_p2 );
LinearFunction( Point input_p1,Point input_p2 );
```

where the first stands for a line through the origin.
Implement again the `evaluate` function so that

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

**Exercise 47.11.** Revisit exercises 47.2Basic functionsexcounter.47.2 and 47.7Point classexcounter.47.7, introducing a `Matrix` class. Your code can now look like

```
newpoint = point.apply(rotation_matrix);
```

or

```
newpoint = rotation_matrix.apply(point);
```

Can you argue in favor of either one?

Suppose you want to write a `Rectangle` class, which could have methods such as `float Rectangle::area()` or `bool Rectangle::contains(Point)`. Since rectangle has four

corners, you could store four `Point` objects in each `Rectangle` object. However, there is redundancy there: you only need three points to infer the fourth. Let's consider the case of a rectangle with sides that are horizontal and vertical; then you need only two points.

Intended API:

```
float Rectangle::area();
```

It would be convenient to store width and height; for

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

**Exercise 47.12.**

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   Rectangle(Point botleft,float width,float height);
   ```

   The logical implementation is to store these quantities. Implement methods:

   ```
   float area(); float rightedge_x(); float topedge_y();
   ```

   and write a main program to test these.
2. Add a second constructor

   ```
   Rectangle(Point botleft,Point topright);
   ```

   Can you figure out how to use *member initializer* lists for the constructors?

**Exercise 47.13.** Make a copy of your solution of the previous exercise, and redesign your class so that it stores two `Point` objects. Your main program should not change.

The previous exercise illustrates an important point: for well designed classes you can change the implementation (for instance motivated by efficiency) while the program that uses the class does not change.

## 47.4 Is-a relationship

*Before doing this section, make sure you study section 9.3Inheritancesection.9.3.*

**Exercise 47.14.** Take your code where a `Rectangle` was defined from one point, width, and height.

Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.

First ask yourself: what should the constructor of a `Square` look like?

**Exercise 47.15.** Revisit the `LinearFunction` class. Add methods `slope` and `intercept`.

Now generalize `LinearFunction` to `StraightLine` class. These two are almost the same except for vertical lines. The `slope` and `intercept` do not apply to vertical lines, so design `StraightLine` so that it stores the defining points internally. Let `LinearFunction` inherit.

## 47.5 Pointers

*Before doing this section, make sure you study section 16.2Making a shared pointersection.16.2.*

The following exercise is a little artificial.

**Exercise 47.16.** Make a `DynRectangle` class, which is constructed from two shared-pointers-to-`Point` objects:

```
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

Calculate the area, scale the top-right point, and recalculate the area:

**Code:**

```
cout << "Area: " <<  lielow.area() <<
    '\n';
/* ... */
// scale the 'fivetwo' point by two
cout << "Area: " <<  lielow.area() <<
    '\n';
```

**Output**

```
[pointer] dynrect:
Area: 10
Area: 40
```

*You can base this off the file `pointrectangle.cxx` in the repository*

## 47.6 More stuff

*Before doing this section, make sure you study section 15.3Reference to class memberssection.15.3.*

The `Rectangle` class stores at most one corner, but may be convenient to sometimes have an array of all four corners.

**Exercise 47.17.** Add a method

```
const vector<Point> &corners()
```

to the `Rectangle` class. The result is an array of all four corners, not in any order. Show by a compiler error that the array can not be altered.

*Before doing this section, make sure you study section 9.5.6Operator overloadingsubsection.9.5.6.*

**Exercise 47.18.** Revisit exercise 47.5 and replace the `add` and `scale` functions by overloaded operators.

Hint: for the `add` function you may need '`this`'.

# Chapter 48

# Zero finding

## 48.1 Root finding by bisection

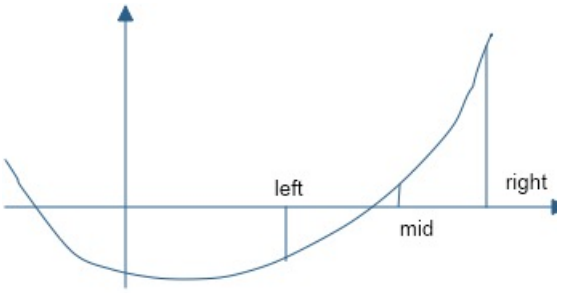



Figure 48.1: Root finding by interval bisection

For many functions $f$, finding their zeros, that is, the values $x$ for which $f(x) = 0$, can not be done analytically. You then have to resort to numerical *root finding* schemes. In this project you will develop gradually more complicated implementations of a simple scheme: root finding by *bisection*.

In this scheme, you start with two points where the function has opposite signs, and move either the left or right point to the mid point, depending on what sign the function has there. See figure 48.1.

In section 48.2 we will then look at Newton's method.

Here we will not be interested in mathematical differences between the methods, though these are important: we will use these methods to exercise some programming techniques.

### 48.1.1 Simple implementation

*Before doing this section, make sure you study section chapter 7 about Functions, and chapter 10 about Vectors.*

Let's develop a first implementation step by step. To ensure correctness of our code we will use a Test-Driven Development (TDD) approach: for each bit of functionality we write a test to ensure its correctness before we integrate it in the larger code. (For more about TDD, and in particular the Catch2 framework, see section 68.2.)

### 48.1.2 Polynomials

First of all, we need to have a way to represent polynomials. For a polynomial of degree $d$ we need $d+1$ coefficients:

$$f(x) = c_0 x^d + \cdots + c_{d-1} x^1 + c_d \tag{48.1}$$

We implement this by storing the coefficients in a `vector<double>`. We make the following arbitrary decisions

1. let the first element of this vector be the coefficient of the highest power, and
2. for the coefficients to properly define a polynomial, this leading coefficient has to be nonzero.

Let's start by having a fixed test polynomial, provided by a function `set_coefficients`. For this function to provide a proper polynomial, it has to satisfy the following test:

```
TEST_CASE( "coefficients represent polynomial" "[1]") {
  vector<double> coefficients = { 1.5, 0., -3 };
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

**Exercise 48.1.** Write a routine `set_coefficients` that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Above we postulated two conditions that an array of numbers should satisfy to qualify as the coefficients of a polynomial. Your code will probably be testing for this, so let's introduce a boolean function `is_proper_polynomial`:

- This function returns `true` if the array of numbers satisfies the two conditions;
- it returns `false` if either condition is not satisfied.

In order to test your function `is_proper_polynomial` you should check that

- it recognizes correct polynomials, and
- it fails for improper coefficients that do not properly define a polynomial.

**Exercise 48.2.** Write a function `is_proper_polynomial` as described, and write unit tests for it, both passing and failing:

```
  vector<double> good = /* proper coefficients */ ;
  REQUIRE( is_proper_polynomial(good) );
  vector<double> notso = /* improper coefficients */ ;
  REQUIRE( not is_proper_polynomial(good) );
```

Next we need polynomial evaluation. We will build a function `evaluate_at` with the following definition:

```
double evaluate_at( const std::vector<double>& coefficients,double x);
```

You can interpret the array of coefficients in (at least) two ways, but with equation (48.1) we proscribed one particular interpretation.

So we need a test that the coefficients are indeed interpreted with the leading coefficient first, and not with the leading coefficient last. For instance:

```
polynomial second( {2,0,1} );
// correct interpretation: 2x^2 + 1
REQUIRE( second.is_proper() );
REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

(where we have left out the `TEST_CASE` header.)

Now we write the function that passes these tests:

**Exercise 48.3.** Write a function `evaluate_at` which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up *Horner's rule* and implement it.

With the polynomial function implemented, we can start working towards the algorithm.

### 48.1.3 Left/right search points

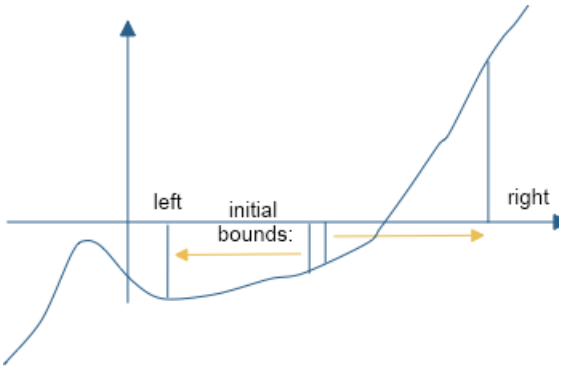



Figure 48.2: Setting the initial search points

Suppose $x_-, x_+$ are such that

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values in the left and right point are of opposite sign. Then there is a zero in the interval $(x_-, x_+)$; see figure 48.1.

But how to find these outer bounds on the search?

If the polynomial is of odd degree you can find $x_-, x_+$ by going far enough to the left and right from any two starting points. For even degree there is no such simple algorithm (indeed, there may not be a zero) so we abandon the attempt.

We start by writing a function `is_odd` that tests whether the polynomial is of odd degree.

**Exercise 48.4.** Make the following code work:

```
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

You could test the above as:

```
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Now we can find $x_-, x_+$: start with some interval and move the end points out until the function values have opposite sign.

**Exercise 48.5.** Write a function `find_initial_bounds` which computes $x_-, x_+$ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

How can you compute this test more compactly?
What is a good prototype for the function?
How do move the points far enough out to satisfy this condition?

Since finding a left and right point with a zero in between is not always possible for polynomials of even degree, we completely reject this case. In the following test we throw an exception (see section 23.2.2, in particularly 23.2.2.3) for polynomials of even degree:

```
right = left+1;
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second,left,right) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
REQUIRE( left<right );
```

Make sure your code passes these tests. What test do you need to add for the function values?

### 48.1.4 Root finding

The root finding process globally looks as follows:

- You start with points $x_-, x_+$ where the function has opposite sign; then you know that there is a zero between them.
- The bisection method for finding this zero looks at the halfway point, and based on the function value in the mid point:

- moves one of the bounds to the mid point, such that the function again has opposite signs in the left and right search point.

The structure of the code is as follows:

```
double find_zero( /* something */ ) {
  while ( /* left and right too far apart */ ) {
    // move bounds left and right closer together
  }
  return something;
}
```

Again, we test all the functionality separately. In this case this means that moving the bounds should be a testable step.

**Exercise 48.6.** Write a function `move_bounds_closer` and test it.

```
void move_bounds_closer
    ( std::vector<double> coefficients,
      double& left,double& right );
```

Implement some unit tests on this function.

Finally, we put everything together in the top level function `find_zero`.

**Exercise 48.7.** Make this call work:

```
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

### 48.1.5 Object implementation

Revisit the exercises of section 48.1.1 and introduce a `polynomial` class that stores the polynomial coefficients. Several functions now become members of this class.

Also update the unit tests.

How can you generalize the polynomial class, for instance to the case of special forms such as $(1+x)^n$?

### 48.1.6 Templating

In the implementations so far we used `double` for the numerical type. Make a templated version that works both with `float` and `double`.

Can you see a difference in attainable precision between the two types?

## 48.2 Newton's method

*Before doing this section, make sure you study section lambda functions; chapter 13.*

In this section we look at Newton's method. This is an iterative method for finding zeros of a function $f$, that is, it computes a sequence of values $\{x_n\}_n$, so that $f(x_n) \to 0$. The sequence is defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

with $x_0$ arbitrarily chosen.

As a specific case, here we use

$$f(x) = x^2 - n, \qquad f'(x) = 2x$$

which has the effect that, if we find an $x$ such that $f(x) = 0$, we have

$$x = \sqrt{n}.$$

It is of course simple to code this specific case; it should take you about 10 lines. However, we want to have a general code that takes any two functions $f, f'$, and then uses Newton's method to find a zero of $f$.

### 48.2.1 Function implementation

Early computers had no hardware for computing a square root. Instead, they used *Newton's method*. Suppose you have a value $y$ and you want want to compute $x = \sqrt{y}$. This is equivalent to finding the zero of

$$f(x) = x^2 - y$$

where $y$ is fixed. To indicate this dependence on $y$, we will write $f_y(x)$. Newton's method then finds the zero by evaluating

$$x_{\text{next}} = x - f_y(x)/f_y'(x)$$

until the guess is accurate enough, that is, until $f_y(x) \approx 0$.

**Exercise 48.8.**

- Write functions `f(x,y)` and `deriv(x,y)`, that compute $f_y(x)$ and $f_y'(x)$ for the definition of $f_y$ above.
- Read a value $y$ and iterate until $|\texttt{f}(\texttt{x},\texttt{y})| < 10^{-5}$. Print $x$.
- Second part: write a function `newton_root` that computes $\sqrt{y}$.

### 48.2.2 Using lambdas

**Exercise 48.9.** The Newton method (HPC book [9], section 22) for finding the zero of a function $f$, that is, finding the $x$ for which $f(x) = 0$, can be programmed by supplying the function and its derivative:

```
double f(double x) { return x*x-2; };
double fprime(double x) { return 2*x; };
```

and the algorithm:

```
double x{1.};
while ( true ) {
  auto fx = f(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime(x);
}
```

Rewrite this code to use lambda functions for `f` and `fprime`.

*You can base this off the file `newton.cxx` in the repository*

Next, we make the code modular by writing a general function `newton_root`, that contains the Newton method of the previous exercise. Since it has to work for any functions $f, f'$, you have to pass the objective function and the derivative as arguments:

```
double root = newton_root( f,fprime );
```

**Exercise 48.10.** Rewrite the Newton exercise above to use a function that is used as:

```
double root = newton_root( f,fprime );
```

Call the function

1. first with the lambda variables you already created;
2. but in a better variant, directly with the lambda expressions as arguments, that is, without assigning them to variables.

Next we extend functionality, but not by changing the root finding function: instead, we use a more general way of specifying the objective function and derivative.

**Exercise 48.11.** Extend the newton exercise to compute roots in a loop:

```
for (int n=2; n<=8; n++) {
  cout << "sqrt(" << n << ") = "
       << newton_root(
/* ... */
                      )
       << '\n';
```

Without lambdas, you would define a function

```
double squared_minus_n( double x,int n ) {
  return x*x-n; }
```

However, the `newton_root` function takes a function of only a real argument. Use a capture to make `f` dependent on the integer parameter.

**Exercise 48.12.** You don't need the gradient as an explicit function: you can approximate it as

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

for some value of $h$.

Write a version of the root finding function

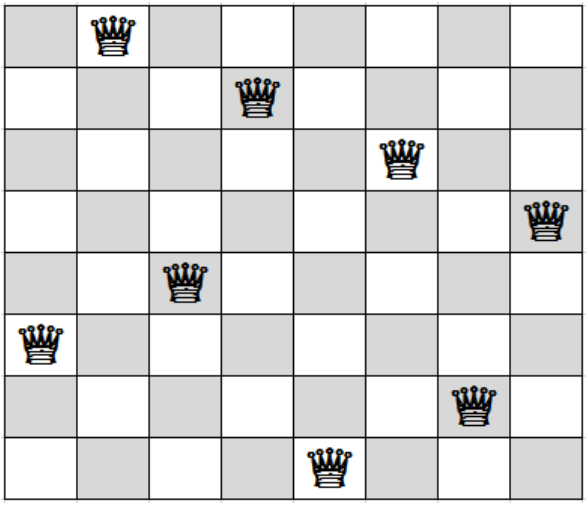
```
double newton_root( function< double(double)> f )
```

that uses this. You can use a fixed value `h=1e-6`. Do not reimplement the whole newton method: instead create a lambda for the gradient and pass it to the function `newton_root` you coded earlier.

# Chapter 49

# Eight queens

A famous exercise for recursive programming is the *eight queens* problem: Is it possible to position eight queens on a chess board, so that no two queens 'threaten' each other, according to the rules of chess?



## 49.1 Problem statement

The precise statement of the 'eight queens problem' is:

- Put eight pieces on an $8 \times 8$ board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.

A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

**Exercise 49.1.** This algorithm will generate all $8^8$ boards. Do you see at least one way to speed up the search?

Since the number eight is, for now, fixed, you could write this code as an eight-deep loop nest. However that is not elegant. For example, the only reason for the number 8 in the above exposition is that this is the traditional size of a chess board. The problem, stated more abstractly as placing $n$ queens on an $n \times n$ board, has solutions for $n \geq 4$.

## 49.2 Solving the eight queens problem, basic approach

This problem requires you to know about arrays/vectors; chapter 10. Also, see chapter 68 for TDD. Finally, see section 24.5.2 for `std::optional`.

The basic strategy will be that we fill consecutive rows, by indicating each time which column will be occupied by the next queen. Using an Object-Oriented (OO) strategy, we make a class `ChessBoard`, that holds a partially filled board.

The basic solution strategy is recursive:

- Let `current` be the current, partially filled board;
- We call `current.place_queens()` that tries to finish the board;
- However, with recursion, this method only fills one row, and then calls `place_queens` on this new board.

So

```
ChessBoard::place_queens() {
  // for c = 1 ... number of columns:
  //   make a copy of the board
  //   put a queen in the next row, column c, of the copy
  //   and call place_queens() on that copy;
  //   investigate the result.....
}
```

This routine returns either a solution, or an indication that no solution was possible.

In the next section we will develop a solution systematically in a TDD manner.

## 49.3 Developing a solution by TDD

We now gradually develop the OO solution to the eight queens problem, using test-driven development.

***The board*** We start by constructing a board, with a constructor that only indicates the size of the problem:

```
ChessBoard(int n);
```

This is a 'generalized chess board' of size $n \times n$, and initially it should be empty.

**Exercise 49.2.** Write this constructor, for an empty board of size $n \times n$.

Note that the implementation of the board is totally up to you. In the following you will get tests for functionality that you need to satisfy, but any implementation that makes this true is a correct solution.

***Bookkeeping: what's the next row?*** Assuming that we fill in the board row-by-row, we have an auxiliary function that returns the next row to be filled:

```
int next_row_to_be_filled()
```

This gives us our first simple test: on an empty board, the row to be filled in is row zero.

**Exercise 49.3.** Write this method and make sure that it passes the test for an empty board.

```
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

By the rules of TDD you can actually write the method so that it only satisfies the test for the empty board. Later, we will test that this method gives the right result after we have filled in a couple of rows, and then of course your implementation needs to be general.

***Place one queen*** Next, we have a function to place the next queen, whether this gives a feasible board (meaning that no pieces can capture each other) or not:

```
void place_next_queen_at_column(int i);
```

This method should first of all catch incorrect indexing: we assume that the placement routine throws an exception for invalid column numbers.

```
ChessBoard::place_next_queen_at_column( int c ) {
  if ( /* c is outside the board */ )
    throw(1); // or some other exception.
```

(Suppose you didn't test for incorrect indexing. Can you construct a simple 'cheating' solution at any size?)

**Exercise 49.4.** Write this method, and make sure it passes the following test for valid and invalid column numbers:

```
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

(From now on we'll only give the body of the test.)

Now it's time to start writing some serious stuff.

***Is a (partial) board feasible?*** If you have a board, even partial, you want to test if it's feasible, meaning that the queens that have been placed can not capture each other.

The prototype of this method is:

```
bool feasible()
```

This test has to work for simple cases to begin with: an empty board is feasible, as is a board with only one piece.

```
ChessBoard empty(n);
REQUIRE( empty.feasible() );
```

```
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

**Exercise 49.5.** Write the method and make sure it passes these tests.

We shouldn't only do successful tests, sometimes referred to as the 'happy path' through the code. For instance, if we put two queens in the same column, the test should fail.

**Exercise 49.6.** Take the above initial attempt with a queen in position $(0, 0)$, and add another queen in column zero of the next row. Check that it passes the test:

```
ChessBoard collide = one;
// place a queen in a 'colliding' location
collide.place_next_queen_at_column(0);
// and test that this is not feasible
REQUIRE( not collide.feasible() );
```

Add a few tests more of your own. (These will not be exercised by the submission script, but you may find them useful anyway.)

***Testing configurations*** If we want to test the feasibility of non-trivial configurations, it is a good idea to be able to 'create' solutions. For this we need a second type of constructor where we construct a fully filled chess board from the locations of the pieces.

```
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

- If the constructor is called with only a vector, this describes a full board.
- Adding an integer parameter indicates the size of the board, and the vector describes only the rows that have been filled in.

**Exercise 49.7.** Write these constructors, and test that an explicitly given solution is a feasible board:

```
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

For an elegant approach to implementing this, see *delegating constructors*; section 9.4.1.

Ultimately we have to write the tricky stuff.

## 49.4 The recursive solution method

The main function

```
optional<ChessBoard> place_queens()
```

takes a board, empty or not, and tries to fill the remaining rows.

One problem is that this method needs to be able to communicate that, given some initial configuration, no solution is possible. For this, we let the return type of `place_queens` be `optional<ChessBoard>`:

- if it is possible to finish the current board resulting in a solution, we return that filled board;
- otherwise we return `{}`, indicating that no solution was possible.

With the recursive strategy discussed in section 49.2, this placement method has roughly the following structure:

```
place_queens() {
  for ( int col=0; col<n; col++ ) {
    ChessBoard next = *this;
    // put a queen in column col on the 'next' board
    // if this is feasible and full, we have a solution
    // if it is feasible but no full, recurse
  }
}
```

The line

```
ChessBoard next = *this;
```

makes a copy of the object you're in.

**Remark 25** *Another approach would be to make a recursive function*

```
bool place_queen( const ChessBoard& current, ChessBoard &next );
// true if possible, false is not
```

***The final step*** Above you coded the method `feasible` that tested whether a board is still a candidate for a solution. Since this routine works for any partially filled board, you also need a method to test if you're done.

**Exercise 49.8.** Write a method

```
bool filled();
```

and write a test for it, both positive and negative.

Now that you can recognize solutions, it's time to write the solution routine.

**Exercise 49.9.** Write the method

```
optional<ChessBoard> place_queens()
```

Because the function `place_queens` is recursive, it is a little hard to test in its entirety.

We start with a simpler test: if you almost have the solution, it can do the last step.

**Exercise 49.10.** Use the constructor

```
ChessBoard( int n,vector<int> cols )
```

to generate a board that has all but the last row filled in, and that is still feasible. Test that you can find the solution:

```
ChessBoard almost( 4, {1,3,0} );
```

```
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Since this test only fills in the last row, it only does one loop, so printing out diagnostics is possible, without getting overwhelmed in tons of output.

***Solutions and non-solutions*** Now that you have the solution routine, test that it works starting from an empty board. For instance, confirm there are no $3 \times 3$ solutions:

```
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

On the other hand, $4 \times 4$ solutions do exist:

```
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```

**Exercise 49.11.** (Optional) Can you modify your code so that it counts all the possible solutions?

**Exercise 49.12.** (Optional) How does the time to solution behave as function of $n$?

# Chapter 50

# Infectious disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

## 50.1 Model design

It is possible to model disease propagation statistically, but here we will build an explicit simulation: we will maintain an explicit description of all the people in the population, and track for each of them their status.

We will use a simple model where a person can be:

- sick: when they are sick, they can infect other people;
- susceptible: they are healthy, but can be infected;
- recovered: they have been sick, but no longer carry the disease, and can not be infected for a second time;
- vaccinated: they are healthy, do not carry the disease, and can not be infected.

In more complicated models a person could be infectious during only part of their illness, or there could be secondary infections with other diseases, et cetera. We keep it simple here: any sick person is can infect others while they are sick.

In the exercises below we will gradually develop a somewhat realistic model of how the disease spreads from an infectious person. We always start with just one person infected. The program will then track the population from day to day, running indefinitely until none of the population is sick. Since there is no re-infection, the run will always end.

### 50.1.1 Other ways of modeling

Instead of capturing every single person in code, a 'contact network' model, it is possible to use an ODE approach to disease modeling. You would then model the percentage of infected persons by a single scalar, and derive relations for that and other scalars [2].

http://mathworld.wolfram.com/Kermack-McKendrickModel.html

This is known as a 'compartmental model', where each of the three SIR states is a compartment: a section of the population. Both the contact network and the compartmental model capture part of the truth. In

fact, they can be combined. We can consider a country as a set of cities, where people travel between any pair of cities. We then use a compartmental model inside a city, and a contact network between cities.

In this project we will only use the network model.

## 50.2 Coding

### 50.2.1 The basics

We start by writing code that models a single person. The main methods serve to infect a person, and to track their state. We need to have some methods for inspecting that state.

The intended output looks something like:

```
On day 10, Joe is susceptible
On day 11, Joe is susceptible
On day 12, Joe is susceptible
On day 13, Joe is susceptible
On day 14, Joe is sick (5 to go)
On day 15, Joe is sick (4 to go)
On day 16, Joe is sick (3 to go)
On day 17, Joe is sick (2 to go)
On day 18, Joe is sick (1 to go)
On day 19, Joe is recovered
```

**Exercise 50.1.** Write a `Person` class with methods:

- `status_string()` : returns a description of the person's state as a `string`;
- `update()` : update the person's status to the next day;
- `infect(n)` : infect a person, with the disease to run for $n$ days;
- `is_stable()` : return a `bool` indicating whether the person has been sick and is recovered.

Your main program could for instance look like:

```
Person joe;

int step = 1;
for ( ; ; step++) {

  joe.update();
  float bad_luck = (float) rand()/(float)RAND_MAX;
  if (bad_luck>.95)
    joe.infect(5);

  cout << "On day " << step << ", Joe is "
       << joe.status_string() << '\n';
  if (joe.is_stable())
    break;
}
```

Here is a suggestion how you can model disease status. Use a single integer with the following interpretation:

- healthy but not vaccinated, value $0$,
- recovered, value $-1$,
- vaccinated, value $-2$,
- and sick, with $n$ days to go before recovery, modeled by value $n$.

The `Person::update` method then updates this integer.

**Remark 26** *Consider a point of programming style. Now that you've modeled the state of a person with an integer, you can use that as*

```
void infect(n) {
    if (state==0)
        state = n;
}
```

*But you can also write*

```
bool is_susceptible() {
  return state==0;
}
void infect(n) {
    if (is_susceptible())
        state = n;
}
```

*Which do you prefer and why?*

### 50.2.2 Population

Next we need a `Population` class. Implement a population as a `vector` consisting of `Person` objects. Initially we only infect one person, and there is no transmission of the disease.

The trace output should look something like:

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

**Remark 27** *Such a display is good for a sanity check on your program behavior. If you include such displays in your writeup, make sure to use a monospace font, and don't use a population size that needs line wrapping. In further testing, you should use large populations, but do not include these displays.*

**Exercise 50.2.** Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:

```
Population population(npeople);
```

- Write a method that infects a random person:

  ```
  population.random_infection();
  ```

  For the 'random' part you can use the C language random number generator (section 24.6.4C random functionsubsection.24.6.4), or the new STL one in section 24.6Random numberssection.24.6.
- Write a method `count_infected` that counts how many people are infected.
- Write an `update` method that updates all persons in the population.
- Loop the `update` method until no people are infected: the `Population::update` method should apply `Person::update` to all person in the population.

Write a routine that displays the state of the popular, using for instance: `?` for susceptible, `+` for infected, `-` for recovered.

### 50.2.3 Contagion

This past exercise was too simplistic: the original patient zero was the only one who ever got sick. Now let's incorporate contagion, and investigate the spread of the disease from a single infected person.

We start with a very simple model of infection.

**Exercise 50.3.** Choose a number $0 \leq p \leq 1$ representing the probability of disease transmission upon contact.

```
population.set_probability_of_transfer(probability);
```

Incorporate this into the program: in each step the direct neighbors of an infected person can now get sick themselves.

1. To convince yourself of the correctness of the code, first try the cases $p = 0$ and $p = 1$.
2. Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

**Exercise 50.4.** Incorporate vaccination: read another number representing the percentage of people that has been vaccinated. Choose those members of the population randomly.

Describe the effect of vaccinated people on the spread of the disease. Why is this model unrealistic?

### 50.2.4 Spreading

To make the simulation more realistic, we let every sick person come into contact with a fixed number of random people every day. This gives us more or less the *SIR model*; https://en.wikipedia.org/wiki/Epidemic_model.

Set the number of people that a person comes into contact with, per day, to 6 or so. (You can also let this be an upper bound for a random value, but that does not essentially change the simulation.) You have already programmed the probability that a person who comes in contact with an infected person gets sick themselves. Again start the simulation with a single infected person.

**Exercise 50.5.** Code the random interactions. Now run a number of simulations varying

- The percentage of people vaccinated, and
- the chance the disease is transmitted on contact.

Record how long the disease runs through the population. With a fixed number of contacts and probability of transmission, how is this number of function of the percentage that is vaccinated?

Report this function as a table or graph. Make sure you have enough data points for a meaningful conclusion. Use a realistic population size. You can also do multiple runs and report the average, to even out the effect of the random number generator.

**Exercise 50.6.** Investigate the matter of 'herd immunity': if enough people are vaccinated, then some people who are not vaccinated will still never get sick. Let's say you want to have the probability of being not vaccinated, yet never getting sick, to be over 95 percent. Investigate the percentage of vaccination that is needed for this as a function of the contagiousness of the disease.

As in the previous exercise, make sure your data set is large enough.

**Remark 28** *The screen output you used above is good for sanity checks on small problems. However, for realistic simulations you have to think what is a realistic population size. If your university campus is a population where random people are likely to meet each other, what would be a population size to model that? How about the city where you live?*

*Likewise, if you test different vaccination rates, what granularity do you use? With increases of 5 or 10 percent you can print all results to you screen, but you may miss things. Don't be afraid to generate large amount of data and feed them directly to a graphing program.*

### 50.2.5 Mutation

The Covid years have shown how important mutations of an original virus can be. Next, you can include mutation in your project. We model this as follows:

- Every so many transmissions, a virus will mutate into a new variant.
- A person who has recovered from one variant is still susceptible to other variants.
- For simplicity assume that each variant leaves a person sick the same number of days, and
- Vaccination is all-or-nothing: one vaccine is enough to protect against all variant;
- On the other hand, having recovered from one variant is not protection against others.

Implementation-wise speaking, we model this as follows. First of all, we need a `Disease` class, so that we can infect a person with an explicit virus;

```
void infect(int);
void infect(Disease);
```

A `Disease` object now carries the information such as the chance of transmission, or how a long a person stays under the weather. Modeling mutation is a little tricky. You could do it as follows:

- There is a global `variants` counter for new virus variants, and a global `transmissions` counter.
- Every time a person infects another, the newly infected person gets a new `Disease` object, with the current variant, and the transmissions counter is updated.

- There is a parameter that determines after how many transmissions the disease mutates. If there is a mutation, the global `variants` counter is updated, and from that point on, every infection is with the new variant. (Note: this is not very realistic. You are free to come up with a better model.)
- A each `Person` object has a vector of variants that they are recovered from; recovery from one variant only makes them immune from that specific variant, not from others.

**Exercise 50.7.** Add mutation to your model. Experiment with the mutation rate: as the mutation rate increases, the disease should stay in the population longer. Does the relation with vaccination rate change that you observed before?

### 50.2.6 Diseases without vaccine: Ebola and Covid-19

*This section is optional, for bonus points*

The project so far applies to diseases for which a vaccine is available, such as MMR for measles, mumps and rubella. The analysis becomes different if no vaccine exists, such as is the case for *ebola* and *covid-19*, as of this writing.

Instead, you need to incorporate 'social distancing' into your code: people do not get in touch with random others anymore, but only those in a very limited social circle. Design a model distance function, and explore various settings.

The difference between ebola and covid-19 is how long an infection can go unnoticed: the *incubation period*. With ebola, infection is almost immediately apparent, so such people are removed from the general population and treated in a hospital. For covid-19, a person can be infected, and infect others, for a number of days before they are sequestered from the population.

Add this parameter to your simulation and explore the behavior of the disease as a function of it.

## 50.3 Ethics

The subject of infectious diseases and vaccination is full of ethical questions. The main one is *The chances of something happening to me are very small, so why shouldn't I bend the rules a little?*. This reasoning is most often applied to vaccination, where people for some reason or other refuse to get vaccinated.

Explore this question and others you may come up with: it is clear that everyone bending the rules will have disastrous consequences, but what if only a few people did this?

## 50.4 Project writeup and submission

### 50.4.1 Program files

In the course of this project you have written more than one main program, but some code is shared between the multiple programs. Organize your code with one file for each main program, and a single 'library' file with the class methods.

You can do this two ways:

1. You make a 'library' file, say `infect_lib.cc`, and your main programs each have a line

   ```
   #include "infect_lib.cc"
   ```

   This is not the best solution, but it is acceptable for now.
2. The better solution requires you to use *separate compilation* for building the program, and you need a *header* file. You would now have `infect_lib.cc` which is compiled separately, and `infect_lib.h` which is included both in the library file and the main program:

   ```
   #include "infect_lib.h"
   ```

   See section 19.2.2Header filessubsection.19.2.2 for more information.

Submit all source files with instructions on how to build all the main programs. You can put these instructions in a file with a descriptive name such as `README` or `INSTALL`, or you can use a *makefile*.

### 50.4.2 Writeup

In the writeup, describe the 'experiments' you have performed and the conclusions you draw from them. The exercises above give you a number of questions to address.

For each main program, include some sample output, but note that this is no substitute for writing out your conclusions in full sentences.

The exercises in section 50.2.4Spreadingsubsection.50.2.4 ask you to explore the program behavior as a function of one or more parameters. Include a table to report on the behavior you found. You can use Matlab or Matplotlib in Python (or even Excell) to plot your data, but that is not required.

## 50.5 Bonus: mathematical analysis

The SIR model can also be modeled through coupled difference or differential equations.

1. The number $S_i$ of susceptible people at time $i$ decreases by a fraction

   $$S_{i+1} = S_i(1 - \lambda_i \, dt)$$

   where $\lambda_i$ is the product of the number of infected people and a constant that reflects the number of meetings and the infectiousness of the disease. We write:

   $$S_{i+1} = S_i(1 - \lambda I_i \, dt)$$

2. The number of infected people similarly increases by $\lambda S_i I_i$, but it also decreases by people recovering (or dying):

   $$I_{i+1} = I_i(1 + \lambda S_i \, dt - \gamma \, dt).$$

3. Finally, the number of 'removed' people equals that last term:

   $$R_{i+1} = R_i(1 + \gamma I_i).$$

**Exercise 50.8.** Code this scheme. What is the effect of varying $dt$?

**Exercise 50.9.** For the disease to become an epidemic, the number of newly infected has to be larger than the number of recovered. That is,

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma/\lambda.$$

Can you observe this in your simulations?

The parameter $\gamma$ has a simple interpretation. Suppose that a person stays ill for $\delta$ days before recovering. If $I_t$ is relatively stable, that means every day the same number of people get infected as recover, and therefore a $1/\delta$ fraction of people recover each day. Thus, $\gamma$ is the reciprocal of the duration of the infection in a given person.

# Chapter 51

# Google PageRank

## 51.1 Basic ideas

We are going to simulate the Internet. In particular, we are going to simulate the *Pagerank* algorithm by which *Google* determines the importance of web pages.

Let's start with some basic classes:

- A `Page` contains some information such as its title and a global numbering in Google's data-center. It also contains a collection of links.
- We represent a link with a pointer to a `Page`. Conceivably we could have a `Link` class, containing further information such as probability of being clicked, or number of times clicked, but for now a pointer will do.
- Ultimately we want to have a class `Web` which contains a number of pages and their links. The web object will ultimately also contain information such as relative importance of the pages.

This application is a natural one for using pointers. When you click on a link on a web page you go from looking at one page in your browser to looking at another. You could implement this by having a pointer to a page, and clicking updates the value of this pointer.

**Exercise 51.1.** Make a class `Page` which initially just contains the name of the page. Write a method to display the page. Since we will be using pointers quite a bit, let this be the intended code for testing:

```
auto homepage = make_shared<Page>("My Home Page");
cout << "Homepage has no links yet:" << '\n';
cout << homepage->as_string() << '\n';
```

Next, add links to the page. A link is a pointer to another page, and since there can be any number of them, you will need a `vector` of them. Write a method `click` that follows the link. Intended code:

```
auto utexas = make_shared<Page>("University Home Page");
homepage->add_link(utexas);
auto searchpage = make_shared<Page>("google");
homepage->add_link(searchpage);
cout << homepage->as_string() << '\n';
```

**Exercise 51.2.** Add some more links to your homepage. Write a method `random_click` for the `Page` class. Intended code:

```
for (int iclick=0; iclick<20; iclick++) {
```

```
  auto newpage = homepage->random_click();
  cout << "To: " << newpage->as_string() << '\n';
}
```

How do you handle the case of a page without links?

## 51.2 Clicking around

**Exercise 51.3.** Now make a class `Web` which foremost contains a bunch (technically: a `vector`) of pages. Or rather: of pointers to pages. Since we don't want to build a whole internet by hand, let's have a method `create_random_links` which makes a random number of links to random pages. Intended code:

```
Web internet(netsize);
internet.create_random_links(avglinks);
```

Now we can start our simulation. Write a method `Web::random_walk` that takes a page, and the length of the walk, and simulates the result of randomly clicking that many times on the current page. (Current page. Not the starting page.)

Let's start working towards PageRank. First we see if there are pages that are more popular than others. You can do that by starting a random walk once on each page. Or maybe a couple of times.

**Exercise 51.4.** Apart from the size of your internet, what other design parameters are there for your tests? Can you give a back-of-the-envelope estimation of their effect?

**Exercise 51.5.** Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

```
    vector<int> landing_counts(internet.number_of_pages(),0);
    for ( auto page : internet.all_pages() ) {
      for (int iwalk=0; iwalk<5; iwalk++) {
        auto endpage = internet.random_walk(page,2*avglinks,tracing);
        landing_counts.at(endpage->global_ID())++;
      }
    }
```

Display the results and analyze. You may find that you finish on certain pages too many times. What's happening? Fix that.

## 51.3 Graph algorithms

There are many algorithms that rely on gradually traversing the web. For instance, any graph can be *connected*. You test that by

- Take an arbitrary vertex $v$. Make a 'reachable set' $R \leftarrow \{v\}$.
- Now see where you can get from your reachable set:

$$\forall_{v\in V}\forall_{w \text{ neighbour of } v} : R \leftarrow R \cup \{w\}$$

- Repeat the previous step until $R$ does not change anymore.

After this algorithm concludes, is $R$ equal to your set of vertices? If so, your graph is called (fully) connected. If not, your graph has multiple *connected components*.

**Exercise 51.6.** Code the above algorithm, keeping track of how many steps it takes to reach each vertex $w$. This is the *Single Source Shortest Path* algorithm (for unweighted graphs).

The *diameter* is defined as the maximal shortest path. Code this.

## 51.4 Page ranking

The Pagerank algorithm now asks, if you keep clicking randomly, what is the distribution of how likely you are to wind up on a certain page. The way we calculate that is with a probability distribution: we assign a probability to each page so that the sum of all probabilities is one. We start with a random distribution:

**Code:**

```
ProbabilityDistribution
    random_state(internet.number_of_pages()
random_state.set_random();
cout << "Initial distribution: " <<
    random_state.as_string() << '\n';
```

**Output**
**[google] pdfsetup:**

```
Initial distribution:
    0:0.00, 1:0.02, 2:0.07,
    3:0.05, 4:0.06, 5:0.08,
    6:0.04, 7:0.04, 8:0.04,
    9:0.01, 10:0.07, 11:0.05,
    12:0.01, 13:0.04,
    14:0.08, 15:0.06,
    16:0.10, 17:0.06,
    18:0.11, 19:0.01,
```

**Exercise 51.7.** Implement a class `ProbabilityDistribution`, which stores a vector of floating point numbers. Write methods for:

- accessing a specific element,
- setting the whole distribution to random, and
- normalizing so that the sum of the probabilities is 1.
- a method rendering the distribution as string could be useful too.

Next we need a method that given a probability distribution, gives you the new distribution corresponding to performing a single click. (This is related to *Markov chains*; see HPC book [9], section 9.2.1.)

**Exercise 51.8.** Write the method

```
ProbabilityDistribution Web::globalclick
    ( ProbabilityDistribution currentstate );
```

Test it by

- start with a distribution that is nonzero in exactly one page;
- print the new distribution corresponding to one click;
- do this for several pages and inspect the result visually.

Then start with a random distribution and run a couple of iterations. How fast does the process converge? Compare the result to the random walk exercise above.

**Exercise 51.9.** In the random walk exercise you had to deal with the fact that some pages have no outgoing links. In that case you transitioned to a random page. That mechanism is lacking in the `globalclick` method. Figure out a way to incorporate this.

Let's simulate some simple 'search engine optimization' trick.

**Exercise 51.10.** Add a page that you will artificially made look important: add a number of pagesthat all link to this page, but no one links to them. (Because of the random clicking they will still sometimes be reached.)

Compute the rank of the artificially hyped page. Did you manage to trick Google into ranking this page high? How many links did you have to add?

Sample output:

```
Internet has 5000 pages
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 3465:0.0009, 4298:0.0008
With fake pages:
Internet has 5051 pages
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 5050:0.0010, 4298:0.0008
Hyped page scores at 4
```

## 51.5 Graphs and linear algebra

The probability distribution is essentially a vector. You can also represent the web as a matrix $W$ with $w_{ij} = 1$ if page $i$ links to page $j$. How can you interpret the `globalclick` method in these terms?

**Exercise 51.11.** Add the matrix representation of the `Web` object and reimplement the `globalclick` method. Test for correctness.

Do a timing comparison.

The iteration you did above to find a stable probability distribution corresponds to the 'power method' in linear algebra. Look up the Perron-Frobenius theory and see what it implies for page ranking.

# Chapter 52

# Redistricting

In this project you can explore 'gerrymandering', the strategic drawing of districts to give a minority population a majority of districts[1].

## 52.1 Basic concepts

We are dealing with the following concepts:

- A state is divided into census districts, which are given. Based on census data (income, ethnicity, median age) one can usually make a good guess as to the overall voting in such a district.
- There is a predetermined number of congressional districts, each of which consists of census districts. A congressional district is not a random collection: the census districts have to be contiguous.
- Every couple of years, to account for changing populations, the district boundaries are redrawn. This is known as redistricting.

There is considerable freedom in how redistricting is done: by shifting the boundaries of the (congressional) districts it is possible to give a population that is in the overall minority a majority of districts. This is known as *gerrymandering*.

For background reading, see `https://redistrictingonline.org/`.

To do a small-scale computer simulation of gerrymandering, we make some simplifying assumption.

- First of all, we dispense with census district: we assume that a district consists directly of voters, and that we know their affiliation. In practice one relies on proxy measures (such as income and education level) to predict affiliation.
- Next, we assume a one-dimensional state. This is enough to construct examples that bring out the essence of the problem:

  Consider a state of five voters, and we designate their votes as `AAABB`. Assigning them to three (contiguous) districts can be done as `AAA|B|B`, which has one 'A' district and two 'B' districts.
- We also allow districts to be any positive size, as long as the number of districts is fixed.

1. This project is obviously based on the Northern American political system. Hopefully the explanations here are clear enough. Please contact the author if you know of other countries that have a similar system.

## 52.2 Basic functions

### 52.2.1 Voters

We dispense with census districts, expressing everything in terms of voters, for which we assume a known voting behavior. Hence, we need a `Voter` class, which will record the voter ID and party affiliation. We assume two parties, and leave an option for being undecided.

**Exercise 52.1.** Implement a `Voter` class. You could for instance let $\pm 1$ stand for `A/B`, and 0 for undecided.

```
cout << "Voter 5 is positive:" << '\n';
Voter nr5(5,+1);
cout << nr5.print() << '\n';

cout << "Voter 6 is negative:" << '\n';
Voter nr6(6,-1);
cout << nr6.print() << '\n';

cout << "Voter 7 is weird:" << '\n';
Voter nr7(7,3);
cout << nr7.print() << '\n';
```

### 52.2.2 Populations

**Exercise 52.2.** Implement a `District` class that models a group of voters.

- You probably want to create a district out of a single voter, or a vector of them. Having a constructor that accepts a string representation would be nice too.
- Write methods `majority` to give the exact majority or minority, and `lean` that evaluates whether the district overall counts as A party or B party.
- Write a `sub` method to creates subsets.

  ```
  District District::sub(int first,int last);
  ```

- For debugging and reporting it may be a good idea to have a method

  ```
  string District::print();
  ```

**Code:**

```
    cout << "Making district with one B
    voter" << '\n';
    Voter nr5(5,+1);
    District nine( nr5 );
    cout << ".. size: " << nine.size() <<
    '\n';
    cout << ".. lean: " << nine.lean() <<
    '\n';
  /* ... */
    cout << "Making district ABA" << '\n';
    District nine( vector<Voter>
                   { {1,-1},{2,+1},{3,-1}
    } );
    cout << ".. size: " << nine.size() <<
    '\n';
    cout << ".. lean: " << nine.lean() <<
    '\n';
```

**Output**

```
[gerry] district:
Making district with one B voter
.. size: 1
.. lean: 1

Making district ABA
.. size: 3
.. lean: -1
```

**Exercise 52.3.** Implement a `Population` class that will initially model a whole state.

**Code:**

```
string pns( "-++--" );
Population some(pns);
cout << "Population from string " << pns
    << '\n';
cout << ".. size: " << some.size() <<
    '\n';
cout << ".. lean: " << some.lean() <<
    '\n';
Population group=some.sub(1,3);
cout << "sub population 1--3" << '\n';
cout << ".. size: " << group.size() <<
    '\n';
cout << ".. lean: " << group.lean() <<
    '\n';
```

**Output**

```
[gerry] population:
Population from string -++--
.. size: 5
.. lean: -1
sub population 1--3
.. size: 2
.. lean: 1
```

In addition to an explicit creation, also write a constructor that specifies how many people and what the majority is:

```
Population( int population_size,int majority,bool trace=false )
```

Use a random number generator to achieve precisely the indicated majority.

### 52.2.3 Districting

The next level of complication is to have a set of districts. Since we will be creating this incrementally, we need some methods for extending it.

**Exercise 52.4.** Write a class `Districting` that stores a `vector` of `District` objects. Write `size` and `lean` methods:

**Code:**

```
cout << "Making single voter population
    B" << '\n';
Population people( vector<Voter>{
    Voter(0,+1) } );
cout << ".. size: " << people.size() <<
    '\n';
cout << ".. lean: " << people.lean() <<
    '\n';

Districting gerry;
cout << "Start with empty districting:"
    << '\n';
cout << ".. number of districts: " <<
    gerry.size() << '\n';
```

**Output**

```
[gerry] gerryempty:
Making single voter population B
.. size: 1
.. lean: 1
Start with empty districting:
.. number of districts: 0
```

**Exercise 52.5.** Write methods to extend a `Districting`:

```
cout << "Add one B voter:" << '\n';
gerry = gerry.extend_with_new_district( people.at(0) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';
cout << "add A A:" << '\n';
gerry = gerry.extend_last_district( Voter(1,-1) );
gerry = gerry.extend_last_district( Voter(2,-1) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';

cout << "Add two B districts:" << '\n';
gerry = gerry.extend_with_new_district( Voter(3,+1) );
gerry = gerry.extend_with_new_district( Voter(4,+1) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';
```

## 52.3 Strategy

Now we need a method for districting a population:

```
Districting Population::minority_rules( int ndistricts );
```

Rather than generating all possible partitions of the population, we take an incremental approach (this is related to the solution strategy called *dynamic programming*):

- The basic question is to divide a population optimally over $n$ districts;
- We do this recursively by first solving a division of a subpopulation over $n-1$ districts,
- and extending that with the remaining population as one district.

This means that you need to consider all the ways of having the 'remaining' population into one district, and that means that you will have a loop over all ways of splitting the population, outside of your recursion; see figure 52.1Multiple ways of splitting a populationfigure.52.1.

- For all $p = 0, \ldots n-1$ considering splitting the state into $0, \ldots, p-1$ and $p, \ldots, n-1$.
- Use the best districting of the first group, and make the last group into a single district.

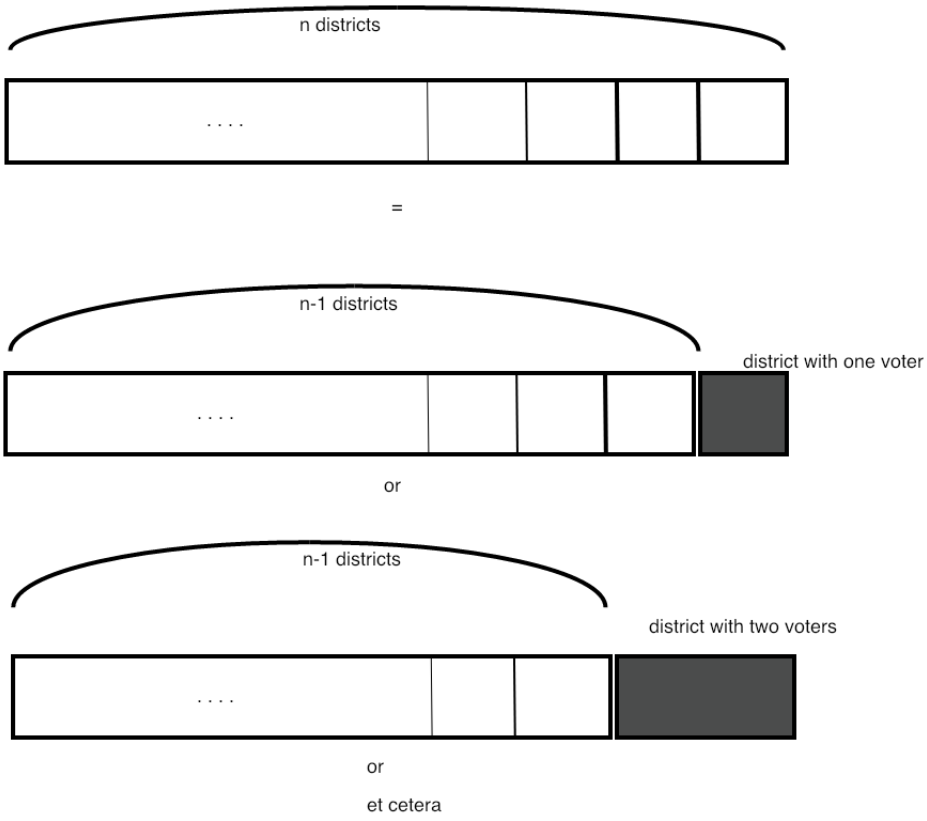



Figure 52.1: Multiple ways of splitting a population

- Keep the districting that gives the strongest minority rule, over all values of $p$.

You can now realize the above simple example:

```
AAABB => AAA|B|B
```

**Exercise 52.6.** Implement the above scheme.

**Code:**

```
Population five("+++--");
cout << "Redistricting population: " <<
    '\n'
     << five.print() << '\n';
cout << ".. majority rule: "
     << five.rule() << '\n';
int ndistricts{3};
auto gerry =
    five.minority_rules(ndistricts);
cout << gerry.print() << '\n';
cout << ".. minority rule: "
     << gerry.rule() << '\n';
```

**Output**

```
[gerry] district5:
Redistricting population:
[0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,],[3:-,],[4:-,],]
.. minority rule: -1
```

Note: the range for $p$ given above is not quite correct: for instance, the initial part of the population needs to be big enough to accommodate $n - 1$ voters.

**Exercise 52.7.** Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

## 52.4 Efficiency: dynamic programming

If you think about the algorithm you just implemented, you may notice that the districtings of the initial parts get recomputed quite a bit. A strategy for optimizing for this is called *memoization*.

**Exercise 52.8.** Improve your implementation by storing and reusing results for the initial sub-populations.

In a way, we solved the program backward: we looked at making a district out of the last so-many voters, and then recursively solving a smaller problem for the first however-many voters. But in that process, we decided what is the best way to assign districts to the first 1 voter, first 2, first 3, et cetera. Actually, for more than one voter, say five voters, we found the result on the best attainable minority rule assigning these five voters to one, two, three, four districts.

The process of computing the 'best' districting forward, is known as *dynamic programming*. The fundamental assumption here is that you can use intermediate results and extend them, without having to reconsider the earlier problems.

Consider for instance that you've considered districting ten voters over up to five districts. Now the majority for eleven voters and five districts is the minimum of

- ten voters and five districts, and the new voter is added to the last district; or
- ten voters and four districts, and the new voter becomes a new district.

**Exercise 52.9.** Code a dynamic programming solution to the redistricting problem.

## 52.5 Extensions

The project so far has several simplifying assumptions.

- Congressional districts need to be approximately the same size. Can you put a limit on the ratio between sizes? Can the minority still gain a majority?

**Exercise 52.10.** The biggest assumption is of course that we considered a one-dimensional state. With two dimensions you have more degrees of freedom of shaping the districts. Implement a two-dimensional scheme; use a completely square state, where the census districts form a regular grid. Limit the shape of the congressional districts to be convex.

The *efficiency gap* is a measure of how 'fair' a districting of a state is.

**Exercise 52.11.** Look up the definition of efficiency gap (and 'wasted votes'), and implement it in your code.

## 52.6 Ethics

The activity of redistricting was intended to give people a fair representation. In its degenerate form of Gerrymandering this concept of fairness is violated because the explicit goal is to give the minority a majority of votes. Explore ways that this unfairness can be undone.

In your explorations above, the only characteristic of a voter was their preference for party A or B. However, in practice voters can be considered part of communities. The Voting Rights Act is concerned about ‘minority vote dilution’. Can you show examples that a color-blind districting would affect some communities negatively?

# Chapter 53

# Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

## 53.1 Problem statement

Scheduling the route of a delivery truck is a well-studied problem. For instance, minimizing the total distance that the truck has to travel corresponds to the *Traveling Salesman Problem (TSP)*. However, in the case of *Amazon delivery truck* scheduling the problem has some new aspects:

- A customer is promised a window of days when delivery can take place. Thus, the truck can split the list of places into sublists, with a shorter total distance than going through the list in one sweep.
- Except that *Amazon prime* customers need their deliveries guaranteed the next day.

## 53.2 Coding up the basics

Before we try finding the best route, let's put the basics in place to have any sort of route at all.

### 53.2.1 Address list

You probably need a class `Address` that describes the location of a house where a delivery has to be made.

- For simplicity, let give a house $(i, j)$ coordinates.
- We probably need a `distance` function between two addresses. We can either assume that we can travel in a straight line between two houses, or that the city is build on a grid, and you can apply the so-called *Manhattan distance*.
- The address may also require a field recording the last possible delivery date.

**Exercise 53.1.** Code a class `Address` with the above functionality, and test it.

**Code:**

```
Address one(1.,1.),
  two(2.,2.);
cerr << "Distance: "
     << one.distance(two)
     << '\n';
```

**Output**

```
[amazon] address:

Address
Distance: 1.41421
.. address
Address 1 should be closest to
    the depot. Check: 1

Route from depot to depot:
    (0,0) (2,0) (1,0) (3,0)
    (0,0)
 has length 8: 8
Greedy scheduling:  (0,0) (1,0)
    (2,0) (3,0) (0,0)
 should have length 6: 6

Square5
Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
.. square5

Original list:  (0,0) (-2,0)
    (-1,0) (1,0) (2,0) (0,0)
 length=8
flip middle two addresses:
    (0,0) (-2,0) (1,0) (-1,0)
    (2,0) (0,0)
 length=12
better:  (0,0) (1,0) (-2,0)
    (-1,0) (2,0) (0,0)
 length=10

Hundred houses
Route in order has length
    25852.6
TSP based on mere listing has
    length: 2751.99 over naive
    25852.6
Single route has length: 2078.43
 .. new route accepted with
    length 2076.65
Final route has length 2076.65
    over initial 2078.43
TSP route has length 1899.4
    over initial 2078.43

Two routes
Route1:  (0,0) (2,0) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (3,1) (2,1)
    (1,2) (1,3) (0,0)
total length 19.6251
start with 9.88635,9.73877
Pass 0
.. down to 9.81256,8.57649
Pass 1
Pass 2
Pass 3
Pass 4
TSP Route1:  (0,0) (3,1) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (2,0) (2,1)
    (1,2) (1,3) (0,0)
total length 18.389
```

Next we need a class `AddressList` that contains a list of addresses.

**Exercise 53.2.** Implement a class `AddressList`; it probably needs the following methods:

- `add_address` for constructing the list;
- `length` to give the distance one has to travel to visit all addresses in order;
- `index_closest_to` that gives you the address on the list closest to another address, presumably not on the list.

### 53.2.2 Add a depot

Next, we model the fact that the route needs to start and end at the depot, which we put arbitrarily at coordinates $(0, 0)$. We could construct an `AddressList` that has the depot as first and last element, but that may run into problems:

- If we reorder the list to minimize the driving distance, the first and last elements may not stay in place.
- We may want elements of a list to be unique: having an address twice means two deliveries at the same address, so the `add_address` method would check that an address is not already in the list.

We can solve this by making a class `Route`, which inherits from `AddressList`, but the methods of which leave the first and last element of the list in place.

### 53.2.3 Greedy construction of a route

Next we need to construct a route. Rather than solving the full TSP, we start by employing a *greedy search* strategy:

> Given a point, find the next point by some local optimality test, such as shortest distance. Never look back to revisit the route you have constructed so far.

Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let's write a method

```
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

**Exercise 53.3.** Write the `greedy_route` method for the `AddressList` class.

1. Assume that the route starts at the depot, which is located at $(0, 0)$. Then incrementally construct a new list by:
2. Maintain an `Address` variable `we_are_here` of the current location;
3. repeatedly find the address closest to `we_are_here`.

Extend this to a method for the `Route` class by working on the subvector that does not contain the final element.

Test it on this example:

**Code:**

```
1     Route deliveries;
2     deliveries.add_address( Address(0,5)
      );
3     deliveries.add_address( Address(5,0)
      );
4     deliveries.add_address( Address(5,5)
      );
5     cerr << "Travel in order: " <<
      deliveries.length() << '\n';
6     assert( deliveries.size()==5 );
7     auto route =
      deliveries.greedy_route();
8     assert( route.size()==5 );
9     auto len = route.length();
10    cerr << "Square route: " <<
      route.as_string()
11          << "\n has length " << len <<
      '\n';
```

**Output**

```
[amazon] square5:

Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
```

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each address a boolean field `done`, but you could also make a copy of the input list, and remove the elements that have been processed. For this, study the `erase` method for `vector` objects.

## 53.3 Optimizing the route

The above suggestion of each time finding the closest address is known as a *greedy search* strategy. It does not give you the optimal solution of the TSP. Finding the optimal solution of the TSP is hard to program – you could do it recursively – and takes a lot of time as the number of addresses grows. In fact, the TSP is probably the most famous of the class of *NP-hard* problems, which are generally believed to have a running time that grows faster than polynomial in the problem size.

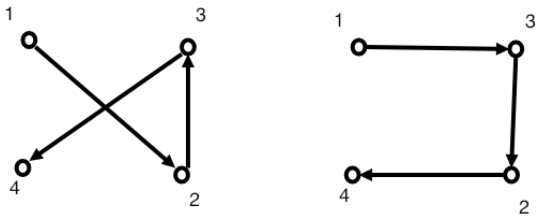



Figure 53.1: Illustration of the 'opt2' idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [16], is based on the *opt2* idea: if you have a path that 'crosses itself', you can make it shorter by reversing

part of it. Figure 53.1Illustration of the 'opt2' idea of reversing part of a pathfigure.53.1 shows that the path $1 - -2 - -3 - -4$ can be made shorter by reversing part of it, giving $1 - -3 - -2 - -4$. Since recognizing where a path crosses itself can be hard, or even impossible for graphs that don't have Cartesian coordinates associated, we adopt a scheme where we try all possible reversals:

```
for all nodes m<n on the path [1..N]:
  make a new route from
    [1..m-1] + [m--n].reversed + [n+1..N]
  if the new route is shorter, keep it
```

**Exercise 53.4.** Code the opt2 heuristic: write a method to reverse part of the route, and write the loop that tries this with multiple starting and ending points. Try it out on some simple test cases to convince you that your code works as intended.

Let's explore issues of complexity. (For an introduction to complexity calculations, see HPC book [9], section 19.) The TSP is one of a class of *NP complete* problems, which very informally means that there is no better solution than trying out all possibilities.

**Exercise 53.5.** What is the runtime complexity of the heuristic solution using opt2? What would the runtime complexity be of finding the best solution by considering all possibilities? Make a very rough estimation of runtimes of the two strategies on some problem sizes: $N = 10, 100, 1000, \ldots$.

**Exercise 53.6.** Earlier you had programmed the greedy heuristic. Compare the improvement you get from the opt2 heuristic, starting both with the given list of addresses, and with a greedy traversal of it.

For realism, how many addresses do you put on your route? How many addresses would a delivery driver do on a typical day?

## 53.4 Multiple trucks

If we introduce multiple delivery trucks, we get the 'Multiple Traveling Salesman Problem' [5]. With this we can module both the cases of multiple trucks being out on delivery on the same day, or one truck spreading deliveries over multiple days. For now we don't distinguish between the two.

The first question is how to divide up the addresses.

1. We could split the list in two, using some geometric test. This is a good model for the case where multiple trucks are out on the same day. However, if we use this as a model for the same truck being out on multiple days, we are missing the fact that new addresses can be added on the first day, messing up the neatly separated routes.
2. Thus it may in fact be reasonable to assume that all trucks get an essentially random list of addresses.

Can we extend the opt2 heuristic to the case of multiple paths? For inspiration take a look at figure 53.2Extending the 'opt2' idea to multiple pathsfigure.53.2: instead of modifying one path, we could switch bits out bits between one path and another. When you write the code, take into account that the other path may be running backwards! This means that based on split points in the first and second path you know have four resulting modified paths to consider.

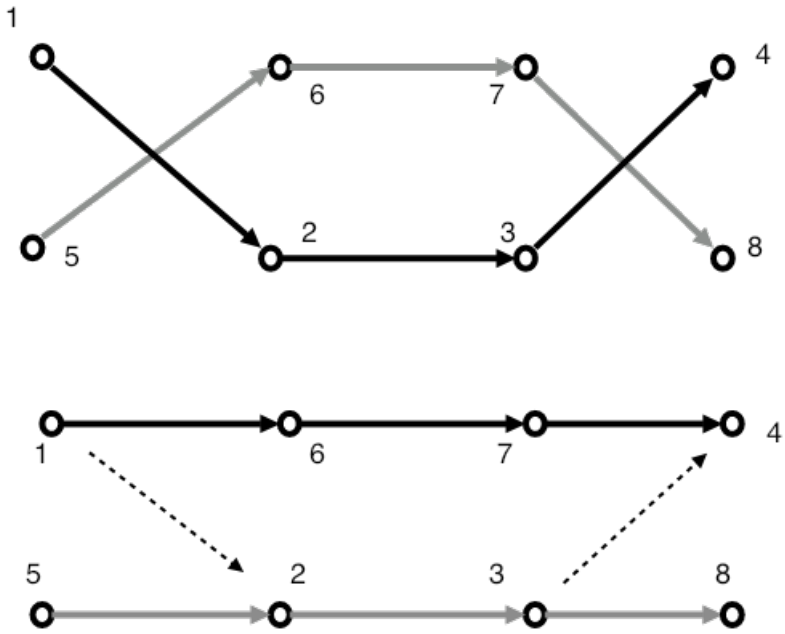



Figure 53.2: Extending the ‘opt2’ idea to multiple paths

**Exercise 53.7.** Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 53.3Multiple paths test casefigure.53.3.

You have quite a bit of freedom here:

- The start points of the two segments should be chosen independently;
- the lengths can be chosen independently, but need not; and finally
- each segment can be reversed.

More flexibility also means a longer runtime of your program. Does it pay off? Do some tests and report results.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

## 53.5 Amazon prime

In section 53.4Multiple truckssection.53.4 you made the assumption that it doesn’t matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

**Exercise 53.8.** Explore a scenario where there are two trucks, and each have a number of addresses that can not be exchanged with the other route. How much longer is the total distance? Experiment with the ratio of prime to non-prime addresses.

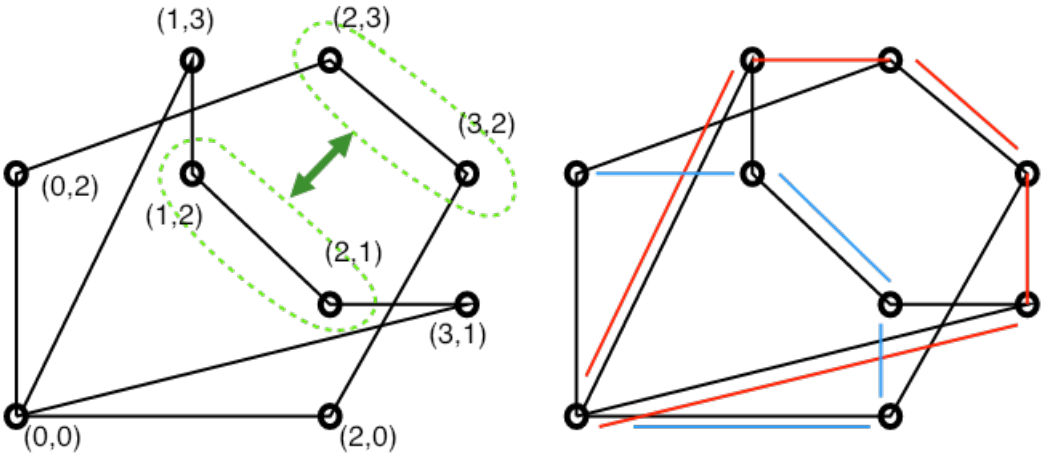



Figure 53.3: Multiple paths test case

## 53.6 Dynamicism

So far we have assumed that the list of addresses to be delivered to is given. This is of course not true: new deliveries will need to be scheduled continuously.

**Exercise 53.9.** Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

## 53.7 Ethics

People sometimes criticize Amazon's labor policies, including regarding its drivers. Can you make any observations from your simulations in this respect?

# Chapter 54

# High performance linear algebra

Linear algebra is fundamental to much of computational science. Applications involving Partial Diffential Equations (PDEs) come down to solving large systems of linear equation; solid state physics involves large eigenvalue systems. But even outside of engineering applications linear algebra is important: the major computational part of Deep Learning (DL) networks involves matrix-matrix multiplications.

Linear algebra operations such as the matrix-matrix product are easy to code in a naive way. However, this does not lead to high performance. In these exercises you will explore the basics of a strategy for high performance.

## 54.1 Mathematical preliminaries

The matrix-matrix product $C \leftarrow A \cdot B$ is defined as

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

Straightforward code for this would be:

```
for (i=0; i<a.m; i++)
  for (j=0; j<b.n; j++)
    s = 0;
    for (k=0; k<a.n; k++)
      s += a[i,k] * b[k,j];
    c[i,j] = s;
```

However, this is not the only way to code this operation. The loops can be permuted, giving a total of six implementations.

**Exercise 54.1.** Code one of the permuted algorithms and test its correctness. If the reference algorithm above can be said to be 'inner-product based', how would you describe your variant?

Yet another implementation is based on a block partitioning. Let $A, B, C$ be split on $2 \times 2$ block form:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11}&=A_{11}B_{11}+A_{12}B_{21}, \\ C_{12}&=A_{11}B_{12}+A_{12}B_{22}, \\ C_{21}&=A_{21}B_{11}+A_{22}B_{21}, \\ C_{22}&=A_{21}B_{12}+A_{22}B_{22} \end{aligned} \tag{54.1}$$

Convince yourself that this actually computes the same product $C = A \cdot B$. For more on block algorithms, see HPC book [9], section 5.3.8.

**Exercise 54.2.** Write a matrix class with a multiplication routine:

```
Matrix Matrix::MatMult( Matrix other );
```

First implement a traditional matrix-matrix multiplication, then make it recursive. For the recursive algorithm you need to implement sub-matrix handling: you need to extract submatrices, and write a submatrix back into the surrounding matrix.

## 54.2 Matrix storage

The simplest way to store an $M \times N$ matrix is as an array of length $MN$. Inside this array we can decide to store the rows end-to-end, or the columns. While this decision is obviously of practical importance for a library, from a point of performance it makes no difference.

**Remark 29** *Historically, linear algebra software such as the Basic Linear Algebra Subprograms (BLAS) has used columnwise storage, meaning that the location of an element $(i, j)$ is computed as $i + j \cdot M$ (we will use zero-based indexing throughout this project, both for code and mathematical expressions.) The reason for this stems from the origins of the BLAS in the Fortran language, which uses column-major ordering of array elements. On the other hand, static arrays (such as `x[5][6][7]`) in the C/C++ languages have row-major ordering, where element $(i, j)$ is stored in location $j + i \cdot N$.*

Above, you saw the idea of block algorithms, which requires taking submatrices. For efficiency, we don't want to copy elements into a new array, so we want the submatrix to correspond to a subarray.

Now we have a problem: only a submatrix that consists of a sequence of columns is contiguous. The formula $i + j \cdot M$ for location of element $(i, j)$ is no long correct if the matrix is a subblock of a larger matrix.

For this reason, linear algebra software describes a submatrix by three parameters $M, N, LDA$, where 'LDA' stands for 'leading dimension of $A$' (see BLAS [13], and Lapack [1]). This is illustrated in figure 54.1Submatrix out of a matrix, with $M$, $N$, $LDA$ of the submatrix indicatedfigure.54.1.

**Exercise 54.3.** In terms of $M, N, LDA$, what is the location of the $(i, j)$ element?

Implementationwise we also have a problem. If we use `std::vector` for storage, it is not possible to take subarrays, since C++ insists that a vector has its own storage. The solution is to use `span`; section 10.9.5Spansubsection.10.9.5.

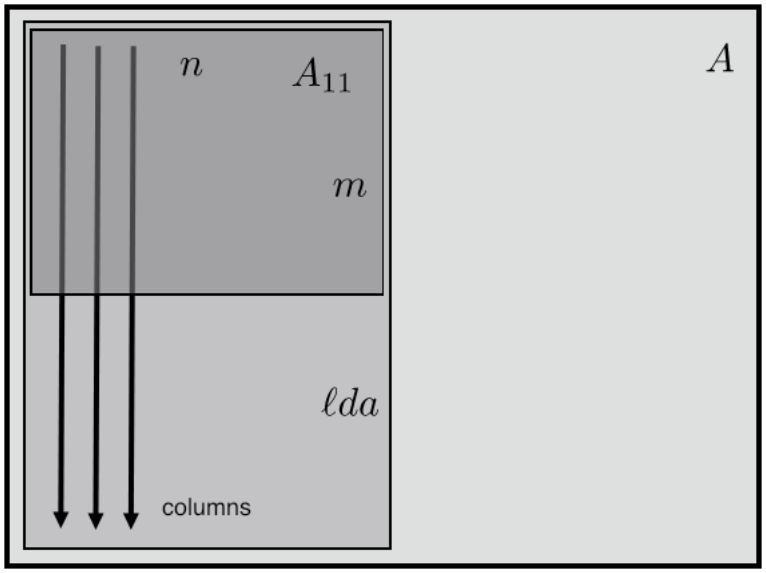



Figure 54.1: Submatrix out of a matrix, with $M$, $N$, $LDA$ of the submatrix indicated

We could have two types of matrices: top level matrices that store a `vector<double>`, and submatrices that store a `span<double>`, but that is a lot of complication. It could be done using `std::variant` (section 24.5.4Variantsubsection.24.5.4), but let's not.

Instead, let's adopt the following idiom, where we create a vector at the top level, and then create matrices from its memory.

```
// example values for M,LDA,N
M = 2; LDA = M+2; N = 3;
// create a vector to contain the data
vector<double> one_data(LDA*N,1.);
// create a matrix using the vector data
Matrix one(M,LDA,N,one_data.data());
```

(If you have not previously programmed in C, you need to get used to the `double*` mechanism. See section 10.10C style arrayssection.10.10.)

**Exercise 54.4.** Start implementing the `Matrix` class with a constructor

```
Matrix::Matrix(int m,int lda,int n,double *data)
```

and private data members:

```
private:
  int m,n,lda;
  span<double> data;
```

Write a method

```
double& Matrix::at(int i,int j);
```

that you can use as a safe way of accessing elements.

Let's start with simple operations.

**Exercise 54.5.** Write a method for adding matrices. Test it on matrices that have the same $M, N$, but different $LDA$.

Use of the `at` method is great for debugging, but it is not efficient. Use the preprocessor (chapter 21Preprocessorchapter.21) to introduce alternatives:

```
#ifdef DEBUG
  c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
  cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

where you access the data directly with

```
auto get_double_data() {
  double *adata;
  adata = data.data();
  return adata;
};
```

**Exercise 54.6.** Implement this. Use a cpp `#define` macro for the optimized indexing expression. (See section 21.2.3Parameterized macrossubsection.21.2.3.)

### 54.2.1 Submatrices

Next we need to support constructing actual submatrices. Since we will mostly aim for decomposition in $2 \times 2$ block form, it is enough to write four methods:

```
Matrix Left(int j);
Matrix Right(int j);
Matrix Top(int i);
Matrix Bot(int i);
```

where, for instance, `Left(5)` gives the columns with $j < 5$.

**Exercise 54.7.** Implement these methods and test them.

## 54.3 Multiplication

You can now write a first multiplication routine, for instance with a prototype

```
void Matrix::MatMult( Matrix& other,Matrix& out );
```

Alternatively, you could write

```
Matrix Matrix::MatMult( Matrix& other );
```

but we want to keep the amount of creation/destruction of objects to a minimum.

### 54.3.1 One level of blocking

Next, write

```
void Matrix::BlockedMatMult( Matrix& other,Matrix& out );
```

which uses the $2 \times 2$ form above.

### 54.3.2 Recursive blocking

The final step is to make the blocking recursive.

**Exercise 54.8.** Write a method

```
void RecursiveMatMult( Matrix& other,Matrix& out );
```

which

- Executes the $2 \times 2$ block product, using again `RecursiveMatMult` for the blocks.
- When the block is small enough, use the regular `MatMult` product.

## 54.4 Performance issues

If you experiment a little with the cutoff between the regular and recursive matrix-matrix product, you see that you can get good factor of performance improvement. Why is this?

The matrix-matrix product is a basic operation in scientific computations, and much effort has been put into optimizing it. One interesting fact is that it is just about the most optimizable operation under the sum. The reason for this is, in a nutshell, that it involves $O(N^3)$ operations on $O(N^2)$ data. This means that, in principle each element fetched will be used multiple times, thereby overcoming the *memory bottleneck*.

To understand performance issues relating to hardware, you need to do some reading. Section HPC book [9], section 1.3.4 explains the crucial concept of a *cache*.

**Exercise 54.9.** Argue that the naive matrix-matrix product implementation is unlikely actually to reuse data.

Explain why the recursive strategy does lead to data reuse.

Above, you set a cutoff point for when to switch from the recursive to the regular product.

**Exercise 54.10.** Argue that continuing to recurse will not have much benefit once the product is contained in the cache. What are the cache sizes of your processor?

Do experiments with various cutoff points. Can you relate this to the cache sizes?

### 54.4.1 Parallelism (optional)

The four clauses of equation 54.1Mathematical preliminariesequation.54.1.1 target independent areas in the $C$ matrix, so they could be executed in parallel on any processor that has at least four cores.

Explore the OpenMP library to parallelize the `BlockedMatMult`.

### 54.4.2 Comparison (optional)

The final question is: how close are you getting to the best possible speed? Unfortunately you are still a way off. You can explore that as follows.

Your computer is likely to have an optimized implementation, accessible through:

```
#include <cblas.h>

cblas_dgemm
  ( CblasColMajor, CblasNoTrans, CblasNoTrans,
    m,other.n,n, alpha,adata,lda,
    bdata,other.lda,
    beta,cdata,out.lda);
```

which computes $C \leftarrow \alpha A \cdot B + \beta C$.

**Exercise 54.11.** Use another cpp conditional to implement `MatMult` through a call to `cblas_dgemm`. What performance do you now get?

You see that your recursive implementation is faster than the naive one, but not nearly as fast as the CBlas one. This is because

- the CBlas implementation is probably based on an entirely different strategy [12], and
- it probably involves a certain amount of assembly coding.

# Chapter 55

# The Great Garbage Patch

This section contains a sequence of exercises that builds up to a *cellular automaton* simulation of turtles and garbage in the ocean. To read more about this: https://theoceancleanup.com/.

Thanks to Ernesto Lima of TACC for the idea and initial code of this exercise.

## 55.1 Problem and model solution

There is lots of plastic floating around in the ocean, and that is harmful to fish and turtles and cetaceans. Here you get to model the interaction between

- Plastic, randomly located;
- Turtles that swim about; they breed slowly, and they die from ingesting plastic;
- Ships that sweep the ocean to remove plastic debris.

The simulation method we use is that of a *cellular automaton*:

- We have a grid of cells;
- Each cell has a 'state' associated with it, namely, it can contain a ship, a turtle, or plastic, or be empty; and
- On each next time step, the state of a cell is a simple function of the states of that cell and its immediate neighbors.

The purpose of this exercise is to make a simulate a number of time steps, and explore the iteraction bbetween parameters: with how much gargage will turtles die out, how many ships are enough to protect the turtles.

## 55.2 Program design

The basic idea is to have an `ocean` object that is populated with turtles, trash, and ships. Your simulation will let the ocean undergo a number of time steps:

```
for (int t=0; t<time_steps; t++)
  ocean.update();
```

Ultimately your purpose is to investigate the development of the turtle population: is it stable, does it die out?

While you can make a 'hackish' solution to this problem, partly you will be judged on your use of modern/clean C++ programming techniques. A number of suggestions are made below.

### 55.2.1 Grid update

Here is a point to be aware of. Can you see what's wrong with with doing an update entirely in-place:

```
for ( i )
  for ( j )
    cell(i,j) = f( cell(i,j) );
```

?

## 55.3 Modern programming techniques

### 55.3.1 Object oriented programming

While you will only have one ocean, you should still make an `ocean` class, rather than having a global array object. All functions are then methods of the single object you create of that class.

### 55.3.2 Data structure

We lose very little generality by ignoring the depth of the ocean and the shape of coastlines, and model it as a 2D grid.

If you write an indexing function `cell(i,j)` you can make your code largely independent of the actual data structure chosen. Argue why `vector<vector<int>>` is not the best storage.

1. What do you use instead?
2. When you have a working code, can you show by timing that your choice is indeed superior?

### 55.3.3 Cell types

Having 'magic numbers' in your code (0 =empty, 1 =turtle, et cetera) is not elegant. Make a `enum` or `enum class` (see section 24.10Enum classessection.24.10) so that you have names for what's in your cells:

```
cell(i,j) = occupy::turtle;
```

If you want to print out your ocean, it might be nice if you can directly `cout` your cells:

```
    for (int i=0; i<iSize; i++) {
      cout << i%10 << " ";
      for (int j=0; j<jSize; j++) {
        cout << setw(hs) << cell(i,j);
      }
      cout << '\n';
    }
```

### 55.3.4 Ranging over the ocean

It is easy enough to write a loop as

```
for (int i=0; i<iSize; i++)
  for (int j=0; j<jSize; j++)
    ... cell(i,j) ...
```

However, it may not be a good idea to always sweep over your domain so orderly. Can you implement this:

```
for ( auto [i,j] : permuted_indices() ) {
    ... cell(i,j) ...
```

? See section 24.4Tuples and structured bindingsection.24.4 about *structured binding*.

Likewise, if you need to count how many pieces of trash there are around a turtle, can you get this code to work:

```
int count_around( int ic,int jc,occupy typ ) const {
  int count=0;
  for ( auto [i,j] : neighbors(ic,jc) ) {
    if (cell(i,j) == typ)
      count++;
  }
  return count;
};
```

### 55.3.5 Random numbers

For the random movement of ships and turtles you need a random number generator. Do not use the old C generator, but the new `random` one; section 24.6Random numberssection.24.6.

Try to find a solution so that you use exactly one generator for all places where you need random numbers. Hint: make the generator `static` in your class.

## 55.4 Testing

It can be complicated to test this program for correctness. The best you can do is to try out a number of scenarios. For that it's best to make your program input flexible: use the `cxxopts` package, and drive your program from a shell script.

Here is a list of things you can test.

1. Start with only a number of ships; check that after 1000 time steps you still have the same number.
2. Likewise with turtles; if they don't breed and don't die; check that their number stays constant.
3. With only ships and trash, does it all get sweeped?
4. With only turtles and trash, do they all die off?

It is harder to test that your turtles and ships don't 'teleport' around, but only move to contiguous cells. For that, use visual inspection; see section 55.4.1.

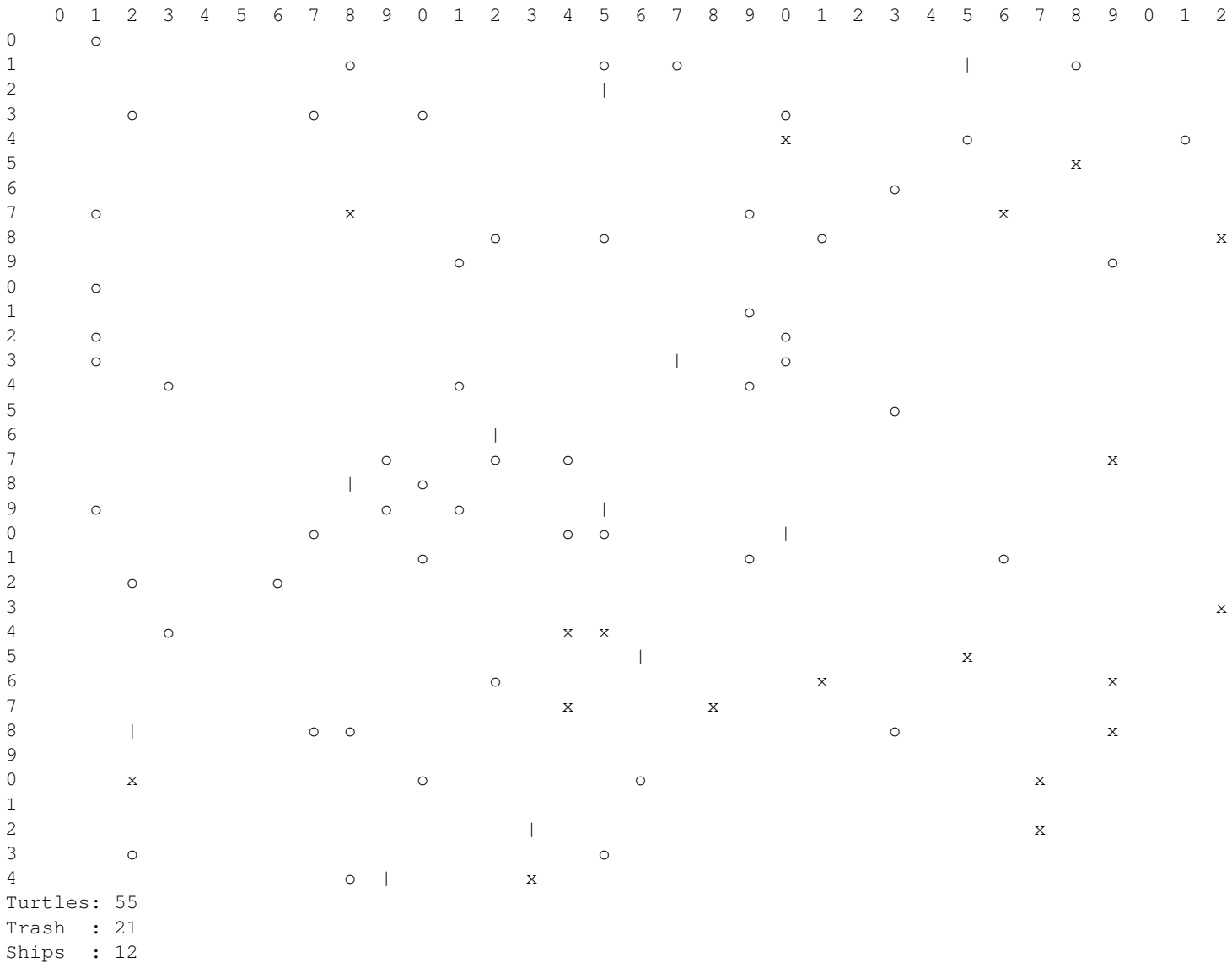



Figure 55.1: Ascii art printout of a time step

### 55.4.1 Animated graphics

The output of this program is a prime candidate for visualization. In fact, some test ('make sure that turtles don't teleport') are hard to do other than by looking at the output. However, not all programming languages generate visual output equally easily. There are very powerful video/graphics libraries in C++, but these are also hard to use. There is a simpler way out.

For simple output such as this program yields, you can make a simple low-budget animation. Every *terminal emulator* under the sun supports *VT100 cursor control* [1]: you can send certain magic output to your screen to control cursor positioning.

In each time step you would

1. Send the cursor to the home position, by this magic output:

```
#include <cstdio>
  /* ... */
    // ESC [ i ; j H
    printf( "%c[0;0H",(char)27);
```

2. Display your grid as in figure 55.1;
3. Sleep for a fraction of a second; see section 26.1.4The current threadsubsection.26.1.4.

## 55.5 Explorations

Instead of having the ships move randomly, can you give them a preferential direction to the closest garbage patch? Does this improve the health of the turtle population?

Can you account for the relative size of ships and turtles by having a ship occupy a $2 \times 2$ block in your grid?

So far you have let trash stay in place. What if there are ocean currents? Can you make the trash 'sticky' so that trash particles start moving as a patch if they touch?

Turtles eat sardines. (No they don't.) What happens to the sardine population if turtles die out? Can you come up with parameter values that correspond to a stable ecology or a de-stabilized one?

### 55.5.1 Code efficiency

Investigate whether your implementation of the `enum` in section 55.3.3 has any effect on timing. Parse the fine print of section 24.10Enum classessection.24.10.

You may remark that ranging over a largely empty ocean can be pretty inefficient. You could contemplate keeping an 'active list' of where the turtles et cetera are located, and only looping over that. How would you implement that? Do you expect to see a difference in timing? Do you actually?

How is runtime affected by choosing a vector-of-vectors implementation for the ocean; see section 55.3.2.

1. https://vt100.net/docs/vt100-ug/chapter3.html

# Chapter 56

# Graph algorithms

In this project you will explore some common graph algorithms, and their various possible implementations. The main theme here will be that the common textbook exposition of algorithms is not necessarily the best way to phrase them computationally.

As background knowledge for this project, you are encouraged to read HPC book [9], chapter 9; for an elementary tutorial on graphs, see HPC book [9], chapter 24.

## 56.1 Traditional algorithms

We first implement the 'textbook' formulations of two *Single Source Shortest Path (SSSP)* algorithms: on unweighted and then on weighted graphs. In the next section we will then consider formulations that are in terms of linear algebra.

In order to develop the implementations, we start with some necessary preliminaries,

### 56.1.1 Code preliminaries

#### *56.1.1.1 Adjacency graph*

We need a class `Dag` for a Directed Acyclic Graph (DAG):

```
class Dag {
private:
  vector< vector<int> > dag;
public:
  // Make Dag of 'n' nodes, no edges for now.
  Dag( int n )
    : dag( vector< vector<int> >(n) ) {};
```

It's probably a good idea to have a function

```
const auto& neighbors( int i ) const { return dag.at(i); };
```

that, given a node, returns a list of the neighbors of that node.

**Exercise 56.1.** Finish the `Dag` class. In particular, add a method to generate example graphs:

- For testing the 'circular' graph is often useful: connect edges

$$0 \to 1 \to \cdots \to N-1 \to 0.$$

- It may also be a good idea to have a graph with random edges.

Write a method that displays the graph.

#### 56.1.1.2 Node sets

The classic formulation of SSSP algorithms, such as *Dijkstra's algorithm* (see HPC book [9], section 9.1.3) uses sets of nodes that are gradually built up or depleted.

You could implement that as a vector:

```
vector< int > set_of_nodes(nnodes);
for ( int inode=0; inode<nnodes; inode++)
  // mark inode as distance unknown:
  set_of_nodes.at(inode) = inf;
```

where you use some convention, such as negative distance, to indicate that a node has been removed from the set.

However, C++ has an actual `set` container with methods for adding an element, finding it, and removing it; see section 24.2.2. This makes for a more direct expression of our algorithms. In our case, we'd need a set of int/int or int/float pairs, depending on the graph algorithm. (It is also possible to use a `map`, using an int as lookup key, and int or float as values.)

For the unweighted graph we only need a set of finished nodes, and we insert node 0 as our starting point:

```
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances;
distances.insert( {0,0} );
```

For Dijkstra's algorithm we need both a set of finished nodes, and nodes that we are still working on. We again set the starting node, and we set the distance for all unprocessed nodes to infinity:

```
const unsigned inf = std::numeric_limits<unsigned>::max();
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances,to_be_done;

to_be_done.insert( {0,0} );
for (unsigned n=1; n<graph_size; n++)
  to_be_done.insert( {n,inf} );
```

(Why do we need that second set here, while it was not necessary for the unweighted graph case?)

**Exercise 56.2.** Write a code fragment that tests if a node is in the `distances` set.

- You can of course write a loop for this. In that case know that iterating over a set gives you the key/value pairs. Use *structured bindings*; section 24.4.
- But it's better to use an 'algorithm', in the technical sense of 'algorithms built into the standard

library'. In this case, `find`.
- ... except that with `find` you have to search for an exact key/value pair, and here you want to search: 'is this node in the `distances` set with whatever value'. Use the `find_if` algorithm; section 24.2.2.

### 56.1.2 Level set algorithm

We start with a simple algorithm: the SSSP algorithm in an unweighted graph; see HPC book [9], section 9.1.1 for details. Equivalently, we find level sets in the graph.

For unweighted graphs, the distance algorithm is fairly simple. Inductively:

- Assume that we have a set of nodes reachable in at most $n$ steps,
- then their neighbors (that are not yet in this set) can be reached in $n+1$ steps.

The algorithm outline is

```
for (;;) {
  if (distances.size()==graph_size) break;
  /*
   * Loop over all nodes that are already done
   */
  for ( auto [node,level] : distances ) {
    /*
     * set neighbors of the node to have distance 'level + 1'
     */
    const auto& nbors = graph.neighbors(node);
    for ( auto n : nbors ) {
      /*
       * See if 'n' has a known distance,
       * if not, add to 'distances' with level+1
       */
/* ... */
        {
          cout << "node " << n << " level " << level+1 << '\n';
          distances.insert( {n,level+1} );
        }
```

**Exercise 56.3.** Finish the program that computes the SSSP algorithm and test it.

This code has an obvious inefficiency: for each level we iterate through all finished nodes, even if all their neighbors may already have been processed.

**Exercise 56.4.** Maintain a set of 'current level' nodes, and only investigate these to find the next level. Time the two variants on some large graphs.

### 56.1.3 Dijkstra's algorithm

In Dijkstra's algorithm we maintain both a set of nodes for which the shortest distance has been found, and one for which we are still determining the shortest distance. Note: a tentative shortest distance for a node may be updated several times, since there may be multiple paths to that node. The 'shortest' path in terms of weights may not be the shortest in number of edges traversed!

The main loop now looks something like:

```
for (;;) {
  if (to_be_done.size()==0) break;
  /*
   * Find the node with least distance
   */
/* ... */
  cout << "min: " << nclose << " @ " << dclose << '\n';
  /*
   * Move that node to done,
   */
  to_be_done.erase(closest_node);
  distances.insert( *closest_node );
  /*
   * set neighbors of nclose to have that distance + 1
   */
  const auto& nbors = graph.neighbors(nclose);
  for ( auto n : nbors ) {
    // find 'n' in distances
/* ... */
      {
        /*
         * if 'n' does not have known distance,
         * find where it occurs in 'to_be_done' and update
         */
/* ... */
        to_be_done.erase( cfind );
        to_be_done.insert( {n,dclose+1} );
/* ... */
}
```

(Note that we erase a record in the `to_be_done` set, and then re-insert the same key with a new value. We could have done a simple update if we had used a `map` instead of a `set`.)

The various places where you find nodes in the finished / unfinished sets are up to you to implement. You can use simple loops, or use `find_if` to find the elements matching the node numbers.

**Exercise 56.5.** Fill in the details of the above outline to realize Dijkstra's algorithm.

## 56.2 Linear algebra formulation

In this part of the project you will explore how much you make graph algorithms look like linear algebra.

### 56.2.1 Code preliminaries

#### *56.2.1.1 Data structures*

You need a matrix and a vector. The vector is easy:

```
class Vector {
private:
  vector<vectorvalue> values;
public:
  Vector( int n )
    : values( vector<vectorvalue>(n,infinite) ) {};
```

For the matrix, use initially a dense matrix:

```
class AdjacencyMatrix {
private:
  vector< vector<matrixvalue> > adjacency;
public:
  AdjacencyMatrix(int n)
    : adjacency( vector<vector<matrixvalue>>
                 ( n,vector<matrixvalue>(n,empty) ) ) {
  };
```

but later we will optimize that.

**Remark 30** *In general it's not a good idea to store a matrix as a vector-of-vectors, but in this case we need to be able to return a matrix row, so it is convenient.*

#### *56.2.1.2 Matrix vector multiply*

Let's write a routine

```
Vector AdjacencyMatrix::leftmultiply( const Vector& left );
```

This is the simplest solution, but not necessarily the most efficient one, as it creates a new vector object for each matrix-vector multiply.

As explained in the theory background, graph algorithms can be formulated as matrix-vector multiplications with unusual add/multiply operations. Thus, the core of the multiplication routine could look like

```
    for ( int row=0; row<n; row++ ) {
      for ( int col=0; col<n; col++) {
        result[col] = add( result[col], mult( left[row],adjacency[row][col] ) );
      }
    }
```

### 56.2.2 Unweighted graphs

**Exercise 56.6.** Implement the `add` / `mult` routines to make the SSSP algorithm on unweighted graphs work.

### 56.2.3 Dijkstra's algorithm

As an example, consider the following adjacency matrix:

```
.  1  .  .  5
.  .  1  .  .
.  .  .  1  .
.  .  .  .  1
1  .  .  .  .
```

The shortest distance $0 \to 4$ is 4, but in the first step a larger distance of 5 is discovered. Your algorithm should show an output similar to this for the successive updates to the known shortest distances:

```
Input :  0 . . . .
step 0:  0 1 . . 5
step 1:  0 1 2 . 5
step 2:  0 1 2 3 5
step 3:  0 1 2 3 4
```

**Exercise 56.7.** Implement new versions of the `add` / `mult` routines to make the matrix-vector multiplication correspond to Dijkstra's algorithm for SSSP on weighted graphs.

### 56.2.4 Sparse matrices

The matrix data structure described above can be made more compact by storing only nonzero elements. Implement this.

### 56.2.5 Further explorations

How elegant can you make your code through operator overloading?

Can you code the all-pairs shortest path algorithm?

Can you extend the SSSP algorithm to also generate the actual paths?

## 56.3 Tests and reporting

You now have two completely different implementations of some graph algorithms. Generate some large matrices and time the algorithms.

Discuss your findings, paying attention to amount of work performed and amount of memory needed.

# Chapter 57

# Memory allocation

**This project is not yet ready**

https://www.youtube.com/watch?v=R3cBbvIFqFk

Monotonic allocator

- base and free pointer,
- always allocate from the free location
- only release when everything has been freed.

appropriate:

- video processing: release everything used for one frame
- event processing: release everything used for handling the event

Stack allocator

# Chapter 58

# Ballistics calculations

**THIS PROJECT IS NOT READY FOR PRIME TIME**

## 58.1 Introduction

From https://encyclopedia2.thefreedictionary.com/ballistics

> Ballistics
>
> the science of the movement of artillery shells, bullets, mortar shells, aerial bombs, rocket artillery projectiles and missiles, harpoons, and so on. Ballistics is a technical military science based on a set of physics and mathematics disciplines. Interior ballistics is distinguished from exterior ballistics.
>
> Interior ballistics is concerned with the movement of a projectile (or other body whose mechanical freedom is restricted by certain conditions) in the bore of a gun under the influence of powder gases as well as the rules governing other processes occurring in the bore or in the chamber of a powder rocket during firing. Interior ballistics views the firing as a complex process of rapid transformation of the powder's chemical energy into heat energy and then into the mechanical work of displacing the projectile, charge, and recoil parts of the gun. In interior ballistics the different periods that are distinguished in the firing are the preliminary period, which is from the start of the powder combustion until the projectile begins to move; the first (primary) period, which is from the start of projectile movement until the end of powder combustion; the second period, which is from the end of powder combustion until the moment that the projectile leaves the bore (the period of adiabatic expansion of the gases); and the period of the aftereffect of the powder gases on the projectile and barrel. The laws governing the processes related to this last period are dealt with in a special division of ballistics, known as intermediate ballistics. The end of the period of aftereffect on the projectile divides the phenomena studied by interior and exterior ballistics.
>
> The main divisions of interior ballistics are "pyrostatics," "pyrodynamics," and ballistic gun design. Pyrostatics is the study of the laws of powder combustion and gas formation during the combustion of powder in a constant volume in which the effect of the chemical composition of the powder and its forms and dimensions on the laws of combustion and gas formation is determined. Pyrodynamics is concerned

with the study of the processes and phenomena that take place in the bore during firing and the determination of the relationships between the design features of the bore, the conditions of loading, and various physical-chemical and mechanical processes that occur during firing. On the basis of a consideration of these processes and also of the forces operating on the projectile and barrel, a system of equations is established that describes the firing process, including the basic equation of interior ballistics, which relates the magnitude of the burned part of the charge, the pressure of powder gases in the bore, the velocity of the projectile, and the length of the path it has traveled. The solution of this system and the discovery of the relationship between change in the pressure $\rho$ of the powder gases, the velocity v of the projectile, and other parameters on path l of the projectile and the time it has moved along the bore is the first main (direct)

to solve this problem the analytic method, numerical integration methods (including those based on computers), and tabular methods are used. In view of the complexity of the firing process and insufficient study of particular factors, certain assumptions are made. The correction formulas of interior ballistics are of great practical significance; they make it possible to determine the change in muzzle velocity of the projectile and maximum pressure in the bore when there are changes made in the loading conditions.

Ballistic gun design is the second main (correlative) problem of interior ballistics. By it are determined the design specifications of the bore and the loading conditions under which a projectile of given caliber and mass will obtain an assigned (muzzle) velocity in flight. The curves of change in the pressure of the gases in the bore and of the velocity of the projectile along the length of the barrel and in time are calculated for the variation of the barrel selected during designing. These curves are the initial data in designing the artillery system as a whole and the ammunition for it. Internal ballistics also includes the study of the firing process in the rifle, in cases when special and combined charges are used, in systems with conical barrels, and in systems in which gases are exhausted during powder combustion (high-low pressure guns and recoilless guns, infantry mortars). Another important division is the interior ballistics of powder rockets, which has developed into a special science. The main divisions of the interior ballistics of powder rockets are pyrostatics of a semiclosed space, which consider the laws of powder combustion at comparatively low and constant pressure; the solution of the basic problem of the interior ballistics of powder rockets, which is to determine (under set loading conditions) the rules of variation in pressure of the powder gases in the chamber with regard to time and to determine the rules of variation in thrust necessary to ensure the required rocket velocity; the ballistic design of powder rockets, which involves determining the energy-producing characteristics of the powder, the weight and form of the charge, and the design parameters of the nozzle which ensure, with an assigned weight of the rocket's warhead, the necessary thrust force during its operation.

Exterior ballistics is concerned with the study of the movement of unguided projectiles (mortar shells, bullets, and so on) after they leave the bore (or launcher) and the factors that affect this movement. It includes basically the study of all the elements of motion of the projectile and of the forces that act upon it in flight (the force of air resistance, the force of gravity, reactive force, the force arising in the

aftereffect period, and so on); the study of the movement of the center of mass of the projectile for the purpose of calculating its trajectory (see Figure 2) when there are set initial and external conditions (the basic problem of exterior ballistics); and the determination of the flight stability and dispersion of projectiles. Two important divisions of exterior ballistics are the theory of corrections, which develops methods of evaluating the influence of the factors that determine the projectile's flight on the nature of its trajectory, and the technique for drawing up firing tables and of finding the optimal exterior ballistics variation in the designing of artillery systems. The theoretical solution of the problems of projectile movement and of the problems of the theory of corrections amounts to making up equations for the projectile's movement, simplifying these equations, and seeking methods of solving them. This has been made significantly easier and faster with the appearance of computers. In order to determine the initial conditions—that is, initial velocity and angle of departure, shape and mass of the projectile—which are necessary to obtain a given trajectory, special tables are used in exterior ballistics. The working out of the technique for drawing up a firing table involves determining the optimal combination of theoretical and experimental research that makes it possible to obtain firing tables of the required accuracy with the minimal expenditure of time. The methods of exterior ballistics are also used in the study of the laws of movement of spacecraft (during their movement without the influence of controlling forces and moments). With the appearance of guided missiles, exterior ballistics played a major part in the formation and development of the theory of flight and became a particular instance of this theory.

The appearance of ballistics as a science dates to the 16th century. The first works on ballistics are the books by the Italian N. Tartaglia, A New Science (1537) and Questions

and Discoveries Relating to Artillery Fire (1546). In the 17th century the fundamental principles of exterior ballistics were established by Galileo, who developed the parabolic theory of projectile movement, by the Italian E. Torricelli, and by the Frenchman M. Mersenne, who proposed that the science of the movement of projectiles be called ballistics (1644). I. Newton made the first investigations of the movement of a projectile, taking air resistance into consideration (Mathematical Principles of Natural Philosophy,1687). During the 17th and 18th centuries the movement of projectiles was studied by the Dutchman C. Huygens, the Frenchman P. Varignon, the Englishman B. Robins, the Swiss D. Bernoulli, the Russian scientist L. Eiler, and others. The experimental and theoretical foundations of interior ballistics were laid in the 18th century in works of Robins, C. Hutton, Bernoulli, and others. In the 19th century the laws of air resistance were established (the laws of N. V. Maievskii and N. A. Zabudskii, Havre's law, and A. F. Siacci's law).

The numerical analysis of ballistics calculations on the *ENIAC* are described in [14].

### 58.1.1 Physics

These are the governing equations:

$$\begin{aligned} x'' &= -E(x' - w_x) + 2\Omega \cos L \sin \alpha y' \\ y'' &= -Ey' - g - 2\Omega \cos L \sin \alpha x' \\ z'' &= -E(z' - w_z) + 2\Omega \sin L x' + 2\Omega \cos L \cos \alpha y' \end{aligned} \tag{58.1}$$

where

- $x, y, z$ are the quantities of interest: distance, altitude, sideways displacement;
- $w_x, w_z$ are wind speed;
- $E$ is a complicated function of $y$, involving air density and speed of sound;
- $\alpha$ is the azimuth, that is, angle of firing;
- All other quantities are needed for physical realism, but will be set $\equiv 1$ in this coding exercise.

### 58.1.2 Numerical analysis

This uses an Euler-MacLaurin scheme of third order:

$$f_1 - f_0 = \frac{1}{2}(f'_0 + f'_1)h + \frac{1}{12}(f'_0 - f'_1)h^2 + O(h^5) \tag{58.2}$$

which works out to

$$\begin{aligned} \bar{x}'_1 &= x'_0 + x''_0 \Delta t \\ x_1 &= x_0 + x'_0 \Delta t \\ x'_1 &= x'_0 + (x''_0 + \bar{x}''_1)\tfrac{\Delta t}{2} \\ x_1 &= x_0 (x'_0 + \bar{x}'_1)\tfrac{\Delta t}{2} + (x''_0 - \bar{x}''_1)\tfrac{\Delta t^2}{12} \end{aligned} \tag{58.3}$$

# Chapter 59

# Cryptography

## 59.1 The basics

While floating point numbers can span a rather large range – up to $10^{300}$ or so for double precision – integers have a much smaller one: up to about $10^9$. That is not enough to do cryptographic applications, which deal in much larger numbers. (Why can't we use floating point numbers?)

So the first step is to write classes `Integer` and `Fraction` that have no such limitations. Use operator overloading to make simple expressions work:

```
Integer big=2000000000; // two billion
big *= 1000000; bigger = big+1;
Integer one = bigger % big;
```

**Exercise 59.1.** Code Farey sequences.

## 59.2 Cryptography

https://simple.wikipedia.org/wiki/RSA_algorithm

https://simple.wikipedia.org/wiki/Exponentiation_by_squaring

## 59.3 Blockchain

Implement a blockchain algorithm.

# Chapter 60

# DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

## 60.1 Basic functions

Refer to section 24.4.

First we set up some basic mechanisms.

**Exercise 60.1.** There are four bases, `A,C,G,T`, and each has a complement: `A` ↔ `T`, `C` ↔ `G`. Implement this through a `map`, and write a function

```
char BaseComplement(char);
```

**Exercise 60.2.** Write code to read a *Fasta* file into a `string`. The first line, starting with >, is a comment; all other lines should be concatenated into a single string denoting the genome.

Read the virus genome in the file `lambda_virus.fa`.

Count the four bases in the genome two different ways. First use a `map`. Time how long this takes. Then do the same thing using an array of length four, and a conditional statement.

Bonus: try to come up with a faster way of counting. Use a vector of length 4, and find a way of computing the index directly from the letters `A,C,G,T`. Hint: research ASCII codes and possibly bit operations.

## 60.2 De novo shotgun assembly

One approach to generating a genome is to cut it to pieces, and algorithmically glue them back together. (It is much easier to sequence the bases of a short read than of a very long genome.)

If we assume that we have enough reads that each genome position is covered, we can look at the overlaps by the reads. One heuristic is then to find the Shortest Common Superset (SCS).

### 60.2.1 Overlap layout consensus

1. Make a graph where the reads are the vertices, and vertices are connected if they overlap; the amount of overlap is the edge weight.
2. The SCS is then a *Hamiltonian path* through this graph – this is already NP-complete.
3. Additionally, we optimize for maximum total overlap.
   ⇒ Traveling Salesman Problem (TSP), NP-hard.

Rather than finding the optimal superset, we can use a greedy algorithm, where every time we find the read with maximal overlap.

Repeats are often a problem. Another is spurious subgraphs from sequencing errors.

### 60.2.2 De Bruijn graph assembly

## 60.3 'Read' matching

A 'read' is a short fragment of DNA, that we want to match against a genome. In this section you will explore algorithms for this type of matching.

While here we mostly consider the context of genomics, such algorithms have other applications. For instance, searching for a word in a web page is essentially the same problem. Consequently, there is a considerable history of this topic.

### 60.3.1 Naive matching

We first explore a naive matching algorithm: for each location in the genome, see if the read matches up.

```
ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT
  ACCAAGAACGTG
    ^ mismatch

ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT
      ACCAAGAACGTG
      total match
```

**Exercise 60.3.** Code up the naive algorithm for matching a read. Test it on fake reads obtained by copying a substring from the genome. Use the genome in `phix.fa`.

Now read the *Fastq* file `ERR266411_1.first1000.fastq`. Fastq files contains groups of four lines: the second line in each group contains the reads. How many of these reads are matched to the genome?

Reads are not necessarily a perfect match; in fact, each fourth line in the fastq file gives an indication of the 'quality' of the corresponding read. How many matches do you get if you take a substring of the first 30 or so characters of each read?

### 60.3.2 Boyer-Moore matching

The *Boyer-Moore* string matching algorithm [6] is much faster than naive matching, since it uses two clever tricks to weed out comparisons that would not give a match.

### Bad character rule

In naive matching, we determined match locations left-to-right, and then tried matching left-to-right. In Bowers-Moore (BM), we still find match locations left-to-right, but we do our matching right-to-left.

```
   vvvv match location
antidisestablishmentarianism
   blis
    ^bad character
```

The mismatch is an ‘l’ in the pattern, which does not match a ‘d’ in the text. Since there is no ‘d’ in the pattern at all, we move the pattern completely past the mismatch:

```
   vvvv match location
antidisestablishmentarianism
       blis
```

in fact, we move it further, to the first match on the first character of the pattern:

```
           vvvv match location
antidisestablishmentarianism
           blis
           ^ first character match
```

The case where we have a mismatch, but the character in the text does appear in the pattern is a little trickier: we find the next occurrence of the mismatched character in the pattern, and use that to determine the shift distance.

```
shoobeedoobeeboobah
edoobeeboob
       ^ mismatch
 ^ other occurrence of 'd'
```

Note that this can be a considerably smaller shift than in the previous case.

```
        v
shoobeedoobeeboobah
      edoobeeboob
       ^ match the bad character 'd'
      ^ new location
```

**Exercise 60.4.** Discuss how efficient you expect this heuristic to be in the context of genomics versus text searching. (See above.)

### Good suffix rule

The ‘good suffix’ consists of the matched characters after the bad character. When moving the read, we try to keep the good suffix intact:

```
desistrust
```

```
  listrest
        ^^good suffix

desistrust
      listrest
        ^^next occurrence of suffix
```

# Chapter 61

# Climate change

*The climate has changed and it is always changing.*
Raj Shah, White House Principal Deputy Press Secretary

The statement that climate always changes is far from a rigorous scientific claim. We can attach a meaning to it, if we interpret it as a statement about the statistical behavior of the climate, in this case as measured by average global temperature. In this project you will work with real temperature data, and do some simple analysis on it. (The inspiration for this project came from [15].)

Ideally, we would use data sets from various measuring stations around the world. Fortran is then a great language because of its array operations (see chapter 40Arrayschapter.40): you can process all independent measurements in a single line. To keep things simple we will use a single data file here that contains data for each month in a time period 1880-2018. We will then use the individual months as 'pretend' independent measurements.

## 61.1 Reading the data

In the repository you find two text files

`GLB.Ts+dSST.txt` `GLB.Ts.txt`

that contain temperature deviations from the 1951–1980 average. Deviations are given for each month of each year 1880–2018. These data files and more can be found at `https://data.giss.nasa.gov/gistemp/`.

**Exercise 61.1.** Start by making a listing of the available years, and an array `monthly_deviation` of size $12 \times \mathrm{nyears}$, where nyears is the number of full years in the file. Use formats and array notation.

The text files contain lines that do not concern you. Do you filter them out in your program, or are you using a shell script? Hint: a judicious use of `grep` will make the Fortran code much easier.

## 61.2 Statistical hypothesis

We assume that mr Shah was really saying that climate has a 'stationary distribution', meaning that highs and lows have a probability distribution that is independent of time. This means that in $n$ data points,

each point has a chance of $1/n$ to be a record high. Since over $n+1$ years each year has a chance of $1/(n+1)$, the $n+1$st year has a chance $1/(n+1)$ of being a record.

We conclude that, as a function of $n$, the chance of a record high (or low, but let's stick with highs) goes down as $1/n$, and that the gap between successive highs is approximately a linear function of the year[1].

This is something we can test.

**Exercise 61.2.** Make an array `previous_record` of the same shape as `monthly_deviation`. This array records (for each month, which, remember, we treat like independent measurements) whether that year was a record, or, if not, when the previous record occurred:

$$\mathrm{PrevRec}(m,y) = \begin{cases} y & \text{if } \mathrm{MonDev}(m,y) = \max_{m'}\bigl(\mathrm{MonDev}(m',y)\bigr) \\ y' & \text{if } \mathrm{MonDev}(m,y) < \mathrm{MonDev}(m,y') \\ & \text{and } \mathrm{MonDev}(m,y') = \max_{m''<m'}\bigl(\mathrm{MonDev}(m'',y)\bigr) \end{cases}$$

Again, use array notation. This is also a great place to use the `Where` clause.

**Exercise 61.3.** Now take each month, and find the gaps between records. This gives you two arrays: `gapyears` for the years where a gap between record highs starts, and `gapsizes` for the length of that gap.

This function, since it is applied individually to each month, uses no array notation.

The hypothesis is now that the gapsizes are a linear function of the year, for instance measured as distance from the starting year. Of course they are not exactly a linear function, but maybe we can fit a linear function through it by *linear regression*.

**Exercise 61.4.** Copy the code from `http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf` and adapt for our purposes: find the best fit for the slope and intercept for a linear function describing the gaps between records.

You'll find that the gaps are decidedly not linearly increasing. So is this negative result the end of the story, or can we do more?

**Exercise 61.5.** Can you turn this exercise into a test of global warming? Can you interpret the deviations as the sum of a yearly increase in temperature plus a stationary distribution, rather than a stationary distribution by itself?

1. Technically, we are dealing with a uniform distribution of temperatures, which makes the maxima and minima have a beta-distribution.

# Chapter 62

# Desk Calculator Interpreter

In this set of exercises you will write a 'desk calculator': a small interactive calculator that combines numerical and symbolic calculation.

These exercises mostly use the material of chapters 37Structures, eh, typeschapter.37, 42Input/outputchapter.42, 36String handlingchapter.36.

## 62.1 Named variables

We start out by working with 'named variables': the `namedvar` type associates a string with a variable:

```
type namedvar
   character(len=20) :: expression = ""
   integer :: value
end type namedvar
```

A named variable has a value, and a string field that is the expression that generated the variable. When you create the variable, the expression can be anything.

```
type(namedvar) :: x,y,z,a
x = namedvar("x",1 )
y = namedvar("yvar",2 )
```

Next we are going to do calculations with these type objects. For instance, adding two objects

- adds their values, and
- concatenates their `expression` fields, giving the expression corresponding to the sum value.

Your first assignment is to write `varadd` and `varmult` functions that get the following program working with the indicated output. This uses string manipulation from sections 36.3 and 42.5.

**Exercise 62.1.** The following main program should give the corresponding output:

**Code:**

```
print *,x
print *,y
z = varadd(x,y)
print *,z
a = varmult(x,z)
print *,a
```

**Output**
**[structf] varhandling:**

```
x                                1
yvar                             2
(x)+(yvar)                       3
(x)*((x)+(yvar))                 3
```

(To be clear: the two routines need to do both numeric and string 'addition' and 'multiplication'.)

*You can base this off the file `namedvar.F90` in the repository*

## 62.2 First modularization

Let's organize the code so far by introducing modules; see chapter 38.

**Exercise 62.2.** Create a module (suggested name: `VarHandling`) and move the `namedvar` type definition and the routines `varadd`, `varmult` into it.

**Exercise 62.3.** Also create a module (suggested name: `InputHandling`) that contains the routines `islower`, `isdigit` from the character exercises in chapter 36. You will also need an `isop` routine to recognize arithmetic operations.

## 62.3 Event loop and stack

In our quest to write an interpreter, we will write an 'event loop': a loop that continually accepts single character inputs, and processes them. An input of `"0"` will mean termination of the process.

**Exercise 62.4.** Write a loop that accepts character input, and only prints out what kind of character was encountered: a lowercase character, a digit, or a character denoting an arithmetic operation `+-*/`.

**Code:**

```
do
   read *,input
   if (input .eq. '0') then
      exit
   else if ( isdigit(input) ) then
```

**Output**
**[structf] interchar:**

```
Inputs: 4 x 3 + 0
 4 is a digit
 x is a lowercase
 3 is a digit
 + is an operator
```

Use the `InputHandling` module introduced above.

### 62.3.1 Stack

Next, we are going to store values in `namedvar` types on a stack. A *stack* is a data structure where new elements go on the top, so we need to indicate with a *stack pointer* that top element. Equivalently, the stack pointer indicates how many elements there already are:

```
type(namedvar),dimension(10) :: stack
integer :: stackpointer=0
```

Since we are using modules, let's keep the stack out of the main program and put it in the appropriate module.

**Exercise 62.5.** Add the `stack` variable and the stack pointer to the `VarHandling` module.

Since Fortran uses 1-based indexing, a starting value of zero is correct. For C/C++ it would have been `-1`.

Next we will start implementing stack operations, such as putting `namedvar` objects on the stack.

### 62.3.2 Stack operations

We extend the above event loop, which was only recognizing the input characters, by actually incorporating actions. That is, we repeatedly

1. read a character from the input;
2. `0` causes the event loop to exit; otherwise:
3. if it is a digit, create a new `namedvar` entry on the top of the stack, with that value both numerically as the `value` field, and as string in the `expression` field.
   You may be tempted to write the following in the main program:

   ```
   if ( isdigit(input) ) then
      stackpointer = stackpointer + 1
      read( input,'( i1 )' ) stack(stackpointer)%value
      stack(stackpointer)%expression = trim(input)
   ```

   (You have already coded `isdigit` in exercise 36.1.) but a cleaner design uses a function call to a method in the `VarHandling` module:

   ```
   else if ( isdigit(input) ) then
      call stack_push(input)
   ```

   ```
   subroutine stack_push(input)
     implicit none
     character,intent(in) :: input
   ```

   Note that the `stack_push` routine does not have the stack or stack pointer as arguments: since they are all in the same module, they are accessible as *global variable*.
   Finally,
4. if it is a letter indicating an operation $+, -, \times, /$,
   (a) take the two top entries from the stack, lowering the stack pointer;
   (b) apply that operation to the operands; and
   (c) push the result onto the stack.

The auxiliary function `stack_display` is a little tricky, so you get that here. This uses string formatting (section 42.3) and implied do loops (section 33.3): Also, note that the `stack` array and the `stackpointer` act like global variables.

```
subroutine stack_display()
  implicit none
  ! local variables
  integer :: istck
```

```
  if (stackpointer.eq.0) return
  print '( 10( a,a, a,i0,"; ") )', ( &
       " expr=",trim(stack(istck)%expression), &
       " val=",stack(istck)%value, &
       istck=1,stackpointer )

end subroutine stack_display
```

Let's add the various options to the event loop.

**Exercise 62.6.** Make your event loop accept digits, creating a new entry:

**Code:**

```
else if ( isdigit(input) ) then
   call stack_push(input)
```

**Output**
**[structf] internum:**

```
Inputs: 4 5 6 0
 expr=4 val=4;
 expr=4 val=4;  expr=5 val=5;
 expr=4 val=4;  expr=5 val=5;  expr=6 val=6;
```

Next we integrate the operations: if the `input` character corresponds to an arithmetic operator, we call `stack_op` with that character. That routine in turn calls the appropriate operation depending on what the character was.

**Exercise 62.7.** Add a clause to your event loop to handle characters that stand for arithmetic operations:

**Code:**

```
else if ( isop(input) ) then
   call stack_op(input)
```

**Output**
**[structf] internumop:**

```
Inputs: 4 5 6 + + 0
 expr=4 val=4;
 expr=4 val=4;  expr=5 val=5;
 expr=4 val=4;  expr=5 val=5;  expr=6 val=6;
 expr=4 val=4;  expr=(5)+(6) val=11;
 expr=(4)+((5)+(6)) val=15;
```

### 62.3.3 Item duplication

Finally, we may want to use a stack entry more than once, so we need the functionality of duplicating a stack entry.

For this we need to be able to refer to a stack entry, so we add a single character label field: the `namedvar` type now stores

1. a single character id,

2. an integer value, and
3. its generating expression as string.

```
type namedvar
   character :: id
   character(len=20) :: expression
   integer :: value
end type namedvar
```

**Exercise 62.8.** Add the `id` field to the `namedvar`, and make sure your program still compiles and runs.

The event loop is now extended with an extra step. If the input character is a lowercase letter, it is used as the `id` of a `namedvar` as follows.

- If there is already a stack entry with that `id`, it is duplicated on top of the stack;
- otherwise, the `id` of the stack top entry is set to this character.

Here is the relevant bit of the new `stack_print` function:

```
print '( 10( a,a1, a,a, a,i0,"; ") )', ( &
     "id:",stack(istck)%id, &
     " expr=",trim(stack(istck)%expression), &
     " val=",stack(istck)%value, &
     istck=1,top )
```

**Exercise 62.9.** Write the missing function and its clause in the event loop:

**Code:**

```
1 stacksearch = find_on_stack(stack,stackpointer,input)
2 if ( stacksearch>=1 ) then
3    stackpointer = stackpointer+1
4    stack(stackpointer) =  stack(stacksearch)
```

**Output**

```
[structf] stackfind:

Inputs: 1 x 2 y x y + z 0
id:. expr=1 val=1;
id:x expr=1 val=1;
id:x expr=1 val=1; id:. expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2; id:x expr=1 val=1;
id:x expr=1 val=1; id:y expr=2 val=2; id:x expr=1 val=1; id:y expr=2
    val=2;
id:x expr=1 val=1; id:y expr=2 val=2; id:. expr=(1)+(2) val=3;
id:x expr=1 val=1; id:y expr=2 val=2; id:z expr=(1)+(2) val=3;
```

(What is in the **else** part of this conditional?)

## 62.4 Modularizing

With the modules and the functions you have developed so far, you have a very clean main program:

```
do
   call stack_display()
   read *,input
   if (input .eq. '0') exit
   if ( isdigit(input) ) then
      call stack_push(input)
   else if ( isop(input) ) then
      call stack_op(input)
   else if ( islower(input) ) then
      call stack_name(input)
   end if
end do
```

You see that by moving the stack into the module, neither the stack variable nor the stack pointer are visible in the main program anymore.

But there is an important limitation to this design: there is exactly one stack, declared as a sort of global variable, accessible through a module.

Whether having global data is good practice is another matter. In this case it's defensible: in a calculator app there will be exactly one stack.

## 62.5 Object orientation

But maybe we do sometimes need more than one stack. Let's bundle up the stack array and the stack pointer in a new type:

```
type stackstruct
   type(namedvar),dimension(10) :: data
   integer :: top=0
 contains
   procedure,public :: display, find_id, name, op, push
end type stackstruct
```

**Exercise 62.10.** Change the event loop so that it calls methods of the `stackstruct` type, rather than functions that take the stack as input.

For instance, the `push` function is called as:

```
if ( isdigit(input) ) then
   call thestack%push(input)
```

### 62.5.1 Operator overloading

The `varadd` and similar arithmetic routines use a function call for what we would like to write as an arithmetic operation.

**Exercise 62.11.** Use *operator overloading* in the `varop` function:

```
if (op=="+") then
   varop = op1 + op2
```

et cetera.

# PART V

# ADVANCED TOPICS

# Chapter 63

# External libraries

If you have a C++ compiler, you can write as much software as your want, by yourself. However, some things that you may need for your work have already been written by someone else. How can you use their software?

## 63.1 What are software libraries?

In this chapter you will learn about the use of *software libraries*: software that is written not as a standalone package, but in such a way that you can access its functionality in your own program.

Software libraries can be enormous, as is the case for scientific libraries, which are often multi-person multi-year projects. On the other hand, many of them are fairly simple utilities written by a single programmer. In the latter case you may have to worry about future support of that software.

### 63.1.1 Using an external library

Using a software library typically means that

- your program has a line

```
#include "fancylib.h"
```

- and you compile and link as:

```
icpc -c yourprogram.cxx -I/usr/include/fancylib
icpc -o yourprogram yourprogram.o -L/usr/lib/fancylib -lfancy
```

You will see specific examples below.

If you are now worried about having to do a lot of typing every time you compile,

- if you use an IDE, you can typically add the library in the options, once and for all; or
- you can use *Make* for building your program. See the tutorial.

### 63.1.2 Obtaining and installing an external library

Sometimes a software library is available through a *package manager*, but we are going to do it the old-fashioned way: downloading and installing it ourselves.

A popular location for finding downloadable software is `github.com`. You can then choose whether to

- clone the repository, or
- download everything in one file, typically with `.tgz` or `.tar.gz` extension; in that case you need to unpack it
  ```
  tar fxz fancylib.tgz
  ```
  This usually gives you a directory with a name such as
  ```
  fancylib-1.0.0
  ```
  containing the source and documentation of the library, but not any binaries or machine-specific files.

Either way, from here on we assume that you have a directory containing the downloaded package.

There are two main types of installation:

- based on *GNU autotools*, which you recognize by the presence of a *configure* program;
  ```
  cmake ## lots of options
  make
  make install
  ```
  or
- based on *Cmake*, which you recognize by the presence of `CMakeLists.txt` file:
  ```
  configure ## lots of options
  make
  make install
  ```

#### *63.1.2.1 Cmake installation*

The easiest way to install a package using `cmake` is to create a build directory, next to the source directory. The `cmake` command is issued from this directory, and it references the source directory:

```
mkdir build
cd build
cmake ../fancylib-1.0.0
make
make install
```

Some people put the build directory inside the source directory, but that is bad practice.

Apart from specifying the source location, you can give more options to `cmake`. The most common are

- specifying an install location, for instance because you don't have *superuser* privileges on that machine; or

- specifying the compiler, because Cmake will be default use the *gcc* compilers, but you may want the Intel compiler.

```
CC=icc CXX=icpc \
cmake \
    -D CMAKE_INSTALL_PREFIX:PATH=${HOME}/mylibs/fancy \
    ../fancylib-1.0.0
```

## 63.2 Options processing: cxxopts

Suppose you have a program that does something with a large array, and you want to be able to change your mind about the array size. You could

- You could recompile your program every time.
- You could let your program parse *argv*, and hope you remember precisely how your commandline options are to be interpreted.
- You could use the *cxxopts* library. This is what we will be exploring now.

### 63.2.1 Traditional commandline parsing

Commandline options are available to the program through the (optional) *argv* and *argc* options of the *main* program. The former is an array of strings, with the second the length:

```
int main( int argc,char **argv ) { /* program */ };
```

For simple cases it would be feasible to parse such options yourself:

**Code:**

```
cout << "Program name: " << argv[0] <<
    '\n';
for (int iarg=1; iarg<argc; iarg++)
  cout << "arg: " << iarg
       << argv[iarg] << " => "
       << atoi( argv[iarg] ) << '\n';
```

**Output**
**[args] argcv:**

```
./argcv 5 12
Program name: ./argcv
arg 1: 5 => 5
arg 2: 12 => 12
./argcv abc 3.14 foo
Program name: ./argcv
arg 1: abc => 0
arg 2: 3.14 => 3
arg 3: foo => 0
```

but this 1. gets tedious quickly 2. is difficult to make robust. Therefore, we will now see a library that makes such options handling relatively easy.

### 63.2.2 The cxxopts library

The *cxxopts* 'commandline argument parser' can be found at https://github.com/jarro2783/cxxopts. After a *Cmake* installation, it is a 'header-only' library.

- Include the header

```
#include "cxxopts.hpp"
```

which requires a compile option:

```
-I/path/to//cxxopts/installdir/include
```

- Declare an options object:

```
cxxopts::Options options("programname", "Program description");
```

- Add options:

```
// define '-n 567' option:
options.add_options()
  ("n,ntimes","number of times",
   cxxopts::value<int>()->default_value("37")
   )
  ;
/* ... */
// read out '-n' option and use:
auto number_of_times = result["ntimes"].as<int>();
cout << "Using number of times: " << number_of_times << '\n';
```

- Add array options

```
//define '-a 1,2,5,7' option:
options.add_options()
  ("a,array","array of values",
   cxxopts::value< vector<int> >()->default_value("1,2,3")
   )
  ;
/* ... */
auto array = result["array"].as<vector<int>>();
cout << "Array: " ;
for ( auto a : array ) cout << a << ", ";
cout << '\n';
```

- Add positional arguments:

```
// define 'positional argument' option:
options.add_options()
  ("keyword","whatever keyword",
   cxxopts::value<string>())
  ;
options.parse_positional({"keyword"});
/* ... */
// read out keyword option and use:
auto keyword = result["keyword"].as<string>();
cout << "Found keyword: " << keyword << '\n';
```

- Parse the options:

```
auto result = options.parse(argc, argv);
```

- Get help flag first:

```
options.add_options()
  ("h,help","usage information")
  ;
/* ... */
auto result = options.parse(argc, argv);
```

```
if (result.count("help")>0) {
  cout << options.help() << '\n';
  return 0;
}
```

- Get result values for both options and arguments:

```
auto number_of_times = result["ntimes"].as<int>();
cout << "Using number of times: " << number_of_times << '\n';

auto keyword = result["keyword"].as<string>();
cout << "Found keyword: " << keyword << '\n';
```

Options can be specified the usual ways:

```
myprogram -n 10
myprogram --nsize 100
myprogram --nsize=1000
myprogram --array 12,13,14,15
```

**Exercise 63.1.** Incorporate this package into primality testing: exercise 46.20.

## 63.3 Catch2 unit testing

Test a simple function

```
int five() { return 5; }
```

Successful test:

**Code:**

```
1 TEST_CASE( "needs to be 5","[1]" ) {
2     REQUIRE( five()==5 );
3 }
```

**Output**
**[catch] require:**

```
Filters: [1]
===============================================================
All tests passed (1
    assertion in 1 test case)
```

Unsuccessful test:

**Code:**

```
1 TEST_CASE( "not six","[2]" ) {
2     REQUIRE( five()==6 );
3 }
```

**Output**
**[catch] requirerr:**

```
require.cxx:30: FAILED:
  REQUIRE( five()==6 )
with expansion:
  5 == 6

===============================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Function that throws:

```
void even( int e ) {
    if (e%2==1) throw(1);
    cout << "Even number: "
         << e << '\n';
}
```

Test that it throws or not:

**Code:**

```
TEST_CASE( "even fun","[3]" ) {
    REQUIRE_NOTHROW( even(2) );
    REQUIRE_THROWS( even(3) );
}
```

**Output**

```
[catch] requireven:

Filters: [3]
Even number: 2
-------------------------------------------------------------------------------
All tests passed (2
    assertions in 1 test case)
```

Run the same test for a set of numbers:

**Code:**

```
TEST_CASE( "even set","[4]" ) {
    int e = GENERATE( 2,4,6,8 );
    REQUIRE_NOTHROW( even(e) );
}
```

**Output**

```
[catch] requirgen:

Filters: [4]
Even number: 2
Even number: 4
Even number: 6
Even number: 8
-------------------------------------------------------------------------------
All tests passed (4
    assertions in 1 test case)
```

How is this different from using a loop? Using `GENERATE` runs each value as a separate program.

Variants:

```
int i = GENERATE( range(1,100) );
int i = GENERATE_COPY( range(1,n) );
```

# Chapter 64

# Programming strategies

## 64.1 A philosophy of programming

Yes, your code will be executed by the computer, but:

- You need to be able to understand your code a month or year from now.
- Someone else may need to understand your code.
- ⇒ make your code readable, not just efficient

- Don't waste time on making your code efficient, until you know that that time will actually pay off.
- Knuth: 'premature optimization is the root of all evil'.
- ⇒ first make your code correct, then worry about efficiency

- Variables, functions, objects, form a new 'language':
  code in the language of the application.
- ⇒ your code should look like it talks about the application, not about memory.
- Levels of abstraction: implementation of a language should not be visible on the use level of that language.

## 64.2 Programming: top-down versus bottom up

The exercises in chapter 46 were in order of increasing complexity. You can imagine writing a program that way, which is formally known as *bottom-up* programming.

However, to write a sophisticated program this way you really need to have an overall conception of the structure of the whole program.

Maybe it makes more sense to go about it the other way: start with the highest level description and gradually refine it to the lowest level building blocks. This is known as *top-down* programming.

https://www.cs.fsu.edu/~myers/c++/notes/stepwise.html

Example:

Run a simulation

becomes

Run a simulation:
    Set up data and parameters
    Until convergence:
        Do a time step

becomes

Run a simulation:
    Set up data and parameters:
        Allocate data structures
        Set all values
    Until convergence:
        Do a time step:
            Calculate Jacobian
            Compute time step
            Update

You could do these refinement steps on paper and wind up with the finished program, but every step that is refined could also be a subprogram.

We already did some top-down programming, when the prime number exercises asked you to write functions and classes to implement a given program structure; see for instance exercise 46.8.

A problem with top-down programming is that you can not start testing until you have made your way down to the basic bits of code. With bottom-up it's easier to start testing. Which brings us to...

### 64.2.1 Worked out example

Take a look at exercise 6.13. We will solve this in steps.

1. State the problem:
```
// find the longest sequence
```

2. Refine by introducing a loop
```
// find the longest sequence:

// Try all starting points
// If it gives a longer sequence report
```

3. Introduce the actual loop:
```
// Try all starting points
for (int starting=2; starting<1000; starting++) {
// If it gives a longer sequence report
}
```

4. Record the length:
```
// Try all starting points
int maximum_length=-1;
```

```
for (int starting=2; starting<1000; starting++) {
  // If the sequence from 'start' gives a longer sequence report:
  int length=0;
  // compute the sequence from 'start'
  if (length>maximum_length) {
    // Report this sequence as the longest
  }
}
```

5. Refine computing the sequence:
```
// compute the sequence from 'start'
int current=starting;
while (current!=1) {
  // update current value
  length++;
}
```

6. Refine the update of the current value:
```
// update current value
if (current%2==0)
  current /= 2;
else
  current = 3*current+1;
```

## 64.3 Coding style

After you write your code there is the issue of *code maintainance*: you may in the future have to update your code or fix something. You may even have to fix someone else's code or someone will have to work on your code. So it's a good idea to code cleanly.

**Naming** Use meaningful variable names: `record_number` instead `rn` or `n`. This is sometimes called 'self-documenting code'.

**Comments** Insert comments to explain non-trivial parts of code.

**Reuse** Do not write the same bit of code twice: use macros, functions, classes.

## 64.4 Documentation

Take a look at Doxygen.

## 64.5 Best practices: C++ Core Guidelines

The C++ language is big, and some combinations of features are not advisable. Around 2015 a number of *Core Guidelines* were drawn up that will greatly increase code quality. Note that this is not about performance: the guidelines have basically no performance implications, but lead to better code.

For instance, the guidelines recommend to use default values as much as possible when dealing with multiple constructors:

```
class Point { // not this way
private:
  double d;
public:
  Point( double x,double y,double fudge ) {
    auto d = ( x*x + y*y ) * (1+fudge); };
  Point( double x,double y ) {
    auto d = ( x*x + y*y ); };
};
```

This is bad because of code duplication. Slightly better:

```
class Point { // not this way
private:
  double d;
public:
  Point( double x,double y,double fudge ) {
    auto d = ( x*x + y*y ) * (1+fudge); };
  Point( double x,double y ) : Point(x,y,0.) {};
};
```

which wastes a couple of cycles if fudge is zero. Best:

```
class Point { // not this way
private:
  double d;
public:
  Point( double x,double y,double fudge=0. ) {
    auto d = ( x*x + y*y ) * (1+fudge); };
};
```

# Chapter 65

# Performance optimization

This section discusses performance and code optimization issues in the context of a random walk exercise.

## 65.1 Problem statement

In 1904, Sir Ronald Ross, the biologist who discovered that malaria was carried by mosquitos, gave a lecture on 'The Logical Basis of the Sanitary Policy of Mosquito Reduction'. In it, he considered the problem of how far a mosquito can fly, and therefore, how far away you need to drain all pools that harbor them.

We can model a mosquito as flying some unit distance in each time period, say, a day, of its life. However, since mosquitos fly in random directions, it will not cover a distance of $N$ in the $N$ days of its life. So how far does it get, statistically?

Ross was only able to compute this for a one-dimensional mosquito, that is, one that can only decide to go forward or backward along a line. In that case, the mosquito will on average get $\sqrt{N}$ away from where it starts.

The more general problem was brought to mathematicians' attention in 1905 by Karl Pearson, and turned out to have been solved in 1880 by Lord Rayleigh. This is now known as a *random walk* problem. Somewhat surprisingly, in 2D a mosquito also is expected to travel $\sqrt{N}$, where $N$ is the number of time steps.

The general $d$-dimensional problem is a little harder, and the mosquito travels a little less than $\sqrt{N}$. Let's code this in all generality.

## 65.2 Coding

Main program setup:

**Code:**

```
float avg_dist{0.f};
for ( int x=0; x<experiments; x++ ) {
  Mosquito m(dim);
  for (int step=0; step<steps; step++)
    m.step();
  avg_dist += m.distance();
}
avg_dist /= experiments;
```

**Output**
**[rand] vec:**

```
D=3 after 10000 steps,
    distance= 83.7997
D=3 after 100000 steps,
    distance= 224.372
D=3 after 1000000 steps,
    distance= 922.599
product took: 2776
    milliseconds
```

where the `Mosquito` class stores its position:

```
class Mosquito {
private:
  vector<float> pos;
public:
  Mosquito( int d )
    : pos( vector<float>(d,0.f) ) { };
```

and the `step` method updates this:

```
void step() {
  int d = pos.size();
  auto incr = random_step(d);
  for (int id=0; id<d; id++)
    pos.at(id) += incr.at(id);
};
```

The random step method produces a random coordinate, normalized to the unit circle. There is a slight problem here: if we generate a random coordinate in the unit cube, and normalize it, it will be biased towards the corners of the cube. Therefore, we iterate until we have a coordinate inside the unit circle, and use that to be normalized:

```
vector<float> random_coordinate( int d ) {
  auto v = vector<float>(d);
  for ( auto& e : v )
    e = random_float();
  return v;
};
```

```
vector<float> random_step(int d) {
  for (;;) {
    auto step = random_coordinate(d);
    if ( auto l=length(step); l<=1.f ) {
      if ( l==0.f ) {
        /*
         * Zero lengths can conceivably happen for d==1
         * but should not for higher d.
         */
        assert(d==1);
      } else {
        normalize(step,l);
        return step;
      }
```

```
    }
  }
};
```

**Exercise 65.1.** Take the basic code, and make a version based on

```
template<int d>
class Mosquito { /* ... */
```

How much does this simplify your code? Do you get any performance improvement?

*You can base this off the file `walk_vec.cxx` in the repository*

### 65.2.1 Optimization: save on allocation

Probably the main problem with this implementation is that each step creates multiple vectors. This sort of memory management is relatively costly, especially since very few operations are performed on each of these.

So we move the creation of the vectors outside of the computational routines. The random coordinates are now written into an array passed as parameter:

```
void random_coordinate( vector<float>& v ) {
  for ( auto& e : v )
    e = random_float();
};
```

Likewise the random step:

```
void random_step( vector<float>& step ) {
  for (;;) {
    random_coordinate(step);
```

This process of passing the arrays in stops at the `step` method, which we want to keep parameterless. So we add an option `cache` to the constructor to store the step vector as well as the position:

**Code:**

```
class Mosquito {
private:
  vector<float> pos;
  vector<float> inc;
  bool cache;
public:
  Mosquito( int d,bool cache=false )
    : pos( vector<float>(d,0.f)
    ),cache(cache) {
    if (cache) inc =
    vector<float>(d,0.f);
  };
```

**Output**
**[rand] pass:**

```
D=3 after 10000 steps,
    distance= 76.7711
D=3 after 100000 steps,
    distance= 257.19
D=3 after 1000000 steps,
    distance= 956.122
run took: 2852 milliseconds
D=3 after 10000 steps,
    distance= 87.034
D=3 after 100000 steps,
    distance= 256.655
D=3 after 1000000 steps,
    distance= 912.033
run took: 1762 milliseconds
```

```
void step() {
  int d = pos.size();
  if (cache) {
    random_step(inc);
    step( inc );
  } else {
    vector<float> incr(d);
    random_step(incr);
    step( incr );
  }
};
```

### 65.2.2 Caching in a static vector

There is still a problem with the `length` calculation. Since there is no reduction operator for 'sum of squares', we need to create a temporary vector for the squares, so that we can do a plus-reduction on it.

**Exercise 65.2.** Explore options for this temporary. Discuss what's most elegant, and measure performance improvement.

- This temporary can be passed in as a parameter;
- it can be stored in a global variable;
- or we can declare it `static`.
- With the C++20 standard, you could also use the `ranges` header.

## 65.3 Vector vs array

In this simulation, we will mostly be working in 2D, so instead of using `vector`, we can use statically allocated `array` objects. This allows for the more elegantly functional interface:

```
template<int d>
float length( const array<float,d>& step ) {
  array<float,d> square = step;
  for_each( square.begin(),square.end(),
            [] (float& x) { x *= x; } );
  auto l = sqrt
    ( std::accumulate( square.begin(),square.end(),0.f ) );
  return l;
};
```

While above we have removed all unnecessary allocation, we get an extra performance boost from optimizations from the compiler knowing the length of the array. Thus, instead of a loop of length two, the compiler will probably replace this by two explicit instructions.

**Code:**

```
float avg_dist{0.f};
for ( int x=0; x<experiments; x++ ) {
  Mosquito<dim> m;
  for (int step=0; step<steps; step++)
    m.step();
  avg_dist += m.distance();
}
avg_dist /= experiments;
```

**Output**
**[rand] arr:**

```
D=3 after 10000 steps,
    distance= 76.3221
D=3 after 100000 steps,
    distance= 247.5
D=3 after 1000000 steps,
    distance= 959.735
product took: 358
    milliseconds
```

# Chapter 66

# Tiniest of introductions to algorithms and data structures

## 66.1 Data structures

The main data structure you have seen so far is the array. In this section we briefly sketch some more complicated data structures.

### 66.1.1 Stack

A *stack* is a data structure that is a bit like an array, except that you can only see the last element:

- You can inspect the last element;
- You can remove the last element; and
- You can add a new element that then becomes the last element; the previous last element becomes invisible: it becomes visible again as the last element if the new last element is removed.

The actions of adding and removing the last element are known as *push* and *pop* respectively.

**Exercise 66.1.** Write a class that implements a stack of integers. It should have methods

```
void push(int value);
int pop();
```

### 66.1.2 Linked lists

*Before doing this section, make sure you study section 16.*

Arrays are not flexible: you can not insert an element in the middle. Instead:

- Allocate a larger array,
- copy data over (with insertion),
- delete old array storage

This is expensive. (It's what happens in a C++ `vector`; section 10.3.2.)

If you need to do lots of insertions, make a *linked list*. The basic data structure is a `Node`, which contains

1. Information, which can be anything; and
2. A pointer (sometimes called 'link') to the next node. If there is no next node, the pointer will be *null*. Every language has its own way of denoting a *null pointer*; C++ has the `nullptr`, while C uses the `NULL` which is no more than a synonym for the value zero.

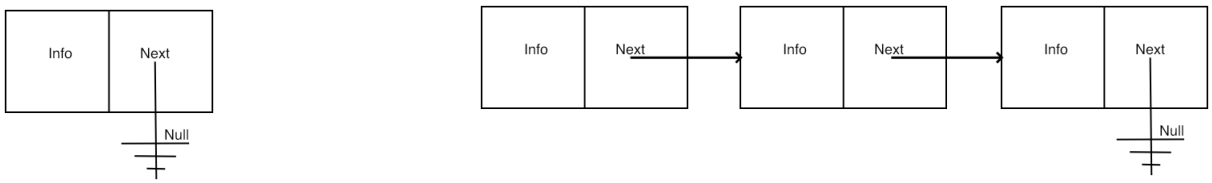



Figure 66.1: Node data structure and linked list of nodes

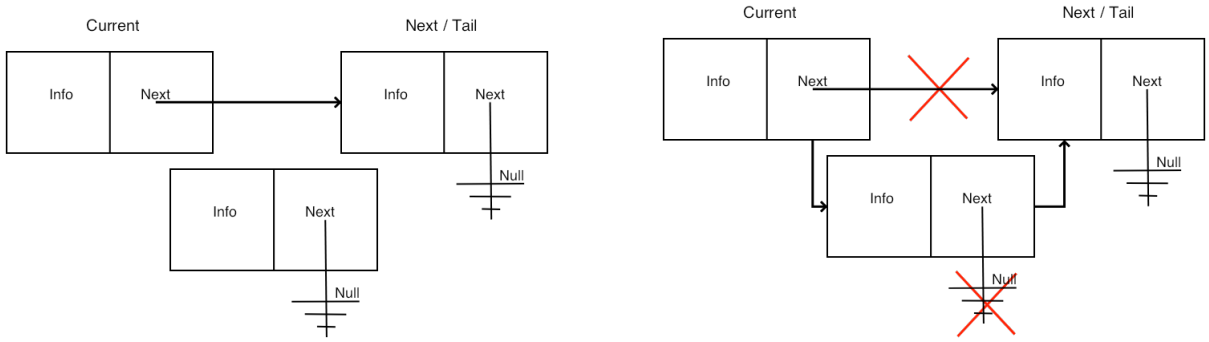



Figure 66.2: Insertion in a linked list

We illustrate this in figure 66.1.

Our main concern will be to implement operations that report some statistic of the list, such as its length, that test for the presence of information in the list, or that alter the list, for instance by inserting a new node. See figure 66.2.

#### *66.1.2.1 Data definitions*

In C++ you have a choice of pointer types. For now, we will use `shared_ptr` throughout; later we will redo our code using `unique_ptr`.

We declare the basic classes. First we declare the `List` class, which has a pointer that is null for an empty list, and pointing to the first node otherwise.

A linked list has as its only member a pointer to a node:

```
class List {
private:
  shared_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

Next, a `Node` has an information field, which we here choose to be an integer, and a counter to record how often a certain number has appeared. The Node also has a pointer to the next node. This pointer is again null for the last node in the list.

A node has information fields, and a link to another node:

```
class Node {
private:
  int datavalue{0},datacount{0};
  shared_ptr<Node> next{nullptr};
public:
  Node() {}
  Node(int value,shared_ptr<Node> next=nullptr)
    : datavalue(value),datacount(1),next(next) {};
  int value() {
    return datavalue; };
  auto nextnode() {
    return next; };
```

A Null pointer indicates the tail of the list.

We are now going to develop methods for the `List` and `Node` classes that support the following code.

List testing and modification.

```
  List mylist;
  cout << "Empty list has length: "
       << mylist.length() << '\n';

  mylist.insert(3);
  cout << "After one insertion the length is: "
       << mylist.length() << '\n';
  if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
```

Let's start with some simple functions.

### 66.1.2.2 Simple functions

For many algorithms we have the choice between an iterative and a recursive version. The recursive version is easier to formulate, but the iterative solution is probably more efficient.

We start with a simple utility function for printing a linked list. (This function is somewhat crude; a better solution uses the strategy from section 12.3.) This implementation illustrates the recursive strategy.

Auxiliary function so that we can trace what we are doing.

Print the list head:

```
void print() {
  cout << "List:";
  if (has_list())
    cout << " => "; head->print();
  cout << '\n';
};
```

Print a node and its tail:

```
void print() {
  cout << datavalue << ":" <<
    datacount;
  if (has_next()) {
    cout << ", "; next->print();
  }
};
```

For recursively computing the length of a list, we adopt this same recursive scheme.

For the list:

```
int recursive_length() {
  if (!has_list())
    return 0;
  else
    return head->listlength();
};
```

For a node:

```
int listlength_recursive() {
  if (!has_next()) return 1;
  else return 1+next->listlength_recursive();
};
```

An iterative version uses a pointer that goes down the list, incrementing a counter at every step.

Use a shared pointer to go down the list:

```
  int length_iterative() {
    int count = 0;
    if (has_list()) {
      auto current_node = head;
      while (current_node->has_next()) {
        current_node = current_node->nextnode(); count += 1;
      }
    }
    return count;
  };
```

(Fun exercise: can do an iterative de-allocate of the list?)

**Exercise 66.2.** Write a function

```
bool List::contains_value(int v);
```

to test whether a value is present in the list.

This can be done recursive and iterative.

#### *66.1.2.3 Modification functions*

The interesting methods are of course those that alter the list. Inserting a new value in the list has basically two cases:

1. If the list is empty, create a new node, and set the head of the list to that node.
2. If the list is not empty, we have several more cases, depending on whether the value goes at the head of the list, the tail, somewhere in the middle. And we need to check whether the value is already in the list.

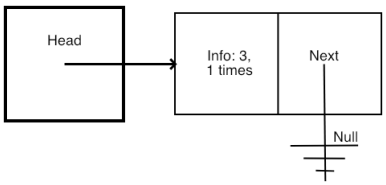
We will write functions

```
void List::insert(int value);
void Node::insert(int value);
```

that add the value to the list. The `List::insert` value can put a new node in front of the first one; the `Node::insert` assumes that the value is great equal that of the current node.

There are a lot of cases here. You can try this by an approach called Test-Driven Development (TDD): first you decide on a test, then you write the code that covers that case.

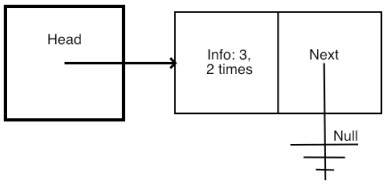
***Step 1: insert the first element*** Adding a first element to an empty list is simple: we need the pointer of the head node to point to a `Node`.




**Exercise 66.3.** Next write the case of `Node::insert` that handles the empty list. You also need a method `List::contains` that tests if an item if in the list.

```
mylist.insert(3);
cout << "After inserting 3 the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
if (mylist.contains_value(4))
  cout << "Hm. Should not contain 4" << '\n';
else
  cout << "Indeed: does not contain 4" << '\n';
cout << '\n';
```

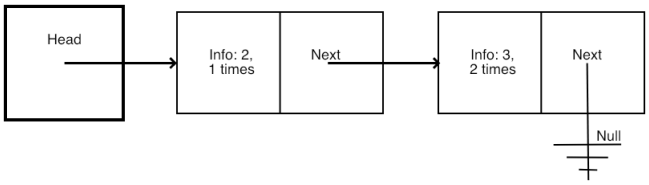
***Step 3: inserting an element that already exists*** If we try to add a value to a list that is already there, inserting does not do anything; if needed we can increment a counter in the `Node` that contains that value.




**Exercise 66.4.** Inserting a value that is already in the list means that the `count` value of a node needs to be increased. Update your `insert` method to make this code work:

```
mylist.insert(3);
cout << "Inserting the same item gives length: "
```

```
      << mylist.length() << '\n';
if (mylist.contains_value(3)) {
  cout << "Indeed: contains 3" << '\n';
  auto headnode = mylist.headnode();
  cout << "head node has value " << headnode->value()
       << " and count " << headnode->count() << '\n';
} else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

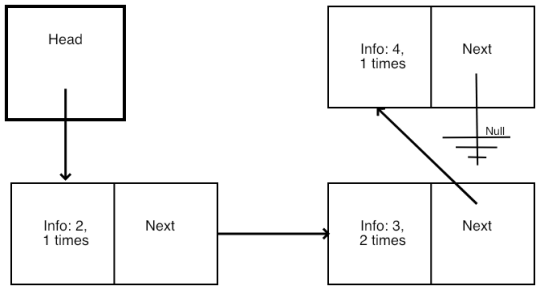



***Step 4: inserting an element at the head***

**Exercise 66.5.** One of the cases for inserting concerns an element that goes at the head. Update your *insert* method to get this to work:

```
mylist.insert(2);
cout << "Inserting 2 goes at the head;\nnow the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(2))
  cout << "Indeed: contains 2" << '\n';
else
  cout << "Hm. Should contain 2" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

***Step 5: inserting an element at the end*** Adding an element to the tail requires traversing the whole list.

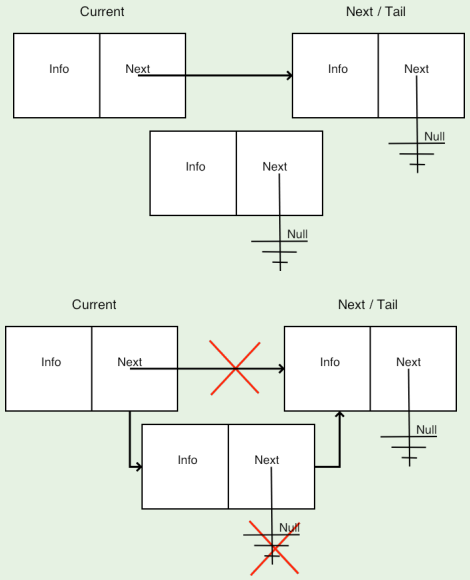



**Exercise 66.6.** If an item goes at the end of the list:

```
mylist.insert(6);
cout << "Inserting 6 goes at the tail;\nnow the length is: "
     << mylist.length()
```

```
    << '\n';
if (mylist.contains_value(6))
  cout << "Indeed: contains 6" << '\n';
else
  cout << "Hm. Should contain 6" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

***Step 6: inserting an element in the middle*** The trickiest case is inserting an element somewhere in the middle of the list. Now you need to compare the current and next element to decide whether to place the element or to move on to the tail.




**Exercise 66.7.** Update your insert routine to deal with elements that need to go somewhere in the middle.

```
mylist.insert(4);
cout << "Inserting 4 goes in the middle;\nnow the length is: "
     << mylist.length()
     << '\n';
if (mylist.contains_value(4))
  cout << "Indeed: contains 4" << '\n';
else
  cout << "Hm. Should contain 4" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

#### 66.1.2.4 Advanced: With unique pointers

Conceptually we can say that the list object owns the first node, and each node owns the next. Therefore, the most appropriate pointer type is the `unique_ptr`.

We can also do this with unique pointers:

A linked list has as its only member a pointer to a node:

```
class List {
private:
  unique_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

A node has information fields, and a link to another node:

```
class Node {
  friend class List;
private:
  int datavalue{0},datacount{0};
  unique_ptr<Node> next{nullptr};
public:
  friend class List;
  Node() {}
  Node(int value,unique_ptr<Node> tail=nullptr)
    : datavalue(value),datacount(1),next(move(tail)) {};
  ~Node() { cout << "deleting node " << datavalue << '\n'; };
```

A Null pointer indicates the tail of the list.

Above we formulated an iterative and a recursive way of computing the length of a list. The iterative code had a shared pointer pointing at successive list elements. We can not do this with unique pointers. Instead, this is a good place to use a *bare pointer*.

Use a *bare pointer*, which is appropriate here because it doesn't own the node.

```
int listlength() {
  Node *walker = next.get(); int len = 1;
  while ( walker!=nullptr ) {
    walker = walker->next.get(); len++;
  }
  return len;
};
```

(You will get a compiler error if you try to make `walker` a smart pointer: you can not copy a unique pointer.)

- Use smart pointers for ownership
- Use bare pointers for pointing but not owning.

- This is an efficiency argument.I'm not totally convinced.

### 66.1.3 Trees

*Before doing this section, make sure you study section 16.*

A tree can be defined recursively:

- A tree is empty, or
- a tree is a node with some number of children trees.

Let's design a tree that stores and counts integers: each node has a label, namely an integer, and a count value that records how often we have seen that integer.

Our basic data structure is the node, and we define it recursively to have up to two children. This is a problem: you can not write

```
class Node {
private:
  Node left,right;
}
```

because that would recursively need infinite memory. So instead we use pointers.

```
class Node {
private:
  int key{0},count{0};
  shared_ptr<Node> left,right;
  bool hasleft{false},hasright{false};
public:
  Node() {}
  Node(int i,int init=1 ) { key = i; count = 1; };
  void addleft( int value) {
    left = make_shared<Node>(value);
    hasleft = true;
  };
  void addright( int value ) {
    right = make_shared<Node>(value);
    hasright = true;
  };
  /* ... */
};
```

and we record that we have seen the integer zero zero times.

Algorithms on a tree are typically recursive. For instance, the total number of nodes is computed from the root. At any given node, the number of nodes of that attached subtree is one plus the number of nodes of the left and right subtrees.

```
int number_of_nodes() {
  int count = 1;
  if (hasleft)
    count += left->number_of_nodes();
  if (hasright)
    count += right->number_of_nodes();
  return count;
};
```

Likewise, the depth of a tree is computed as a recursive max over the left and right subtrees:

```
int depth() {
  int d = 1, dl=0,dr=0;
  if (hasleft)
    dl = left->depth();
  if (hasright)
    dr = right->depth();
  d = max(d+dl,d+dr);
  return d;
};
```

Now we need to consider how actually to insert nodes. We write a function that inserts an item at a node. If the key of that node is the item, we increase the value of the counter. Otherwise we determine whether to add the item in the left or right subtree. If no such subtree exists, we create it; otherwise we descend in the appropriate subtree, and do a recursive insert call.

```
  void insert(int value) {
    if (key==value)
      count ++;
    else if (value<key) {
      if (hasleft)
        left->insert(value);
      else
        addleft(value);
    } else if (value>key) {
      if (hasright)
        right->insert(value);
      else
        addright(value);
    } else throw(1); // should not happen
  };
```

### 66.1.4 Other graphs

The nodes in a tree have a relation from parent-to-child, so they are a special case of a *directed graph*.

**Exercise 66.8.** If you know about the relationship between graphs and their *adjacency matrix*, can you express directedness in terms of matrix properties?

An important subset or directed graphs, that of DAGs, has a property that trees do not have: for some pairs of nodes they may have more than one path between them.

For more details, see HPC book [9], chapter 24.

## 66.2 Algorithms

This *really* **really** goes beyond this book.

- Simple ones: numerical
- Connected to a data structure: search

### 66.2.1 Sorting

Unlike the tree algorithms above, which used a non-obvious data structure, sorting algorithms are a good example of the combination of very simple data structures (mostly just an array), and sophisticated analysis of the algorithm behavior. We very briefly discuss two algorithms.

The standard library has a sorting routine built-in; see section 14.2.6.

#### 66.2.1.1 Bubble sort

An array $a$ of length $n$ is sorted if

$$\forall_{i<n-1}\colon a_i \leq a_{i+1}.$$

A simple sorting algorithm suggests itself immediately: if $i$ is such that $a_i > a_{i+1}$, then reverse the $i$ and $i+1$ locations in the array.

```
void swapij( vector<int> &array,int i ) {
  int t = array[i];
  array[i] = array[i+1];
  array[i+1] = t;
}
```

(Why is the array argument passed by reference?)

If you go through the array once, swapping elements, the result is not sorted, but at least the largest element is at the end. You can now do another pass, putting the next-largest element in place, and so on.

This algorithm is known as *bubble sort*. It is generally not considered a good algorithm, because it has a time complexity (section 70.1.1) of $n^2/2$ swap operations. Sorting can be shown to need $O(n \log n)$ operations, and bubble sort is far above this limit.

#### 66.2.1.2 Quicksort

A popular algorithm that can attain the optimal complexity (but need not; see below) is *quicksort*:

- Find an element, called the pivot, that is approximately equal to the median value.
- Rearrange the array elements to give three sets, consecutively stored: all elements less than, equal, and greater than the pivot respectively.
- Apply the quicksort algorithm to the first and third subarrays.

This algorithm is best programmed recursively, and you can even make a case for its parallel execution: every time you find a pivot you can double the number of active processors.

**Exercise 66.9.** Suppose that, by bad luck, your pivot turns out to be the smallest array element every time. What is the time complexity of the resulting algorithm?

### 66.2.2 Graph algorithms

We briefly discuss some graph algorithms. For further discussion see HPC book [9], chapter 9.

First consider the SSSP problem: given a graph and a starting node, what is the shorted path to every other node. Initially, we consider an *unweighted graph*, or equivalently, set the distance between any pair of connected nodes to 1.

The algorithm proceeds by levels: each next level consists of the neighbors of already explored nodes.

- Set the distance to the starting node to zero.
- Until done,
- Loop over all nodes with known distance, and
- for all of its neighbors,
  - unless the neighbor already has a known distance
  - set the distances to $d + 1$,

We use a simple data structure for the distances:

```
map<int,int> distances;
int starting_node = 0;
distances[starting_node] = 0;
```

Until all nodes are mapped, we iterate over nodes for which the distances are know, and set the distance for their neighbors:

```
// while not done
for (;;) {
  int updates{0};
  // iterate over all mapped nodes
  for ( auto [node,dist] : distances ) {
    // for each, iterate over all of their neighbors
    }
  }
}
```

Assume that the graph data structure supports getting the list of neighbors of a node:

```
for ( const auto& neighbor : graph.neighbors(node) ) {
  if ( auto find_neighbor = distances.find(neighbor) ;
    find_neighbor==distances.end() ) {
      distances[neighbor] = dist+1; updates++;
  }
}
```

Since we use a until-done loop, we need to break out of explicitly. A simple test would be 'if the number of mapped nodes equals the number of nodes in the graph'.

**Exercise 66.10.** Can you see a problem with this, and a way to fix it?

The above algorithm is a little wasteful: in each pass it traverses all mapped nodes, but we only need the newly added ones. Therefore we add:

```
set<int> current_front;
current_front.insert(starting_node);
for (;;) {
  set<int> next_front;
  // iterate over current_front
  for ( auto node : current_front ) {
    // iterate over neighbors
```

```
    for ( const auto& neighbor : graph.neighbors(node) ) {
      // if neighbor not yet mapped
        if ( /* ... */ ) {
          distances[neighbor] = dist_to_this_node+1;
          next_front.insert(neighbor);
        }
    }
  }
  current_front = next_front;
}
```

## 66.3 Programming techniques

### 66.3.1 Memoization

In section 7.6 you saw some examples of recursion. The factorial example could be written in a loop, and there are both arguments for and against doing so.

The Fibonacci example is more subtle: it can not immediately be converted to an iterative formulation, but there is a clear need for eliminating some waste that comes with the simple recursive formulation. The technique we can use for this is known as *memoization*: store intermediate results to prevent them from being recomputed.

Here is an outline.

```
int fibonacci(int n) {
  vector<int> fibo_values(n);
  for (int i=0; i<n; i++)
    fibo_values[i] = 0;
  fibonacci_memoized(fibo_values,n-1);
  return fibo_values[n-1];
}
int fibonacci_memoized( vector<int> &values, int top ) {
  int minus1 = top-1, minus2 = top-2;
  if (top<2)
    return 1;
  if (values[minus1]==0)
    values[minus1] = fibonacci_memoized(values,minus1);
  if (values[minus2]==0)
    values[minus2] = fibonacci_memoized(values,minus2);
  values[top] = values[minus1]+values[minus2];
  //cout << "set f(" << top << ") to " << values[top] << '\n';
  return values[top];
}
```

# Chapter 67

# Provably correct programs

Programming often seems more an art, even a black one, than a science. Still, people have tried systematic approaches to program correctness. One can distinguish between

- proving that a program is correct, or
- writing a program so that it is guaranteed to be correct.

This distinction is only imaginary. A more fruitful approach is to let the proof drive the coding. As E.W. Dijkstra pointed out

> The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

We will see a couple of examples of this.

## 67.1 Loops as quantors

Quite often, algorithms can be expressed mathematically. In that case you should make your program look like the mathematics.

### 67.1.1 Forall-quantor

Consider a simple example: testing if a number is prime. The predicate 'isprime' can be expressed as:

$$\text{isprime}(n) \equiv \forall_{2 \leq f < n} : \neg\, \text{divides}(f, n)$$

You see that proving the original `isprime` predicate for some value $n$ now involves

1. a quantor over a new variable $f$ – we call this a 'bound variable';
2. and a new predicate `divides` that involves the original variable $n$ and the variable $f$ that is bound by the quantor.

We now spell out the 'for all' quantor iteratively as a loop where each iteration needs to be true. That is, we do an 'and' reduction on some iteration-dependent result.

$$\neg\, \text{divides}(2, n) \cap \ldots \cap \neg\, \text{divides}(n-1, n)$$

And this sequence of 'and' conjunctions can be programmed:

```
for (int f=2; f<n; f++)
  isprime = isprime && not divides(f,p)
```

You notice that the loop variable is the variable $f$ that was introduced by the quantor.

Now our only worry is how to initialize `isprime`. The initial value corresponds to an 'and' conjunction over an empty set, which is true, so:

```
bool isprime{true};
for (int f=2; f<n; f++)
  isprime = isprime && not divides(f,p)
```

**Exercise 67.1.** Now that you have a loop that computes the right thing you can start worrying about performance. Can the loop be terminated prematurely in some cases? How would you program that?

### 67.1.2 Thereis-quantor

What if we had expressed primeness as:

$$\text{isprime}(n) \equiv \neg\exists_{2\leq f<n}\colon \text{divides}(f,n)$$

To get a pure quantor, and not a negated one, we write:

$$\text{isnotprime}(n) \equiv \exists_{2\leq f<n}\colon \text{divides}(f,n)$$

Spelling out the exists-quantor as

$$\text{isnotprime}(n) \equiv \text{divides}(2,n) \cup \ldots \cup \text{divides}(n-1,n)$$

we see that we need a loop where we test if any iteration satisfies a predicate. That is, we do an 'or'-reduction on the results of each iteration. As before, the loop variable is the variable introduced by the quantor.

```
for (int f=2; f<n; f++)
  isnotprime = isnotprime or divides(f,p)
bool isprime = not isnotprime;
```

Also as before, we take care to initialize the reduction variable correctly: applying $\exists_{s\in S}P(s)$ over an empty set $S$ is `false`:

```
bool isnotprime{false};
for (int f=2; f<n; f++)
  isnotprime = isnotprime or divides(f,p)
bool isprime = not isnotprime;
```

**Exercise 67.2.** Same question as for the 'forall' quantor: can this loop be terminated prematurely? How would you code that?

## 67.2 Predicate proving

For programs that have a clear loop structure you can take an approach that is similar to doing a 'proof by induction'.

Let us consider the Collatz conjecture again, where for brevity we define

$$c(\ell) = \text{ the length of the Collatz sequences, starting on } \ell.$$

Now we consider the Collatz conjecture as proving a predicate

$$P(\ell_k, m_k, k) = \begin{cases} \ell_k < k \text{ is the location of the longest sequence:} \\ c(\ell_k) = m_k \text{: the length of sequence } \ell_k \text{ is } m_k \\ \text{all other sequences } \ell < k \text{ are shorter} \end{cases}$$

Formally:

$$P(\ell_k, m_k, k) = \left[ \begin{array}{ll} & \ell_k < k \\ \wedge & c(\ell_k) = m_k \text{(only if } k > 0) \\ \wedge & \forall_{\ell<k} \colon c(\ell) \leq m_k \end{array} \right.$$

for $k = N$.

We develop the code that makes this predicate inductively true. We start out with

$$\ell_0 = -1, \quad m_0 = 0 \Rightarrow P(\ell_0, m_0, 0).$$

The inductive proof corresponds to a loop:

- we assume that at the start of the $k$-th iteration $P(\ell_k, m_k, k)$ is true;
- the iteration body is such that at the end of the $k$-th iteration $P(\ell_{k+1}, m_{k+1}, k+1)$ is true;
- this of course sets up the predicate at the start of the next iteration.

The loop structure is then:

```
k=0;
{P(l_k, m_k, k)}
while ( k<N ) {
  {P(l_k, m_k, k)}
  update;
  {P(l_{k+1}, m_{k+1}, k+1)}
  k = k+1;
}
```

The update has to extend the predicate from $k$ to $k+1$. Let us consider the parts of it.

We need to establish

$$\forall_{\ell<k+1} \colon c(\ell) \leq m_{k+1}$$

We split the range $\ell < k+1$ into $\ell < k$ and $\ell = k$:

- the first part

  $$\forall_{\ell<k} \colon c(\ell) \leq m_{k+1}$$

  is true if $m_{k+1} \geq m_k$;

- the part

$$\ell = k \colon c(\ell) \leq m_{k+1}$$

states that $m_{k+1} \geq c(k)$.

Together we get that

$$m_{k+1} \geq \max(m_k, c(k))$$

Finally, the clause

$$c(\ell_{k+1}) = m_{k+1}$$

can be satisfied:

- If $c(k) > m_k$, we need to set $m_{k+1} = c(k)$ and $\ell_{k+1} = k$.
- (Strictly speaking, there is a possibility $m_{k+1} > c(k)$. This is not possible, because we can not satisfy $m_{k+1} = c(\ell_k)$ for any $k$.)
- If $c(k) \leq m_k$, we need to set $m_{k+1} \geq m_k$. Again, $m_{k+1} > m_k$ can not be satisfied by any $\ell_{k+1}$, so we conclude $m_{k+1} = m_k$.

## 67.3 Flame

Above, you saw a Dijkstra quote where he argues that testing is insufficient to show correctness of a program. So how does Dijkstra envision that correctness can be ensured? That can be found in the second part of his quote:

> The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

Let us develop this though, using matrix-vector multiplication as a simple example: we will derive the algorithm hand-in-hand with its correctness proof.

Many linear algebra algorithms are loop-based, and the foundation to a correctness proof is the derivation of a *loop invariant*: a predicate that is inductively shown to be true in each loop iteration, thereby guaranteeing the correctness of the whole algorithm.

### 67.3.1 Derivation of the common algorithm

As a preliminary to deriving a loop invariant, we first consider the result of the computation in a partitioned form.

$$y = Ax$$

Partitioned:

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Two equations:

$$\begin{cases} y_T = A_T x \\ y_B = A_B x \end{cases}$$

The key to the inductive proof is to take this partitioned form, and assume that it is only partly satisfied.

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Assume only equation

$$y_T = A_T x$$

is satisfied.

Now we are getting close to an inductive proof: we consider the algorithm as aimed at increasing the size of the block for which the predicate is true. If this block equals the size of the problem, we have correctness of the full result.

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

While $T$ is not the whole system
    Predicate: $y_T = A_T x$ true
    Update: grow $T$ block by one
    Predicate: $y_T = A_T x$ true for new/bigger $T$ block

Note initial and final condition.

Now we compare the true statement in one iteration, and that in the next, and compare the two blocks for which the predicate holds.

Here is the big trick
Before

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

split:

$$\begin{pmatrix} y_1 \\ \cdots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \cdots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

Then the update step, and
After

$$\begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \cdots \\ A_3 \end{pmatrix} (x)$$

and unsplit

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Comparing these two blocks gives us the extra predicate that was made to be satisfied by the iteration we consider:

Before the update:

$$\begin{pmatrix} y_1 \\ \cdots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \cdots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

so

$$y_1 = A_1 x$$

is true
Then the update step, and
After

$$\begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \cdots \\ A_3 \end{pmatrix} (x)$$

so

$$\begin{cases} y_1 = A_1 x & \text{we had this} \\ y_2 = A_2 x & \text{we need this} \end{cases}$$

This extra predicate can trivially be converted to an elementary computation, and so we find the full algorithm:

While $T$ is not the whole system
    Predicate: $y_T = A_T x$ true
    Update: $y_2 = A_2 x$
    Predicate: $y_T = A_T x$ true for new/bigger $T$ block

#### 67.3.1.1 Derivation of the algorithm by columns

In the previous section we derived the matrix-vector product, expressed the usual way: each element of the output vector is the inner product of a matrix row and the input vector. You may think that this is a lot of to-do for not much result.

Consider then that we can derive other algorithms for the matrix-vector product, using the same general principle. Our basic assumption was that we split the matrix horizontally in two block rows. What would happen if we split the matrix vertically in two block columns?

We divide the matrix into two block columns, and express the result of the matrix-vector product on this form.

$$y = Ax$$

Partitioned:

$$\begin{pmatrix} y \end{pmatrix} = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Equation:

$$\left\{ y = A_L x_T + A_R x_B \right.$$

Again we make a statement about a partially completed computation.

$$\begin{pmatrix} y \end{pmatrix} = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Assume

$$y = A_L x_T$$

is constructed, and grow the $T$ block.

Again we compare splitting the matrix in one iteration and the next, and comparing the partial predicates:

Before

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

split:

$$(y) = \left(\begin{array}{c|cc} A_1 & A_2 & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

Then the update step, and
After

$$(y) = \left(\begin{array}{cc|c} A_1 & A_2 & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

and unsplit

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

This gives us by 'subtracting' one predicate from the other the extra result that was derived by the single iteration, and therefore the computation that was done in that iteration.

Before the update:

$$(y) = \left(\begin{array}{c|cc} A_1 & A_2 & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1$$

is true
Then the update step, and
After

$$(y) = \left(\begin{array}{cc|c} A_1 & A_2 & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1 + A_2 x_2$$

in other words, we need

$$y \leftarrow y + A_2 x_2$$

As a result we obtain a second form of the matrix-vector product.

While $T$ is not the whole system
    Predicate: $y = A_L x_T$ true
    Update: $y \leftarrow y + A_2 x_2$
    Predicate: $y = A_L x_T$ true for new/bigger $T$ block

In quasi-Matlab notation we express both algorithms:

for $r = 1, m$
    $y_r = A_{r,*} x_*$

$y \leftarrow 0$
for $c = 1, n$
    $y \leftarrow y + A_{*,c} x_c$

# Chapter 68

# Unit testing and Test-Driven Development

In an ideal world, you would prove your program correct, but in practice that is not always feasible, or at least: not done. Most of the time programmers establish the correctness of their code by testing it.

Yes, there is a quote by *Edsger Dijkstra* that goes:

> Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (cue laughter)

but that doesn't mean that you can't at least gain some confidence in your code by testing it.

## 68.1 Types of tests

Testing code is an art, even more than writing the code to begin with. That doesn't mean you can't be systematic about it. First of all, we distinguish between some basic types of test:

- *Unit tests* that test a small part of a program by itself;
- *System tests* test the correct behavior of the whole software system; and
- *Regression tests* establish that the behavior of a program has not changed by adding or changing aspects of it.

In this section we will talk about unit testing.

A program that is written in a sufficiently modular way allows for its components to be tested without having to wait for an all-or-nothing test of the whole program. Thus, testing and program design are aligned in their interests. In fact, writing a program with the thought in mind that it needs to be testable can lead to cleaner, more modular code.

In an extreme form of this you would write your code by Test-Driven Development (TDD), where code development and testing go hand-in-hand. The basic principles can be stated as follows:

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

In a strict interpretation, you would even for each part of the program first write the test that it would satisfy, and then the actual code.

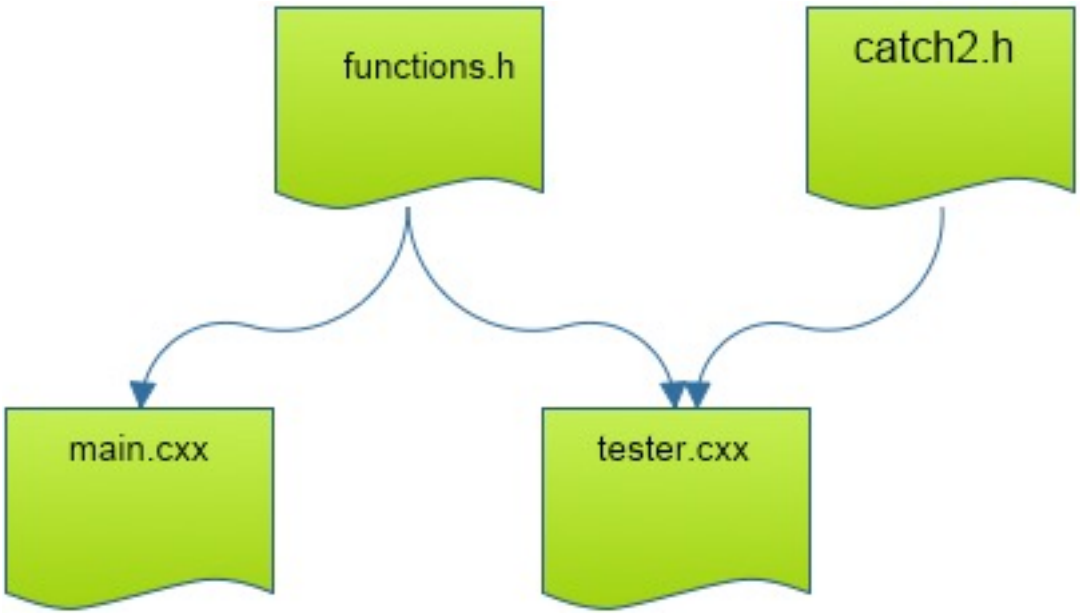



Figure 68.1: File structure for unit tests

## 68.2 Unit testing frameworks

There are several 'frameworks' that help you with unit testing. In the remainder of this chapter we will use *Catch2*, which is one of the most used ones in C++.

You can find the code at https://github.com/catchorg.

### 68.2.1 File structure

Let's assume you have a file structure with

- a very short main program, and
- a library file that has all functions used by the main.

In order to test the functions, you supply another main, which contains only unit tests; This is illustrated in figure 68.1.

In fact, with Catch2 your main file doesn't actually have a `main` program: that is supplied by the framework. In the tester main file you only put the test cases.

The framework supplies its own main:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

#include "library_functions.h"
/*
   here follow the unit tests
*/
```

One important question is what header file to include. You can do

```
#include "catch.hpp"
```

which is the 'header only' mode, but that makes compilation very slow. Therefore, we will assume you have installed Catch through *Cmake*, and you include

```
#include "catch2/catch_all.hpp"
```

Note: as of September 2021 this requires the development version of the repository, not any `2.x` release.

### 68.2.2 Compilation

The setup suggested above requires you to add compile and link flags to your setup. This is system-dependent.

One-line solution:

```
icpc -o tester test_main.cxx \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

Variables for a Makefile:

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2Main -lCatch2
```

### 68.2.3 Test cases

A test case is a short program that is run as an independent main. In the setup suggested above, you put all your unit tests in the tester main program, that is, the file that has the

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"
```

magic lines.

Each test case needs to have a unique name, which is printed when a test fails. You can optionally add keys to the test case that allow you to select tests from the commandline.

```
TEST_CASE( "name of this test" ) {
  // stuf
}
TEST_CASE( "name of this test","[key1][key2]" ) {
  // stuf
}
```

The body of the test case is essentially a main program, where some statements are encapsulated in test macros. The most common macro is *REQUIRE*, which is used to demand correctness of some condition.

Tests go in `tester.cxx`:

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

- `TEST_CASE` acts like independent main program.
  can have multiple cases in a tester file
- `REQUIRE` is like `assert` but more sophisticated

**Exercise 68.1.**

1. Write a function

   ```
   double f(int n) { /* .... */ }
   ```

   that takes on positive values only.
2. Write a unit test that tests the function for a number of values.

*You can base this off the file `tdd.cxx` in the repository*

Boolean:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Integer:

```
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!=0 );
```

Beware floating point:

```
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!=Catch::Approx(1.0) );
```

In general exact tests don't work.

For failing tests, the framework will give the name of the test, the line number, and the values that were tested.

Run the tester:

```
-------------------------------
test the increment function
-------------------------------
test.cxx:25
...............................

test.cxx:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2
```

```
      ================================
      test cases: 1 | 1 failed
      assertions: 1 | 1 failed
```

In the above case, the error message printed out the offending value of `f(n)`, not the value of `n` for which it occurs. To determine this, insert `INFO` specifications, which only get print out if a test fails.

`INFO`: print out information at a failing test

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    INFO( "function fails for " << n );
    REQUIRE( f(n)>0 );
}
```

If your code throws exceptions (section 23.2.2) you can test for these.

Suppose function `g(n)`

- succeeds for input $n > 0$
- fails for input $n \leq 0$:
  throws exception

```
TEST_CASE( "test that g only works for positive" ) {
  for (int n=-100; n<+100; n++)
    if (n<=0)
      REQUIRE_THROWS( g(n) );
    else
      REQUIRE_NOTHROW( g(n) );
}
```

A common occurrence in unit testing is to have multiple tests with a common setup or tear down, to use terms that you sometimes come across in unit testing. Catch2 supports this: you can make 'sections' for part in between setup and tear down.

Use `SECTION` if tests have intro/outtro in common:

```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

(sometimes called setup/teardown)

## 68.3 Example: zero-finding by bisection

Development of the zero-finding algorithm by bisection can be found in section 48.1.

## 68.4 An example: quadratic equation roots

We revisit exercise 24.4, which used `std::variant` to return 0,1,2 roots of a quadratic equation. Here we use TDD to arrive at the code.

Throughout, we represent the polynomial

$$ax^2 + bx + c$$

as

```
using quadratic = tuple<double,double,double>;
```

When needed, you can unpack this again with

```
auto [a,b,c] = coefficients;
```

(Can you think of an `assert` statement here that might be useful?)

**Exercise 68.2.** Write a function

```
double discriminant( quadratic coefficients );
```

that computes $b^2 - 4ac$, and test:

```
TEST_CASE( "discriminant" ) {
  REQUIRE( discriminant( make_tuple(0., 2.5, 0.) ) ==Catch::Approx(6.25) );
  REQUIRE( discriminant( make_tuple(1., 0., 1.5 ) ) ==Catch::Approx(-6.) );
  REQUIRE( discriminant( make_tuple(.1, .1, .1*.5 ) ) ==Catch::Approx(-.01) );
}
```

It may be illustrative to see what happens if you leave out the approximate equality test:

```
REQUIRE( discriminant( make_tuple(.1, .1, .1*.5 ) ) == -.01 );
```

With this function it becomes easy to detect the case of no roots: the discriminant $D < 0$. Next we need to have the criterium for single or double roots: we have a single root if $D = 0$.

**Exercise 68.3.** Write a function

```
bool discriminant_zero( quadratic coefficients );
```

that passes the test

```
quadratic coefficients = make_tuple(a,b,c);
d = discriminant( coefficients );
z = discriminant_zero( coefficients );
INFO( a << "," << b << "," << c << " d=" << d );
REQUIRE( z );
```

Using for instance the values:

```
a = 2; b = 4; c = 2;
a = 2; b = sqrt(40); c = 5; // !!!
a = 3; b = 0; c = 0.;
```

This exercise is the first one where we run into numerical subtleties. The second set of test values has the discriminant zero in exact arithmetic, but nonzero in computer arithmetic. Therefore, we need to test whether it is small enough, compared to $b$.

**Exercise 68.4.** Be sure to also test the case where *discriminant_zero* returns false.

Now that we've detected a single root, we need the function that computes it. There are no subtleties in this one.

**Exercise 68.5.** Write the function *simple_root* that returns the single root. For confirmation, test

```
auto r = simple_root(coefficients);
REQUIRE( evaluate(coefficients,r)==Catch::Approx(0.).margin(1.e-14) );
```

The remaining case of two distinct roots is arrived at by elimination, and the only thing to do is write the function that returns them.

**Exercise 68.6.** Write a function that returns the two roots as a indexcstdpair:

```
pair<double,double> double_root( quadratic coefficients );
```

Test:

```
quadratic coefficients = make_tuple(a,b,c);
auto [r1,r2] = double_root(coefficients);
auto
  e1 = evaluate(coefficients,r1),
  e2 = evaluate(coefficients,r2);
REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14) );
REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14) );
```

The final bit of code is the function that tests for how many roots there are, and returns them as a `std::variant`.

**Exercise 68.7.** Write a function

```
variant< bool,double, pair<double,double> >
    compute_roots( quadratic coefficients);
```

Test:

```
TEST_CASE( "full test" ) {
  double a,b,c; int index;
  SECTION( "no root" ) {
    a=2.0; b=1.5; c=2.5;
    index = 0;
  }
  SECTION( "single root" ) {
    a=1.0; b=4.0; c=4.0;
    index = 1;
  }
  SECTION( "double root" ) {
    a=2.2; b=5.1; c=2.5;
    index = 2;
  }
  quadratic coefficients =
    make_tuple(a,b,c);
  auto result =
    compute_roots(coefficients);
  REQUIRE( result.index()==index );
}
```

## 68.5 Eight queens example

See 49.3.

# Chapter 69

# Debugging with gdb

## 69.1 A simple example

The following program does not have any bugs; we use it to show some of the basics of gdb.

```
void say(int n) {
  cout << "hello world " << n << '\n';
}

int main() {

  for (int i=0; i<10; i++) {
    int ii;
    ii = i*i;
    ii++;
    say(ii);
  }

  return 0;
}
```

### 69.1.1 Invoking the debugger

After you compile your program, instead of running it the normal way, you invoke *gdb*:

```
gdb myprogram
```

That puts you in an environment, recognizable by the `(gdb)` prompt:

```
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
[stuff]
(gdb)
```

where you can do a controlled run of your program with the *run* command:

```
(gdb) run
Starting program: /home/eijkhout/gdb/hello
hello world 1
hello world 2
hello world 5
```

```
hello world 10
hello world 17
hello world 26
hello world 37
hello world 50
hello world 65
hello world 82
[Inferior 1 (process 30981) exited normally]
```

## 69.2 Example: integer overflow

The following program shows *integer overflow*. (We are using `short` to force this to happen soon.)

**Code:**

```
1  void say(short n) {
2    cout << "hello world " << n << '\n';
3  }
4
5  int main() {
6
7    for (short i=0; ; i+=20) {
8      short ii;
9      ii = i*i;
10     ii++;
11     say(ii);
12   }
13
14   return 0;
15 }
```

**Output**
**[gdb] hello:**

```
hello world 1
hello world 401
hello world 1601
hello world 3601
hello world 6401
hello world 10001
hello world 14401
hello world 19601
hello world 25601
hello world 32401
hello world -25535
hello world -17135
hello world -7935
hello world 2065
hello world 12865
hello world 24465
hello world -28671
hello world -15471
hello world -1471
hello world 13329
```

## 69.3 More gdb

### 69.3.1 Run with commandline arguments

This program is self-contained, but if you had a program that takes *commandline arguments*:

```
./myprogram 25
```

you can supply those in gdb:

```
(gdb) run 25
```

### 69.3.2 Source listing and proper compilation

Inside gdb, you can get a source listing with the *list* command.

Let's try our program again:

```
[] icpc -o hello hello.cxx
[] gdb hello
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
Copyright (C) 2013 Free Software Foundation, Inc.
Reading symbols from /home/eijkhout/gdb/hello...
    (no debugging symbols found)...
done.
(gdb) list
No symbol table is loaded.  Use the "file" command.
```

See the repeated reference to 'symbols'? You need to supply the `-g` compiler option for the *symbol table* to be included in the binary:

```
[] icpc -g -o hello hello.cxx
[] gdb hello
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
[stuff]
Reading symbols from /home/eijkhout/gdb/hello...done.
(gdb) list
13      using std::cout;
14      using std::endl;
[et cetera]
```

(If you hit return now, the list command is repeated and you get the next block of lines. Doing `list -` gives you the block above where you currently are.)

### 69.3.3 Stepping through the source

Let's now make a more controlled run of the program. In the source, we see that line 22 is the first executable one:

```
20      int main() {
21
22        for (int i=0; i<10; i++) {
23          int ii;
24          ii = i*i;
...
```

We introduce a *breakpoint* with the *break* command:

```
(gdb) break 22
Breakpoint 1 at 0x400a03: file hello.cxx, line 22.
```

(If your program is spread over multiple files, you can specify the file name: `break otherfile.cxx:34`.)

Now if we run the program, it will stop at that line:

```
(gdb) run
Starting program: /home/eijkhout/gdb/hello

Breakpoint 1, main () at hello.cxx:22
22        for (int i=0; i<10; i++) {
```

To be precise: the program is stopped in the state before it executes this line.

We can now use *cont* (for 'continue') to let the program run on. Since there are no further breakpoints, the program will run to completion. This is not terribly useful, so let us change our minds about the location of the breakpoint: it would be more useful if the execution stopped at the start of every iteration.

Recall that the breakpoint had a number of 1, so we use *delete* to remove it, and we set a breakpoint inside the loop body instead, and continue until we hit it.

```
(gdb) delete 1
(gdb) break 23
Breakpoint 2 at 0x400a29: file hello.cxx, line 23.
(gdb) cont
Continuing.
Breakpoint 2, main () at hello.cxx:24
24          ii = i*i;
```

(Note that line 23 is not executable, so execution stops on the first line after that.)

Now if we continue, the program runs until the next break point:

```
(gdb) cont
Continuing.
hello world 5

Breakpoint 1, main () at hello.cxx:24
24          ii = i*i;
```

To get to the next statement, we use *next*:

```
(gdb) next
25          ii++;
(gdb)
```

Hitting return re-executes the previous command, so we go to the next line:

```
(gdb)
26          say(ii);
(gdb)
hello world 2

Breakpoint 1, main () at hello.cxx:24
24          ii = i*i;
```

You observe that the function call

1. is executed, as is clear from the `hello world 1` output, but
2. is not displayed in detail in the debugger.

The conclusion is that `next` goes to the next executable statement in the current subprogram, not into functions and such that get called from it.

If you want to go into the function `say`, you need to use *`step`*:

```
(gdb) next
25          ii++;
(gdb) next
26          say(ii);
(gdb) step
say (n=10) at hello.cxx:17
17        cout << "hello world " << n << endl;
```

The debugger reports the function name, and the names and values of the arguments. Another 'step' executes the current line and brings us to the end of the function, and the next 'step' puts us back in the main program:

```
(gdb)
hello world 10
18      }
(gdb)
main () at hello.cxx:24
24          ii = i*i;
```

### 69.3.4 Inspecting values

When execution is stopped at a line (remember, that means right before it is executed!) you can inspect any values in that subprogram:

```
24          ii = i*i;
(gdb) print i
$1 = 4
```

You can even let expressions be evaluated with local variables:

```
(gdb) print 2*i
$2 = 8
```

You can combine this looking at values with breakpoints. Say you want to know when the variable `ii` gets more than 40:

```
(gdb) break 26 if ii>40
Breakpoint 1 at 0x4009cd: file hello.cxx, line 26.
(gdb) run
Starting program: /home/eijkhout/intro-programming-private/code/gdb/hello
hello world 1
```

```
hello world 2
hello world 5
hello world 10
hello world 17
hello world 26
hello world 37

Breakpoint 1, main () at hello.cxx:26
26          say(ii);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x8
(gdb) print i
$1 = 7
```

### 69.3.5 A NaN example

The following program:

```
17      float root(float n)
18      {
19        float r;
20        float n1 = n-1.1;
21        r = sqrt(n1);
22        return r;
23      }
24
25      int main() {
26        float x=9,y;
27        for (int i=0; i<20; i++) {
28          y = root(x);
29          cout << "root: " << y << endl;
30          x -= 1.1;
31        }
32
33        return 0;
34      }
```

prints some numbers that are 'not-a-number':

```
[] ./root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214
root: -nan
root: -nan
root: -nan
```

Suppose you want to figure out why this happens.

The line that prints the 'nan' is 29, so we want to set a breakpoint there, and preferably a conditional breakpoint. But how do you test on 'nan'? This takes a little trick.

```
(gdb) break 29 if y!=y
Breakpoint 1 at 0x400ea6: file root.cxx, line 28.
(gdb) run
Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 1, main () at root.cxx:29
29          cout << "root: " << y << endl;
```

We discover what iteration this happens:

```
(gdb) print i
$1 = 8
```

so now we can rerun the program, and investigate that particular iteration:

```
(gdb) break 28 if i==8
Breakpoint 2 at 0x400eaf: file root.cxx, line 28.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 2, main () at root.cxx:28
28          y = root(x);
```

We now go into the `root` routine to see what is going wrong there:

```
(gdb) step
root (n=0.200000554) at root.cxx:20
20          float n1 = n-1.1;
(gdb)
21          r = sqrt(n1);
(gdb) print n
$2 = 0.200000554
(gdb) print n1
$3 = -0.89999944
(gdb) next
22          return r;
(gdb) print r
$4 = -nan(0x400000)
```

And there we have the problem: our input `n` is used to compute another number `n1` of which we compute the square root, and sometimes this number gets negative.

### 69.3.6 Assertions

Instead of running a program and debugging it if you happen to spot a problem (and note that this may not always be the case!) you can also make your program more robust by including *assertions*. These are things that you know should be true, from your knowledge of the problem you are solving.

For instance, in the previous example there was a square root function, and you just 'knew' that the input was always going to be positive. So you edit your program as follows:

```
// header to allow assertions:
#include <cassert>

float root(float n)
{
  float r;
  float n1 = n-1.1;
  assert(n1>=0); // NOTE!
  r = sqrt(n1);
  return r;
}
```

Now if you run your program, you get:

```
[] ./assert
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214
```

```
assert: assert.cxx:22: float root(float): Assertion 'n1>=0' failed.
Aborted (core dumped)
```

What does this give you?

- It only tells you that an assertion failed, not with what values;
- it does not give you a traceback or so; on the other hand
- assertions can help you detect error conditions that you might otherwise have overlooked!

# Chapter 70

# Complexity

## 70.1 Order of complexity

### 70.1.1 Time complexity

**Exercise 70.1.** For each number $n$ from 1 to 100, print the sum of all numbers 1 through $n$.

There are several possible solutions to this exercise. Let's assume you don't know the formula for the sum of the numbers $1 \ldots n$. You can have a solution that keeps a running sum, and a solution with an inner loop.

**Exercise 70.2.** How many operations, as a function of $n$, are performed in these two solutions?

### 70.1.2 Space complexity

**Exercise 70.3.** Read numbers that the user inputs; when the user inputs zero or negative, stop reading. Add up all the positive numbers and print their average.

This exercise can be solved by storing the numbers in a `std::vector`, but one can also keep a running sum and count.

**Exercise 70.4.** How much space do the two solutions require?

# PART VI

# INDEX AND SUCH

# Chapter 71

# Index of C++ keywords

# Chapter 72

## Index of Fortran keywords

# Chapter 73

# Bibliography


[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11. IEEE Computer Society Press, Los Alamitos, California, 1990. 476

[2] Roy M. Anderson and Robert M. May. Population biology of infectious diseases: Part I. *Nature*, 280:361–367, 1979. doi:10.1038/280361a0. 447

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997. 306

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc webpage. http://www.mcs.anl.gov/petsc, 2011. 306

[5] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006. 472

[6] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. 502

[7] Yen-Lin Chen, Chuan-Yen Chiang, Yo-Ping Huang, and Shyan-Ming Yuan. A project-based curriculum for teaching C++ object-oriented programming. In *Proceedings of the 2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*, UIC-ATC '12, pages 667–672, Washington, DC, USA, 2012. IEEE Computer Society. 29

[8] Victor Eijkhout. HPC carpentry. `https://theartofhpc.com/carpentry.html`. 415

[9] Victor Eijkhout. The science of computing. `http://theartofhpc.com/istc.html`. 20, 44, 272, 289, 290, 291, 438, 457, 472, 476, 479, 487, 488, 489, 542, 543

[10] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling 17, page 20?29, New York, NY, USA, 2017. Association for Computing Machinery. 29

[11] Google. Google developer documentation style guide. 415

[12] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008. 480

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. 476

[14] Harry L. Reed. Firing table computations on the eniac. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM '52, page 103?106, New York, NY, USA, 1952. Association for Computing Machinery. 497

[15] Juan M. Restrepo, , and Michael E. Mann. This is how c̈limate is always changing.̈ *APS Physics, GPS newletter*, February 2018. 505

[16] Lin S. and Kernighan B. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973. 471