<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.5</td>
<td>C strings</td>
<td>156</td>
</tr>
<tr>
<td>12</td>
<td><strong>Input/output</strong></td>
<td>157</td>
</tr>
<tr>
<td>12.1</td>
<td>Screen output</td>
<td>157</td>
</tr>
<tr>
<td>12.1.1</td>
<td>Floating point output</td>
<td>160</td>
</tr>
<tr>
<td>12.1.2</td>
<td>Saving and restoring settings</td>
<td>161</td>
</tr>
<tr>
<td>12.2</td>
<td><strong>File output</strong></td>
<td>162</td>
</tr>
<tr>
<td>12.3</td>
<td><strong>Output your own classes</strong></td>
<td>162</td>
</tr>
<tr>
<td>12.4</td>
<td><strong>Output buffering</strong></td>
<td>163</td>
</tr>
<tr>
<td>12.4.1</td>
<td>The need for flushing</td>
<td>163</td>
</tr>
<tr>
<td>12.4.2</td>
<td>Performance considerations</td>
<td>163</td>
</tr>
<tr>
<td>12.5</td>
<td><strong>Input</strong></td>
<td>164</td>
</tr>
<tr>
<td>12.5.1</td>
<td>File input</td>
<td>164</td>
</tr>
<tr>
<td>12.5.2</td>
<td>Input streams</td>
<td>165</td>
</tr>
<tr>
<td>13</td>
<td>Lambda functions</td>
<td>167</td>
</tr>
<tr>
<td>13.1</td>
<td>Passing lambdas around</td>
<td>168</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Lambda members of classes</td>
<td>168</td>
</tr>
<tr>
<td>13.2</td>
<td>Captures</td>
<td>169</td>
</tr>
<tr>
<td>13.3</td>
<td>Capture by reference</td>
<td>169</td>
</tr>
<tr>
<td>13.4</td>
<td>More</td>
<td>172</td>
</tr>
<tr>
<td>13.4.1</td>
<td>Making lambda stateful</td>
<td>172</td>
</tr>
<tr>
<td>13.4.2</td>
<td>Generic lambdas</td>
<td>172</td>
</tr>
<tr>
<td>13.4.3</td>
<td>Algorithms</td>
<td>172</td>
</tr>
<tr>
<td>13.4.4</td>
<td>C-style function pointers</td>
<td>173</td>
</tr>
<tr>
<td>14</td>
<td>Iterators, Algorithms, Ranges</td>
<td>175</td>
</tr>
<tr>
<td>14.1</td>
<td>Iterators</td>
<td>175</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Iterators</td>
<td>175</td>
</tr>
<tr>
<td>14.1.2</td>
<td>Simple iterating, the hard way</td>
<td>176</td>
</tr>
<tr>
<td>14.1.3</td>
<td>Forming sub-arrays</td>
<td>176</td>
</tr>
<tr>
<td>14.1.4</td>
<td>Vector operations through iterators</td>
<td>177</td>
</tr>
<tr>
<td>14.2</td>
<td>Algorithms</td>
<td>179</td>
</tr>
<tr>
<td>14.2.1</td>
<td>Any/all/each</td>
<td>179</td>
</tr>
<tr>
<td>14.2.2</td>
<td>Reduction</td>
<td>180</td>
</tr>
<tr>
<td>14.2.3</td>
<td>Sorting</td>
<td>181</td>
</tr>
<tr>
<td>14.3</td>
<td>Ranges</td>
<td>182</td>
</tr>
<tr>
<td>14.3.1</td>
<td>Infinite sequences</td>
<td>184</td>
</tr>
<tr>
<td>15</td>
<td>References</td>
<td>185</td>
</tr>
<tr>
<td>15.1</td>
<td>Reference</td>
<td>185</td>
</tr>
<tr>
<td>15.2</td>
<td>Pass by reference</td>
<td>185</td>
</tr>
<tr>
<td>15.3</td>
<td>Reference to class members</td>
<td>186</td>
</tr>
<tr>
<td>15.4</td>
<td>Reference to array members</td>
<td>188</td>
</tr>
<tr>
<td>15.5</td>
<td>rvalue references</td>
<td>189</td>
</tr>
<tr>
<td>16</td>
<td>Pointers</td>
<td>191</td>
</tr>
<tr>
<td>16.1</td>
<td>The ‘arrow’ notation</td>
<td>191</td>
</tr>
<tr>
<td>16.2</td>
<td>Making a shared pointer</td>
<td>192</td>
</tr>
<tr>
<td>16.2.1</td>
<td>Pointers and arrays</td>
<td>193</td>
</tr>
<tr>
<td>16.2.2</td>
<td>Smart pointers versus address pointers</td>
<td>194</td>
</tr>
</tbody>
</table>
Contents

16.3 Garbage collection 195
16.4 More about pointers 196
16.4.1 Get the pointed data 196
16.5 Advanced topics 197
16.5.1 Unique pointers 197
16.5.2 Base and derived pointers 197
16.5.3 Shared pointer to ‘this’ 197
16.5.4 Weak pointers 198
16.5.5 Null pointer 199
16.5.6 Opaque pointer 199
16.5.7 Pointers to non-objects 199
16.6 Smart pointers vs C pointers 200
17 C-style pointers and arrays 201
17.1 What is a pointer 201
17.2 Pointers and addresses, C style 201
17.3 Arrays and pointers 203
17.4 Pointer arithmetic 205
17.5 Multi-dimensional arrays 205
17.6 Parameter passing 205
17.6.1 Allocation 206
17.6.2 Use of new 209
17.7 Memory leaks 209
17.8 Const pointers 210
18 Const 211
18.1 Const arguments 211
18.2 Const references 211
18.2.1 Const references in range-based loops 213
18.3 Const methods 213
18.4 Overloading on const 213
18.5 Const and pointers 214
18.5.1 Old-style const pointers 215
18.6 Mutable 216
18.7 Compile-time constants 217
19 Prototypes and header files 219
19.1 Include files 219
19.2 Prototypes for functions 220
19.2.1 Separate compilation 221
19.2.2 Header files 221
19.2.3 C and C++ headers 222
19.3 Prototypes for class methods 223
19.4 Header files and templates 223
19.5 Namespaces and header files 224
19.6 Global variables and header files 224
19.7 Modules 225
20 Namespaces 227
20.1 Solving name conflicts 227
Contents

28.9.3 Lvalue and rvalue 281
29 C++ review questions 283
29.1 Arithmetic 283
29.2 Looping 283
29.3 Functions 283
29.4 Vectors 284
29.5 Vectors 284
29.6 Objects 284

III Fortran 287
30 Basics of Fortran 289
30.1 Source format 289
30.2 Compiling Fortran 290
30.3 Main program 290
30.3.1 Program structure 290
30.3.2 Statements 291
30.3.3 Comments 291
30.4 Variables 291
30.4.1 Declarations 292
30.4.2 Range 294
30.4.3 Initialization 295
30.5 Quick comparison Fortran vs C++ 295
30.5.1 Statements 295
30.5.2 Input/Output, or I/O as we say 296
30.5.3 Expressions 296
30.6 Complex numbers 297
30.7 Review questions 297
31 Conditionals 299
31.1 Forms of the conditional statement 299
31.2 Operators 299
31.3 Select statement 300
31.4 Boolean variables 300
31.5 Review questions 301
32 Loop constructs 303
32.1 Loop types 303
32.2 Interruptions of the control flow 305
32.3 Implied do-loops 306
32.4 Review questions 306
33 Scope 309
33.1 Scope 309
33.1.1 Variables local to a program unit 309
33.1.2 Variables in an internal procedure 310
34 Procedures 311
34.1 Subroutines and functions 311
34.2 Return results 313
34.2.1 The ‘result’ keyword 315
34.2.2 The ‘contains’ clause 315
34.3 Arguments 316
34.3.1 Keyword and optional arguments 317
34.4 Types of procedures 318
34.5 Local variable save-ing 318
35 String handling 321
35.1 String denotations 321
35.2 Characters 321
35.3 Strings 321
35.4 Strings versus character arrays 322
35.5 Formatted I/O 322
35.5.1 Format letters 322
35.5.2 Repeating and grouping 323
36 Structures, eh, types 325
37 Modules 329
37.1 Modules for program modularization 330
37.2 Module definition 330
37.3 Separate compilation 331
37.4 Access 331
37.5 Polymorphism 332
37.6 Operator overloading 332
38 Classes and objects 333
38.1 Classes 333
38.1.1 Final procedures: destructors 335
38.2 Inheritance 336
38.3 Operator overloading 337
39 Arrays 339
39.1 Static arrays 339
39.1.1 Initialization 340
39.1.2 Array sections 340
39.1.3 Integer arrays as indices 342
39.2 Multi-dimensional 342
39.2.1 Querying an array 344
39.2.2 Reshaping 345
39.3 Arrays to subroutines 345
39.4 Allocatable arrays 346
39.4.1 Returning an allocated array 346
39.5 Array output 347
39.6 Operating on an array 347
39.6.1 Arithmetic operations 347
39.6.2 Intrinsic functions 347
39.6.3 Restricting with where 349
39.6.4 Global condition tests 349
39.7 Array operations 350
39.7.1 Loops without looping 350
40.1 Basic pointer operations 355
40.2 Pointers and arrays 358
40.3 Example: linked lists 359
40.3.1 Type definitions 359
40.3.2 Attach a node at the end 360
40.3.3 Insert a node in sort order 360

41 Input/output 363
41.1 Types of I/O 363
41.2 Print to terminal 363
41.2.1 Print on one line 364
41.2.2 Printing arrays 364
41.3 File and stream I/O 364
41.3.1 Units 364
41.3.2 Other write options 365

42 Leftover topics 367
42.1 Interfaces 367
42.1.1 Polymorphism 367
42.2 Random numbers 368
42.3 Timing 368

43 Fortran review questions 369
43.1 Fortran versus C++ 369
43.2 Basics 369
43.3 Arrays 369
43.4 Subprograms 370

IV Julia (largely yet to be written) 371
44 Basics of Julia 373
44.1 Statements 373
44.2 Variables 374
44.2.1 Datatypes 374
44.2.2 Truth values 375
44.3 Expressions 376
44.3.1 Terminal input 376
44.4 Tuples 376
45 Conditionals 377
45.1 Basic syntax 377
45.2 Advanced topics 378
45.2.1 Short-circuit operators 378
45.2.2 Ternary if 378
46 Looping 381
53.4  Is-a relationship  410
53.5  More stuff  411
54  Zero finding  413
54.1  Root finding by bisection  413
54.1.1  Simple implementation  413
54.1.2  Polynomials  414
54.1.3  Left/right search points  415
54.1.4  Root finding  416
54.1.5  Object implementation  417
54.1.6  Templating  417
54.2  Newton’s method  417
55  Eight queens  419
55.1  Problem statement  419
55.2  Solving the eight queens problem, basic approach  420
55.3  Developing a solution by TDD  420
55.4  The recursive solution method  422
56  Infectious disease simulation  425
56.1  Model design  425
56.1.1  Other ways of modeling  425
56.2  Coding  426
56.2.1  The basics  426
56.2.2  Population  427
56.2.3  Contagion  428
56.2.4  Spreading  428
56.2.5  Diseases without vaccine: Ebola and Covid-19  429
56.3  Ethics  429
56.4  Project writeup and submission  430
56.4.1  Program files  430
56.4.2  Writeup  430
56.5  Bonus: mathematical analysis  430
57  Google PageRank  433
57.1  Basic ideas  433
57.2  Clicking around  434
57.3  Graph algorithms  434
57.4  Page ranking  435
57.5  Graphs and linear algebra  436
58  Redistricting  437
58.1  Basic concepts  437
58.2  Basic functions  438
58.2.1  Voters  438
58.2.2  Populations  438
58.2.3  Districting  439
58.3  Strategy  440
58.4  Efficiency: dynamic programming  441
58.5  Extensions  442
58.6  Ethics  442
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming strategies</strong></td>
<td>481</td>
</tr>
<tr>
<td>66.1 A philosophy of programming</td>
<td>481</td>
</tr>
<tr>
<td>66.2 Programming: top-down versus bottom up</td>
<td>481</td>
</tr>
<tr>
<td>66.2.1 Worked out example</td>
<td>482</td>
</tr>
<tr>
<td>66.3 Coding style</td>
<td>483</td>
</tr>
<tr>
<td>66.4 Documentation</td>
<td>483</td>
</tr>
<tr>
<td>66.5 Best practices: C++ Core Guidelines</td>
<td>484</td>
</tr>
<tr>
<td><strong>Tiniest of introductions to algorithms and data structures</strong></td>
<td>485</td>
</tr>
<tr>
<td>67.1 Data structures</td>
<td>485</td>
</tr>
<tr>
<td>67.1.1 Stack</td>
<td>485</td>
</tr>
<tr>
<td>67.1.2 Linked lists</td>
<td>485</td>
</tr>
<tr>
<td>67.1.3 Trees</td>
<td>490</td>
</tr>
<tr>
<td>67.2 Algorithms</td>
<td>492</td>
</tr>
<tr>
<td>67.2.1 Sorting</td>
<td>492</td>
</tr>
<tr>
<td>67.3 Programming techniques</td>
<td>493</td>
</tr>
<tr>
<td>67.3.1 Memoization</td>
<td>493</td>
</tr>
<tr>
<td><strong>Provably correct programs</strong></td>
<td>495</td>
</tr>
<tr>
<td>68.1 Loops as quantors</td>
<td>495</td>
</tr>
<tr>
<td>68.1.1 Forall-quantor</td>
<td>495</td>
</tr>
<tr>
<td>68.1.2 Thereis-quantor</td>
<td>496</td>
</tr>
<tr>
<td>68.2 Predicate proving</td>
<td>496</td>
</tr>
<tr>
<td>68.3 Flame</td>
<td>498</td>
</tr>
<tr>
<td><strong>Unit testing and Test-Driven Development</strong></td>
<td>503</td>
</tr>
<tr>
<td>69.1 Types of tests</td>
<td>503</td>
</tr>
<tr>
<td>69.2 Unit testing frameworks</td>
<td>504</td>
</tr>
<tr>
<td>69.2.1 File structure</td>
<td>504</td>
</tr>
<tr>
<td>69.2.2 Compilation</td>
<td>505</td>
</tr>
<tr>
<td>69.2.3 Test cases</td>
<td>505</td>
</tr>
<tr>
<td>69.3 An example: quadratic equation roots</td>
<td>508</td>
</tr>
<tr>
<td>69.4 Eight queens example</td>
<td>510</td>
</tr>
<tr>
<td><strong>Debugging with gdb</strong></td>
<td>511</td>
</tr>
<tr>
<td>70.1 Invoking the debugger</td>
<td>511</td>
</tr>
<tr>
<td>70.1.1 Run with commandline arguments</td>
<td>512</td>
</tr>
<tr>
<td>70.2 Stepping through the source</td>
<td>513</td>
</tr>
<tr>
<td>70.3 Inspecting values</td>
<td>515</td>
</tr>
<tr>
<td>70.4 A NaN example</td>
<td>515</td>
</tr>
<tr>
<td>70.5 Assertions</td>
<td>517</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>519</td>
</tr>
<tr>
<td>71.1 Order of complexity</td>
<td>519</td>
</tr>
<tr>
<td>71.1.1 Time complexity</td>
<td>519</td>
</tr>
<tr>
<td>71.1.2 Space complexity</td>
<td>519</td>
</tr>
</tbody>
</table>

**VII Index and such** 521

**General index of terms** 523
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>Index of C++ keywords</td>
<td>531</td>
</tr>
<tr>
<td>74</td>
<td>Index of Fortran keywords</td>
<td>535</td>
</tr>
<tr>
<td>75</td>
<td>Bibliography</td>
<td>539</td>
</tr>
</tbody>
</table>
PART I

INTRODUCTION
Chapter 1

Introduction

1.1 Programming and computational thinking

1.1.1 History

**Earliest computers** Historically, computers were used for big physics calculations, for instance, atom bomb calculations

(Robert Oppenheimer and John von Neumann)

**Hands-on programming** Very early computers such as ENIAC were hardwired

later became ‘stored program’ computer.

see [http://eniacprogrammers.org/](http://eniacprogrammers.org/)
1. Introduction

Program entry  Later programs were written on punchcards

The first programming language  Initial programming was about translating the math formulas; after
a while they made a language for that: FORmula TRANslating

Programming is everywhere  Programming is used in many different ways these days.

- You can make your own commands in Microsoft Word.
- You can make apps for your smartphone.
- You can solve the riddles of the universe using big computers.

This course is aimed at people in the last category.

1.1.2  Is programming science, art, or craft?

In the early days of computing, hardware design was seen as challenging, while programming was little
more than data entry. The fact that Fortran stands for ‘formula translation’ speaks of this: once you have
the math, programming is nothing more than translating the math into code. The fact that programs could
have subtle errors, or bugs, came as quite a surprise.
The fact that programming was not as highly valued also had the side-effect that many of the early programmers were 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.

There are scientific aspects to programming:

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

But 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 behaviour 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

Programming is not simple  Programs can get pretty big:

Margarine Hamilton, director of the Software Engineering Division, the MIT Instrumentation Laboratory, which developed on-board software for the Apollo program.

It’s not just translating formulas anymore.
Translating ideas to computer code: computational thinking.

Computational thinking: elevator scheduling  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 ≠ existence proof.
Computational thinking: actual design of a solution

1. Introduction

- 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?

Computation and complexity 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!

Programming languages are about ideas

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

Abstraction

- 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 is good 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.
1.1. Programming and computational thinking

(Bad programming style may lead to lower grade.)

**Trust him, he’s Dutch!**
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

<table>
<thead>
<tr>
<th>Data abstraction</th>
<th>What is the structure of the data in your program?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack: you can only get at the top item</td>
<td>Queue: items get added in the back, processed at the front</td>
</tr>
</tbody>
</table>

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

### 1.1.4 Hardware

**Do you have to know much about 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 TACC’s ISTC course).

### 1.1.5 Algorithms

**What is an algorithm?**
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

**Program steps**
- Simple instructions: arithmetic.
- Complicated instructions: control structures
1. Introduction

– conditionals
– loops

Program data
- 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.

Comparing two languages  Python vs C++ on bubblesort:

```python
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
```

```c++
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.

The right language is not all Python with quicksort algorithm:

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

[] python arraysort.py 5000
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.
1. Introduction
Chapter 2

Warming up

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 author of this book is very much in favour of the emacs editor. The main reason is its support for programming languages. Most of the time it 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 colouring’ to indicate the difference between keywords and variables.
2. Warming up

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
   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

**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 ‘good practices’ approach, where students learn enough of each topic to become a competent programmer. 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 timeline 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. Teachers guide

3.2 Timeline 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 timeline 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

Depending on time availability, the topics in table 3.2 can be taught.

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, we have given 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 52.
- **geom** Geometry related concepts; this is mostly an exercise in object-oriented programming. Chapter 53.
- **root** Numerical zero-finding methods. Chapter 54.

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 56.
- **pagerank** The Google Pagerank algorithm. Students program a simulated internet, and explore pageranking, including 'search engine optimization'. This exercise uses lots of pointers. Chapter 57.
- **gerrymandering** Redistricting through dynamic programming. Chapter 58.
- **scheduling** Exploration of concepts related to the Multiple Traveling Salesman Problem (MTSP), modeling Amazon Prime. Chapter 59.
- **lapack** Exploration of high performance linear algebra. Chapter 60.

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

The project assignemnts give a fairly detailed step-by-step suggested approach. This acknowledges the theory of Cognitive Load [8].

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.
### 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.

<table>
<thead>
<tr>
<th>lesson</th>
<th>Topic</th>
<th>in-class</th>
<th>homework</th>
<th>prime</th>
<th>geom</th>
<th>infect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Statements and expressions</td>
<td>4.4</td>
<td>4.11 (T)</td>
<td>52.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Conditionals</td>
<td>5.2 (S), 5.3</td>
<td>5.4 (T)</td>
<td>52.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Looping</td>
<td>6.5 (S), 6.4</td>
<td>6.6 (T)</td>
<td>52.3, 52.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>continue</td>
<td>7.1 (S), 7.2</td>
<td>7.6</td>
<td>52.6, 52.7 (T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Functions</td>
<td>7.11</td>
<td>12.1</td>
<td>53.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>continue</td>
<td>9.7 (S), 52.9 (T), 52.11</td>
<td>53.3</td>
<td>56.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>I/O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Objects</td>
<td>10.1 (S), 10.21</td>
<td>52.16</td>
<td>56.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>continue</td>
<td>53.9 (T), 53.10, 53.12</td>
<td>53.3, 53.14, 53.15</td>
<td>56.2 and further</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>has-a relation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>inheritance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Vectors</td>
<td>10.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>continue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Strings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>lesson</th>
<th>Topic</th>
<th>Book</th>
<th>Slide</th>
<th>Prerequisite</th>
<th>Exercises</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Code modularity, prototypes</td>
<td>19</td>
<td>7b</td>
<td></td>
<td>Collatz: 6.12; swap: 7.5; vectors: 10.19</td>
</tr>
<tr>
<td>2</td>
<td>Unit testing</td>
<td>54.1</td>
<td>18</td>
<td>Separate compilation</td>
<td>54.1–54.5</td>
</tr>
<tr>
<td>3, 4</td>
<td>Objects</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>hasa/isa</td>
<td>11, 12</td>
<td></td>
<td>Rect/square:</td>
<td>53.14</td>
</tr>
<tr>
<td>6</td>
<td>objects and vectors</td>
<td>10.5</td>
<td></td>
<td>Pascal:</td>
<td>10.21</td>
</tr>
<tr>
<td>7</td>
<td>Lambda functions</td>
<td>13</td>
<td>7c</td>
<td></td>
<td>54.8 (S)</td>
</tr>
<tr>
<td>8</td>
<td>Iterators</td>
<td>14.1</td>
<td>14b</td>
<td>lambda</td>
<td>14.2</td>
</tr>
<tr>
<td>9</td>
<td>Pointers</td>
<td>16</td>
<td>15b</td>
<td>section</td>
<td>67.1.2</td>
</tr>
<tr>
<td>10</td>
<td>STL and other libraries</td>
<td>24.8</td>
<td></td>
<td></td>
<td>52.15</td>
</tr>
<tr>
<td>11</td>
<td>libraries and cmake</td>
<td>24.5.2</td>
<td></td>
<td></td>
<td>52.15</td>
</tr>
<tr>
<td></td>
<td>optional</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>chrono</td>
<td>24.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nnn</td>
<td>C legacy: pointers</td>
<td>17</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nnn</td>
<td>Exceptions</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prototypes, namespaces</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>formal approaches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Introduction to Scientific Programming
3.3 Scaffolding

As much as possible, when introducing a new topic I try to give 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 table, exercises are indicated for which a skeleton code is given.

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.

Correctness of the results is secondary: 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.

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.
3. Teachers guide

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:

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

3.4.3 Conditionals

3.4.4 Looping

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

Local declaration:

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

Global declaration:

```cpp
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.

3.4.5 Functions

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

Defined before:

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

Only proto given before:

```cpp
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: not C-style int* parameters and such.

3.4.6 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++

Introduction to Scientific Programming
3.4. Grading guide

3.4.7 Structures

Do not declare a single global structure:

Good:

```cpp
struct Point { double x, y; }
int main() {
  Point mypoint;
  cin >> mypoint.x;
}
```

Not good:

```cpp
struct Point { double x, y; } mypoint;
int main() {
  cin >> mypoint.x;
}
```

3.4.8 Classes

- Do not declare data members `public`. Write accessor functions, but only if needed.
- 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.

3.4.9 Arrays vs vectors

- Do not use old-style C arrays.
  ```cpp
  int indexes[5];
  ```
- Certainly never use `malloc` or `new`.
- Iterator operations (`begin, erase`) are seldom used in this course, and should only be used if strictly needed.

3.4.10 Strings

Do not use old-style C strings:

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

3.4.11 Other remarks

3.4.11.1 Uninitialized variables

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

```cpp
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;
    }
```
This whole issue can be sidestepped 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 | ./missinginit
```

Run-Time Check Failure: The variable 'found_solution' is being used in missinginit.cxx(22,5) without being initialized

Aborted (core dumped)

### 3.4.11.2 Clearing objects and vectors

The following idiom is found regularly:

```cpp
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: how do you deal with any program?

Two kinds of files 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++*.)

Let’s do an example.

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

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

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

and compile it. Intel compiler:

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


Run this program (it gives no output):

```
./zeroprogram
```

**Compile line**

- `icpc`: compiler. Alternative: use `g++` or `clang++`
- `-o zeroprogram`: output into a binary name of your choosing
- `zero.cc`: your source file.

In the above program:

1. The first three lines are magic, for now. Always include them.
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 4.6.4.)

As you may have guessed, this program produces absolutely no output when you run it.

**Exercise 4.2.** Add this line:

```
    cout << "Hello world!" << endl;
```

(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?)

**File names** File names can have extensions: the part after the dot.

- `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**.

**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

---

*Introduction to Scientific Programming*
4.2 Statements

- The option/value pair \texttt{-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:

\begin{verbatim}
g++ -O3 -o myprogram myprogram.cxx
\end{verbatim}

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

\begin{verbatim}
g++ -o myprogram -O3 myprogram.cxx
\end{verbatim}

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

\begin{verbatim}
g++ -o myprogram.cxx myprogram
\end{verbatim}

then Unix will reason: ‘\texttt{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 \texttt{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 26.7.) In this course we focus on a fairly recent standard: C++17. Unfortunately, your compiler will assume an earlier standard, so constructs taught here may be marked as ungrammatical.

You can tell your compiler to use the modern standard:

\begin{verbatim}
icpc -std=c++17 [other options]
\end{verbatim}

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

\begin{verbatim}
alias icpc='icpc -std=c++17'
\end{verbatim}

in your shell startup files. On the class \texttt{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.

Program statements

- 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:

\begin{verbatim}
\| cout << "Hello world" << endl; // say hi!
\end{verbatim}
4. Basic elements of C++

and arbitrary:

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

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?

Errors  There are two types of errors that your program can have:

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.

Fixed elements  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.

A word about using. After

```cpp
    #include <iostream>
```

you can write std::cout in your program, but the ‘using’ statements allow you to just write cout.

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?

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.

Typical variable lifetime

```
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 once in a variable declaration, but it can be given a (new) value multiple times. It is not an error to use a variable that has not been given a value, but it may lead to strange behaviour at runtime, since the variable may contain random memory contents.

Variable names

• 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.

Declaration There are a couple of ways to make the connection between a name and a type. Here is a simple variable declaration, which establishes the name and the type:

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

Where do declarations go? 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 that’s not a good idea. Please only declare 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 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 \( i = 5 \).

Assignment 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:

\[
\begin{align*}
\text{n} & = 3; \\
\text{x} & = 1.5; \\
\text{n1} & = 7; \text{n2} = \text{n1} + 3;
\end{align*}
\]

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.

Special forms Update:

\[
\begin{align*}
\text{x} & = \text{x} + 2; \text{ y} = \text{y} / 3; \\
& \text{ // can be written as} \\
& \text{x} += 2; \text{ y} /= 3;
\end{align*}
\]

Integer add/subtract one:

\[
\begin{align*}
\text{i}++; \text{j}--; & \text{ // same as: } i=i+1; j=j-1;
\end{align*}
\]

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. \( n /= k; \)
4.3. Variables

Exercise 4.6.
```cpp
#include <iostream>
using std::cout;
using std::endl;
int main() {
    int i;
    int j = i + 1;
    cout << j << endl;
    return 0;
}
```

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

4.3.3 Terminal input

In most of your program, variables will be assigned values that arise somewhere in the course of the computation. However, especially in the start of your program, you may want to have some values that are specified interactively, by you typing them in.

**Terminal input** To make a program run dynamic, you can set starting values from keyboard input. For this, use `cin`, which takes keyboard input and puts it in a numerical (or string) variable.

```cpp
// add at the top of your program:
using std::cin;

// then in your main:
int i;
cin >> i;
```

4.3.4 Datatypes

**Datatypes** 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.

At some point you may start to wonder precisely what the range of integers or real numbers is that is stored in an `int` or `float` variable. This is addressed in section 24.6.
4. Basic elements of C++

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.

C++ integers are stored as binary number and a sign bit, though not as naively as this sounds. The upshot is that normally within a certain range 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 representible integer range?

The integer type in C++ is int:

```cpp
int my_integer;
my_integer = 5;
cout << my_integer << endl;
```

For more integer types, see section 4.6.1.

4.3.4.2 Floating point variables

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

\[ +6 \cdot 0.22 \times 10^{23} \]

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 also referred to as ‘real numbers’ (in fact, in the Fortran language they are defined with the keyword Real), but this is sloppy working. Since only a finite number of bits/digits is available, only terminating fractions are representable. For instance, since computer numbers are binary, \( \frac{1}{2} \) is representable but \( \frac{1}{3} \) is not.

Exercise 4.8. Can you think of a way that non-terminating fractions, including such numbers such as \( \sqrt{2} \), are still 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.

Warning: floating point arithmetic Floating point arithmetic is full of pitfalls.

- Don’t count on \( 3 \times (1/3) \) being exactly 1.
- Not even associative.

(See Eijkhout, Introduction to High Performance Computing, chapter 3.)

The following exercise illustrates another point about computer numbers.
Exercise 4.9. Define

\[ \texttt{float one} = 1.0; \]

and

1. Read a \texttt{float eps},
2. Make a new variable that has the value \texttt{one+eps}. Print this.
3. Make another variable that is the previous one minus \texttt{one}. Print the result again.
4. Test your program with \texttt{.001, .0001, .00001, .000001}. Do you understand the result?

Complex numbers exist, see section 24.1.

4.3.4.3 Boolean values

Truth values So far you have seen integer and real variables. There are also boolean values which represent truth values. There are only two values: \texttt{true} and \texttt{false}.

\[
\begin{align*}
\text{bool found} &\equiv \texttt{false}; \\
\text{found} &\equiv \texttt{true};
\end{align*}
\]

4.3.4.4 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 now, if you want to use strings:

Quick intro to strings

- Add the following at the top of your file:

\[
\begin{align*}
\texttt{#include <string>} \\
\texttt{using std::string};
\end{align*}
\]

- Declare string variables as

\[
\texttt{string name;}
\]

- And you can now \texttt{cin} and \texttt{cout} them.

Exercise 4.10. Write a program that asks for the user’s first name, uses \texttt{cin} to read that, and prints something like \texttt{Hello, Susan!} in response.

What happens if you enter first and last name?

4.3.5 Complex numbers

See section 24.1.

4.3.6 Initialization

It is 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 4.3.2).
4. Basic elements of C++

**Initialization syntax** There are two ways of initializing a variable

```cpp
int i = 5;
int j = 6;
```

Note that writing

```cpp
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.

### 4.4 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 output** You have already seen `cout`:

```cpp
float x = 5;
cout << "Here is the root: " << sqrt(x) << endl;
```

**Terminal input** `cin` is limited. There is also `getline`, which is more general.

For fine-grained control over the output, and 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 appending. Let's start with constants.

#### 4.5.1 Numbers

#### 4.5.1.1 Integers

Integers consist of a sequence of digits, preceeded by a plus or minus sign.

Hexadecimal integers start with `0x` or `0X`.

Binary numbers start with `0b`.

Introduction to Scientific Programming
4.5.1.2 Floating point numbers

Floating point numbers are by default double: double precision.

\[
\begin{align*}
1.5 \\
1.5e+5
\end{align*}
\]

Use a suffix \(1.5f\) for float: single precision.

\[
\begin{align*}
1.5f \\
1.5e+5f
\end{align*}
\]

Use a suffix \(1.5L\) for long double: quadruple precision.

\[
\begin{align*}
1.5L \\
1.5e+5L
\end{align*}
\]

Hexadecimal floating points numbers have a complicated format.

4.5.2 Numerical expressions

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

- Mathematical operators: + - / and \(\ast\) for multiplication.
- Integer modulus: 5 \(\%\) 2
- You can use parentheses: \(5 \ast (x+y)\). 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 library calls  Math function in \texttt{cmath}:

\[
\begin{align*}
\texttt{#include <cmath>} \\
\texttt{......} \\
\texttt{x = pow(3,.5);} \\
\end{align*}
\]

For squaring, usually better to write \(x\ast x\) than \texttt{pow(x,2)}.

Exercise 4.11.  Write a program that:

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

4.5.3 Truth values

In addition to numerical types, there are truth values, \texttt{true} and \texttt{false}, with all the usual logical operators defined on them.
4. Basic elements of C++

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

```cpp
if ( x>=0 && sqrt(x)<5 ) {};
```

4.5.3.1 Short-cut operators
Logical expressions in C++ are evaluated using shortcut operators. Strictly speaking, the expression

```cpp
x>=0 && sqrt(x)<2
```

would be a problem for negative \( x \) values, because you have to evaluate both members.

However, C++ has a strict left-to-right evaluation of expressions. If \( x \) is negative, the ‘and’ conjunction can already be concluded to be false, and so the second part will never be evaluated. Similarly, ‘or’ conjunctions will only be evaluated until the first true clause.

4.5.3.2 Bit operators
The ‘true’ and ‘false’ constants could strictly speaking be stored in a single bit. C++ does not do that, but there are bit operators that you can apply to, for instance, all the bits in an integer.

Bit operators Bitwise: & | ~

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

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

or

```cpp
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 4.12.
- 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?
- What happens when you assign a float to a double? Try various numbers for the original float. Can you explain the result? (Hint: think about the conversion between binary and decimal.)
The rules for type conversion in expressions are not entirely logical. Consider

\[
\begin{align*}
\text{float } x; & \quad \text{int } i=5, j=2; \\
x = i/j;
\end{align*}
\]

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

\[
\begin{align*}
x = (1.*i)/j;
\end{align*}
\]

or any other mechanism that forces a conversion, without changing the result. Another mechanism is the cast; this will be discussed in section 26.3.

**Exercise 4.13.** Write a program that asks for two integer numbers \(n_1, n_2\).

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

\[
\begin{align*}
n_1 \mod n_2
\end{align*}
\]

(without using the \(\%\) modulus operator!)

Print out the value you get.

- Also print out the result from using the modulus operator: \(\%\).
- Investigate the behaviour of your program for negative inputs. Do you get what you were expecting?

**Exercise 4.14.** Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

\[
\begin{align*}
C = (F - 32) \cdot 5/9, & \quad F = 9/5 C + 32
\end{align*}
\]

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 or 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. \(\text{int } i = 5/3.;\) The variable \(i\) is 2.
6. \(\text{float } x = 2/3;\) The variable \(x\) is approximately \(0.6667\).
4.6 Advanced topics

4.6.1 Integer precision

In addition to `int`, there are also `short` and `long` integers.

- 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.

For the whole truth of these so-called data models, see [https://en.cppreference.com/w/cpp/language/types](https://en.cppreference.com/w/cpp/language/types).

4.6.2 Unsigned types

For the integer types `int`, `short`, `long` there are unsigned types

```cpp
unsigned int i;
unsigned short s;
unsigned long l;
```

which contain nonnegative values. Consequently they have twice the range:

```cpp
Code:
cout << "max int : " << numeric_limits<int>::max() << \n"n";
cout << "max unsigned: " << numeric_limits<unsigned int>::max();
```

Output

<table>
<thead>
<tr>
<th>[int] limit:</th>
</tr>
</thead>
<tbody>
<tr>
<td>max int : 2147483647</td>
</tr>
<tr>
<td>max unsigned: 4294967295</td>
</tr>
</tbody>
</table>

Unsigned values are fraught with danger. For instance, comparing them to integers gives counter-intuitive results:

```cpp
Code:
unsigned int one(1);
int mone(-1);
cout << "less: " << boolalpha << (mone < one);,
```

Output

<table>
<thead>
<tr>
<th>cmp:</th>
</tr>
</thead>
<tbody>
<tr>
<td>less: false</td>
</tr>
</tbody>
</table>

For this reason, C++20 has introduced utility functions `cmp_equal` and such (in the `utility` header) that do these comparisons correctly.

4.6.3 Floating point precision

See section 24.6.

4.6.4 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`.

The returned `int` can be specified several ways:
• If not `return` statement is given, implicitly `return 0` is performed.
• 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
```

• You can also use the `exit` function:

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

### 4.6.5 Number values and undefined values
A computer allocates a fixed amount of space for integers and floating point numbers, typically 4 or 8 bytes. That means that not all numbers are representable.

- Using 4 bytes, that is 32 bits, we can represent $2^{32}$ integers. Typically this is the range $-2^{31} \ldots 0 \ldots 2^{31} - 1$.
- Floating point numbers are represented by a sign bit, an exponent, and a number of significant digits. For 4-byte numbers, the number of significant (decimal) digits is around 6; for 8-byte numbers it is around 15.

If you compute a number that ‘fall in between the gaps’ of the representable numbers, it gets truncated or rounded. The effects of this on your computation constitute its own field of numerical mathematics, called roundoff error analysis.

If a number goes outside the bounds of what is representable, it becomes either:

- $\text{Inf}$: infinity. This happens if you add or multiply enough large numbers together. There is of course also a value $-\text{Inf}$. Or:
- $\text{NaN}$: not a number. This happens if you subtract one $\text{Inf}$ from another, or do things such as taking the root of a negative number.

Your program will not be interrupted if a $\text{NaN}$ or $\text{Inf}$ is generated: the computation will merrily (and at full speed) progress with these numbers. See section 24.6 for detection of such quantities.

Some people advocate filling uninitialized memory with such illegal values, to make it recognizable as such.

### 4.6.6 Constants

**Floating point constants**

- Default: `double`
- Float: `3.14f` or `3.14F`
### 4. Basic elements of C++

- **Long double**: `1.22l` or `1.22L`.

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

```cpp
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.

#### 4.6.7 Numerical precision

Truncation and precision are tricky things.

**Code:**

```cpp
double point3d = .3/.7;
float
    point3f = .3f/.7f,
    point3t = point3d;
cout << "double precision: "
    << point3d << endl
    << "single precision: "
    << point3f << endl
    << "difference with truncation:"
    << point3t - point3f
    << endl;
```

**Output**

```
[basic] point3:
  double precision: 0.428571
  single precision: 0.428571
  difference with truncation:-2.98023e-08
```

### 4.7 Review questions

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

```cpp
int m=32, n=17;
cout << n%m << endl;
```

**Review 4.6.** Given

```cpp
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 \( x > 0 \), 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.

5.1 Conditionals

Here are some forms a conditional can take.

A single statement, executed if the test is true:

\[
\text{if (} x < 0 \text{)} \quad x = -x;
\]

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

\[
\text{if (} x \geq 0 \text{)} \quad x = 1; \\
\text{else} \quad x = -1;
\]

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

\[
\text{if (} x < 0 \text{)} \quad \{ \quad x = 2 \times x; \quad y = y/2; \quad \} \\
\text{else} \quad \{ \quad x = 3 \times x; \quad y = y/3; \quad \}
\]

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

\[
\text{if (} x > 0 \text{)} \quad \{ \quad .... \quad \} \text{ else if (} x < 0 \text{)} \quad \{ \\
\text{else} \quad \{ \quad .... \quad \}
\]

Conditionals can also be nested:
5. Conditionals

```c
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’?

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

```c
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. Here is a fragment of language grammar that spells out what is legal. You see that most of the rules are recursive, but there is an important exception.

**What are logical expressions?**

```plaintext
logical_expression ::= 
    comparison_expression
  | NOT comparison_expression
  | logical_expression CONJUNCTION comparison_expression

comparison_expression ::= 
    numerical_expression COMPARE numerical_expression

numerical_expression ::= 
    quantity
  | numerical_expression OPERATOR quantity

quantity ::= number | variable
```

**Comparison and logical operators**  Here are the most common *logic operators* and *comparison*
5.2. Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>meaning</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
<td>( x = y - 1 )</td>
</tr>
<tr>
<td>!=</td>
<td>not equals</td>
<td>( x \times x = 5 )</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater</td>
<td>( y &gt; x - 1 )</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal</td>
<td>( \sqrt{y} \geq 7 )</td>
</tr>
<tr>
<td>&lt;,&lt;=</td>
<td>less, less equal</td>
<td>( x &lt; 1 ) and ( x &gt; 0 )</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>( x &lt; 1 ) and ( x &gt; 0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>( x = 1 ) and ( x &lt; 2 )</td>
</tr>
</tbody>
</table>

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

Remark 1 There are also bitwise operators.

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ( x = 6, y = 3; )</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;6</td>
<td>3 = &quot; &lt;&lt; ((x</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;6&amp;3 = &quot; &lt;&lt; ((x&amp;y)) &lt;&lt; endl;</td>
<td>6&amp;3 = 2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ( i; )</td>
<td>[basic] if:</td>
</tr>
<tr>
<td>cin &gt;&gt; i;</td>
<td>... with 50 as input ....</td>
</tr>
<tr>
<td>if ( ( i &gt; 100 ) )</td>
<td>... with 150 as input ....</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;That number &quot; &lt;&lt; ( i ) &lt;&lt; &quot; had more than 2 digits&quot;</td>
<td>That number 150 had more than 2 digits</td>
</tr>
</tbody>
</table>

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*

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

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( /* your test here */ )</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;even&quot; &lt;&lt; endl;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;odd&quot; &lt;&lt; endl;</td>
<td></td>
</tr>
</tbody>
</table>

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!’; If it is a multiple of both three and five print ‘Fizzbuzz!’; Otherwise print nothing.
5. Conditionals

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

Review 5.1. True or false?
- The tests `if (i>0)` and `if (0<i)` are equivalent.
- The test
  ```c
  if (i<0 && i>1)
  cout << "foo"
  ```
  prints `foo` if `i < 0` and also if `i > 1`.
- The test
  ```c
  if (0<i<1)
  cout << "foo"
  ```
  prints `foo` if `i` is between zero and one.

Review 5.2. Any comments on the following?
```c
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.

Switch statement example Cases are executed consecutively until you ‘break’ out of the switch statement:
```
Code:
```
```c
switch (n) {
  case 1 :
    cout << "very small" << endl;
    break;
  case 2 :
    cout << "trinity" << endl;
    break;
  default :
    cout << "large" << endl;
}
```
Output
```
[basic] switch:
for v in 1 2 3 4 5 ; do \
  echo $v | ./switch ; \
  done
very small
very small
trinity
large
```
Exercise 5.5. Suppose the variable \( n \) is a nonnegative integer. Write a `switch` statement that has the same effect as:

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

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.

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

See chapter 8 for more detail.

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 when the result is clear. For instance, in

```cpp
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

```cpp
clause1 or clause2
```

the second clause is not evaluated if the first one is `true`.

This mechanism allows you to write

```cpp
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

```cpp
if (foo)
    x = 5;
else
    y = 6;
```
But what about the case where the true and false branch perform in effect the same action, and only differ by an expression?

```cpp
if (foo)
  x = 5;
else
  x = 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:

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

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

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

### 5.5.3 Initializer

The C++17 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.

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

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

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

with the variable globally defined.

### 5.6 Review questions

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

```cpp
#include <iostream>
int main() {
  if (true)
    int i = 1;
  else
    int i = 2;
```

---

*Introduction to Scientific Programming*
5.6. Review questions

```cpp
std::cout << i;
return 0;
}
```

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

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

compare:

```cpp
if (cond1)
    i = 1;
else {
    if (cond2)
        i = 2;
    else
        i = 3;
}
5. Conditionals
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 6.2.

6.1 The ‘for’ loop

In the most common case, a for loop has a loop counter, ranging from some lower to some upper bound. An example showing the syntax for this simple case is:

```cpp
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.

We will now investigate the components of a loop. After the for keyword, there is the loop specification which is enclosed in parentheses, and contains three expressions (using the term metaphorically), separated by semicolons.
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. 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.

Loop syntax: variable

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!

Next there is a test, which is usually called a ‘stopping test’, but to be technically correct it is actually executed at the start of each iteration. It needs to evaluate to a boolean expression.

Loop syntax: test

- If this test is true, do the next iteration.
- Done before the first iteration too!
- Test can be empty. What does this mean?

```
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.

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.

Loop syntax: increment

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.
6.1. The ‘for’ loop

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 6.1.** 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. What are the starting and final value of the loop variable and what is the update? There are several possibilities!

**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 << endl;
```

and modify if 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?

<table>
<thead>
<tr>
<th>for (int i=1; i&lt;=2; i=i+2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i=1; i&lt;=5; i+=2)</td>
</tr>
<tr>
<td>for (int i=0; i&lt;0; i--)</td>
</tr>
<tr>
<td>for (int i=5; i&gt;=0; i--)</td>
</tr>
<tr>
<td>for (int i=5; i&gt;0; i--)</td>
</tr>
</tbody>
</table>

Some variations on the simple case.

**Loop variable declaration** It is preferable to declare the loop variable in the loop header:

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

The variable only has meaning inside the loop so it should only be defined there. However, it can also be defined outside the loop:

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

You will see an example where this makes sense below.

**No stopping test** The stopping test can be omitted

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

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

**Count down** 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.

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

**Stopping test is pre-test** The test is performed before each iteration:

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

Output

```
[basic] pretest:
  before the loop
  after the loop
```
6.1. The ‘for’ loop

**Loop body** The loop body can be a single statement:

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

or a block:

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

6.1.1 Loop syntax

The loop variable is typically an integer. You can use floating point numbers, but this may complicate the stopping test:

```c
for (float x=.1; x<=.5; x+=.1)
```

Because of numerical rounding it is not clear when this loop will end.

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.

```c
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. Below you’ll see that there are also different ways.

**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?

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 = constant` and `j = constant` respectively, a diagonal is defined by `i + j = constant`.

Victor Eijkhout 69
6. Looping

6.2 Looping until

The basic for loop looks pretty deterministic: a loop variable ranges through a more-or-less prescribes 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:

```c
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.

```c
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`.

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

\[
\textbf{for} \; (;;) \; \{ \text{/* some code */} \}
\]

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

```c
int var;
...
for ( ; var<upper; var++) {
    ...
    statements ...
    if (some condition) break
    ...
    more statements ...
}
...
```

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

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

```c
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.

Skip iteration

\[
\textbf{for} \; (\text{int} \; \text{var=low}; \; \text{var}<\text{N}; \; \text{var}++) \; \{ 
\]
6. Looping

```c
statement;
if (some_test) {
    statement;
    statement;
}
```

Alternative:

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

The only difference is in layout.

**While loop** The other possibility for ‘looping until’ is a while loop, which repeats until a condition is met.

Syntax:

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

or

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

The while loop does not have a counter or an update statement; if you need those, you have to create them yourself.

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

**Pre-test while loop**

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

Here is an example in which the second syntax is more appropriate.

**While syntax 1**
6.3. Advanced topics

6.3.1 Parallelism

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

---

```
Code:
#include <iostream>

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

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

---

```
Code:
#include <iostream>

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

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

---

Problem: code duplication.

### While syntax 2

```
Code:
#include <iostream>

int invar;

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

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 6.8.** At this point you are ready to do the exercises in the prime numbers project, section 52.3.

**Exercise 6.9.** 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 6.10.** 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.

---

6.3 Advanced topics

---

Victor Eijkhout
6. Looping

- 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.

6.4 Exercises

**Exercise 6.11.** 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 6.12.** 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 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \\
7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 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 6.13.** 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, plus a header and (maybe) a return statement.
- The function definition is placed before the main program.
- The function is called by its name.

Introducing a function

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 int and std::vector your program will use application terms, such as Point or Line.

7.1 Function definition and call

There are two aspects to a function:

program

int main() {
    x = ...

    // foo computation
    xtmp = ... x ...
    ytmp = ... x ... xtmp ....
    y = .... xtmp .... ytmp ....

    .... y ....
}

function

float foo_compute(float x) {
    // foo computation
    xtmp = ... x ...
    ytmp = ... x ... xtmp ....
    return .... xtmp .... ytmp ....
}

program

int main() {
    x = ...
    y = foo_compute(x);
    .... y ....
}
7. Functions

- 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 54.1 for details.

**Program without functions**  Example: zero-finding through bisection.

\[
? : f(x) = 0, \quad f(x) = x^3 - x^2 - 1
\]

(where the question mark quantor stands for ‘for which \(x\)).

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

**Code:**

```c++
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 << endl;
```

**Output**

```
[bisection] bisect1:
ld: warning: directory not found
    for option '-L/Library/
    Developer/CommandLineTools/
    SDKs/MacOSX.sdk/usr/lib'
Zero happens at: 1.4375
```

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

**Introducing functions, step 1**

Introduce function for the expression \(m^3 - m^2 - 1\):

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

Used in main:

```c++
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.

**Introducing functions, step 2**
7.1. Function definition and call

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, and out of the main program. 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 `(float l, float r)` is called the ‘parameter list’: it says that the function takes an `int` as input. For purposes of the function, the `int` will have the name `n`, but this is not necessarily the same as the name 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.

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:

\[
\begin{align*}
l &= \text{new}_l(l, \text{mid}, \text{fmid}) \\
r &= \text{new}_r(r, \text{mid}, \text{fmid})
\end{align*}
\]

You can base this off the file `bisect.cxx`.

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

### 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. The two occurrences of the function code do not have to be identical:

**Code reuse** Suppose you do the same computation twice:

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

With a function this can be replaced by:

```cpp
double computation(double in) {
    return .... computation from ‘in’ ....  
}
y = computation(x);  
w = computation(v);  
```

**Code reuse**  Example: multiple norm calculations:

Repeated code:

```cpp
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:

```cpp
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.
- Maintainence: 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.
### 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:

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

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

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.

### Function call

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:

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

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

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.

7.2.1 Another option for defining functions

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 called a function header or function prototype.

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

See chapter 19 for more details.

7.3 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.

```cpp
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;
}
```

```cpp
void print_result(int day, float value) {
    cout << "***************" << endl;
    cout << "* Output *" << endl;
    cout << "***************" << endl;
    cout << "The results for day " << day << ":" << endl;
    cout << " " << value << endl;
}
```
7.4 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.

7.4.1 Pass by value

The following style of programming is very much inspired by mathematical functions, and is known as functional programming\(^1\).

- A function has one result, which is returned through a return statement. The function call then looks like
  \[ y = f(x_1,x_2,x_3); \]
- Example:

```cpp
double squared(double x) {
    double y = x*x;
    return y;
}

/* ... */
number = 5.1;
cout << "Input starts as: " << number << endl;
other = squared(number);
cout << "Output var is: " << other << endl;
cout << "Input var is now: " << number << endl;
```

Output [func] passvalue:

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

---

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.
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:

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

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

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

and a program that uses them to swap two values:

```cpp
int i = 5, j = 17;
... biggest(i,j) ...
... smallest(i,j) ...
if (i==17 && j==5)
    cout << "Correct!" << endl;
```

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:

```cpp
void change_scalar(int i) {
    i = 1;
}
/* ... */
number = 3;
cout << "Number is 3: "
    << number << endl;
change_scalar(number);
cout << "is it still 3? Let’s see: "
    << number << endl;
```
Exercise 7.3. If you are doing the prime numbers project (chapter 52) you can now do exercise 52.6.

Background: square roots by Newton’s method  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
\[
    x_{\text{next}} = x - \frac{f_y(x)}{f'_y(x)}
\]
until the guess is accurate enough, that is, until \( f_y(x) \approx 0 \).

Exercise 7.4.

- Write functions \( f(x,y) \) and \( \text{deriv}(x,y) \), that compute \( f_y(x) \) and \( f'_y(x) \) for the definition of \( f_y \) above.
- Read a value \( y \) and iterate until |\( f(x,y) \)| < 10\(^{-5} \). Print \( x \).
- Second part: write a function \text{newton_root} that computes \( \sqrt{y} \).

7.4.2 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.

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

Code:

```cpp
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << endl;
i *= 2;
cout << i << "," << ri << endl;
ri -= 3;
cout << i << "," << ri << endl;
```

Output

```
[basic] ref:
5,5
10,10
7,7
```

(You will not use references often this way.)
7. Functions

**Reference create by initialize** Correct:

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

Not correct:

```c
float x{1.5};
float &xref;
xref = x;
float &threeref = 3; // WRONG: only reference to 'lvalue'
```

**Advanced note:** If you already know about pointers, you may wonder about the similarities.

- There are no ‘null’ references. There is a `nullptr`.
- References are bound when they are created.
- You can not change what a reference is bound to; the pointer target can change.
- Reference syntax is cleaner.
- Pointer use has implications about ownership: use only for specific purposes.

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.

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

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

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.
7.4. Parameter passing

Pass by reference example 1

Code:

```c++
void f(int &i) {
    i = 5;
}
int main() {
    int var = 0;
    f(var);
    cout << var << endl;
}
```

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.

*Advanced note:* There are other solutions for this. You could for instance use tuples (section 24.4), or, in C++17, `variant` (section 24.5.3).

Pass by reference example 2

```c++
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’ ....
}
```

However, see `std::optional`; section 24.5.2.

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

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.

Exercise 7.7. If you are doing the geometry project, you should now do the exercises in section 53.1.

7.5 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

\[
\begin{align*}
    f_0 &= 1, \\
    \forall n > 0: f_n &= n \times f_{n-1}.
\end{align*}
\]

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

\[
    F(n) = n \times F(n-1) \quad \text{if } n > 0, \text{ otherwise 1}
\]

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

\[
\begin{align*}
   \text{int factorial(int n)}
\end{align*}
\]

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

\[
\begin{align*}
   \text{int factorial(int n) \{} \\
   \text{    return n*factorial(n-1);} \\
   \text{\} // almost correct, but not quite}
\end{align*}
\]

So what happens if you write

\[
\begin{align*}
   \text{int f3; f3 = factorial(3);} \\
\end{align*}
\]
7.5. Recursive functions

Well,

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

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

- Now \( \text{factorial}(0) \) is 1, so \( \text{factorial}(1) \) is \( 1 \times \text{factorial}(0) \), which is 1,...
- ... so \( \text{factorial}(2) \) is 2, and \( \text{factorial}(3) \) is 6.

**Exercise 7.8.** The sum of squares:

\[
S_n = \sum_{n=1}^{N} n^2
\]

can be defined recursively as

\[
S_1 = 1, \quad S_n = n^2 + S_{n-1}.
\]

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

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

```
[func] mult:
1: warning: directory not found for option '-L/Library/
Developer/CommandLineTools/
SDKs/MacOSX.sdk/usr/lib'
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.
Exercise 7.10. 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:
  $x$ plus the multiplication $x \times (n - 1)$

Extend the code of exercise 7.9 to implement this.

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

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

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

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](#).

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 67.3.1.

7.6 Other function topics

7.6.1 Math functions

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

```cpp
#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:

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

(see section 14.2.)
7.6.2 Default arguments

Default arguments Functions can have default argument(s):

```c
double distance(double x, double y=0.) {
    return sqrt((x-y)*(x-y));
}
```

...  
```c
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.

Useful idiom Don’t trace a function unless I say so:

```c
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
```

7.6.3 Polymorphic functions

Polymorphic functions You can have multiple functions with the same name:

```c
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.

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

7.6.4 Stack overflow

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

$$\forall_{n\geq1}: g_n = (n-1) \cdot g(n-1), \quad g(1) = 1$$

and its implementation:

```c
int multifact(int n) {
    if (n==1)
        return 1;
    else {
        int oneless = n-1;
```
7. Functions

```cpp
    return oneless*multifact(oneless);
}
```

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

```cpp
    int oneless = 2;
```

and then the computation of \( g_2 \). But that computation involved

```cpp
    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, 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.

7.7 Review questions

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

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

```cpp
int add1(int i) {
    return i+1;
}
```

```cpp
void add1(int i) {
    i = i+1;
}
```

```cpp
int main() {
    int i=5;
    i = add1(i);
    cout << i << endl;
}
```

```cpp
int main() {
    int i=5;
    add1(i);
    cout << i << endl;
}
```

---

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

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

Review 7.4. Suppose a function

```c
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

```c
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

```c
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

```c
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.
7. Functions
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.

```cpp
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:
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 variables i in the conditionals.

This is independent of dynamic / runtime behaviour!

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

```cpp
#include <iostream>

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

int main() {
  bool something bigint;
  something;
  cout << "Hello world!";
}
```

**Output**

[basic] shadowtrue:

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

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

```cpp
#include <iostream>

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

int main() {
  bool something bigint;
  something;
  cout << "Hello world!";
}
```

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

```cpp
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:

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

```cpp
def 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,

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

you need a forward declaration

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

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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.

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

For example

```cpp
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:

```c
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:

```c
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:
class SomeObject {  
public:  
    SomeObject() {  
        cout << "calling the constructor" << endl;
    }  
    ~SomeObject() {  
        cout << "calling the destructor" << endl;
    }  
};
```

```
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
8. Scope

Review 8.1.  Is this a valid program?

```c
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:

```c
#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:

```c
#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:

```c
#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: objects belonging to classes.

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 members of the object; and
- to make these actions possible the object probably stores data, the data members.
- Objects comes in classes. A class is like a datatype: you can make objects of a class like 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.)
- 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.

**Object functionality** Small illustration: point objects.

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

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?

**The object workflow**

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

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

  (details later) typically before the `main`.

- You create specific objects with a declaration

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

- You let the objects do things:

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

**Private and public** Best practice we will use:

```cpp
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.

Let’s now introduce the details of all these steps.
9.1. What is an object?

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:

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

The declaration of an object \( x \) of class \( \text{Point} \); the coordinates of the point are initially set to \( 1.5, 2.5 \).

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

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, you actually call this constructor.

Usually you write your own constructor, for instance to initialize data members. In the case of the class \( \text{Point} \) the function of the constructor is to take the coordinates of the point and to copy them to private members of the \( \text{Point} \) 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

**Food for thought: constructor vs data** The arguments of the constructor imply nothing about what data members are stored!

Example: create a vector from \( x, y \) cartesian coordinates, but store \( r, \theta \) polar coordinates:

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

```cpp
Victor Eijkhout 101
```
9. Classes and objects

\[
\theta = \text{atan2}(y/x);
\]

**Member visibility**
- 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 what its distance is 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. For an object method,

- it’s still 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:**

```cpp
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 pl(1.0,1.0);
float d = pl.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.*

**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. Initialization

There are various ways of setting the initial values of the data members of an object. You have already seen how the constructor can be used for that:

```cpp
Point( double in_x, double in_y ) {
    x = in_x; y = in_y;
};
```

9.1.5.1 Default values

But it can also make sense for objects to have default values if nothing else is specified. This can be done with setting default values on the data members:

**Member default values** Class members can have default values, just like ordinary variables:

```cpp
class Point {
    private:
        float x=3., y=.14;
    private:
        // et cetera
};
```

Each object will have its members initialized to these values.

9.1.5.2 Initialization in the constructor

If you want to set the values in the constructor, you can use the above mechanism, but the preferred way is by using *member initializers*. (See section 10.5 for why this is preferred.)

```cpp
class Point {
    private:
        double x,y;
    public:
        Point( double userx,double usery ) :
            x(userx),y(usery) {
        }
    }
```

You can even save yourself from having to think of too many names:

```cpp
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()
        << endl;
```

**Output**

```
[geom] pointinitxy:
p1 = 1,2
```
9. Classes and objects

The initialization $x(x)$ should be read as $\text{membername}(\text{argumentname})$. Yes, having $x$ twice is a little confusing.

**Constructors and contained classes** Finally, if a class contains objects of another class,

```cpp
class Inner {
public:
    Inner(int i) { /* ... */ }
};
class Outer {
private:
    Inner contained;
public:
};
```

then

**When are contained objects created?**

```cpp
Outer(int n) {
    contained = Inner(n);
};
```

1. This first calls the default constructor
2. then calls the $\text{Inner}(n)$ constructor,
3. then copies the result over the $\text{contained}$ member.

```cpp
Outer(int n) {
    : contained(Inner(n)) {
        /* ... */
    }
};
```

1. This creates the $\text{Inner}(n)$ object,
2. placed it in the $\text{contained}$ member,
3. does the rest of the constructor, if any.

**Remark 5** 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.

9.1.5.3 Aggregate initialization

You can initialize an object from a brace-delimited initializer list, and in particular use that list itself to stand for an object.

**Initializer lists** Initializer lists can be used as denotations.

```cpp
Point(float ux, float uy) {
    /* ... */
};
Rectangle(Point bl, Point tr) {
    /* ... */
};
Point origin(0., 0.);
Rectangle lielow( origin, {5,2} );
```

9.1.6 Methods

You have just seen examples of class methods: 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:

```cpp
class Point {
private:
  double x, y, alpha;
public:
  Point(double x, double y)
    : x(x), y(y) {
      alpha = // 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 determining by its methods.
- If you change the design of the class data, you need to change the constructor call.

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.

Methods that alter the object For instance, you may want to scale a vector by some amount:

```cpp
class Point {
  /* ... */
  void scaleby( double a ) {
    px *= a; py *= a; }
  /* ... */
};
/* ... */
Point pl(1., 2.);
cout << "pl to origin "
    << pl.length() << endl;
pl.scaleby(2.);
cout << "pl to origin "
    << pl.length() << endl;
```

Output [geom] pointscaleby:

```
p1 to origin 2.23607
p1 to origin 4.47214
```
9. Classes and objects

Exercise 9.6. Implement a method \texttt{shift\_right} for the \texttt{Point} class.

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:

\begin{verbatim}
class Point {
    /* ... */
    Point scale(double a) {
        auto scaledpoint = Point(x*a, y*a);
        return scaledpoint;
    } /* ... */
    cout << "p1 to origin "
        << p1.length() << endl;
    Point p2 = p1.scale(2.);
    cout << "p2 to origin "
        << p2.length() << endl;
}
\end{verbatim}

\textbf{Anonymous objects} Create a point by scaling another point:

\begin{verbatim}
new_point = old_point.scale(2.81);
\end{verbatim}

Two ways of handling the \texttt{return} statement:

\begin{verbatim}
Naive: Point Point::scale(double a) {
    Point scaledpoint = Point(x*a, y*a);
    return scaledpoint;
}

Better: Point Point::scale(double a) {
    return Point(x*a, y*a);
}
\end{verbatim}

Creates point, copies it to \texttt{new\_point}  
Creates point, moves it directly to \texttt{new\_point}

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 \textit{default constructor}. For example:

\begin{verbatim}
class IamZero {
    private:
    int i=0;
    public:
    void print() { cout << i; }
};
\end{verbatim}

You recognize the default constructor by the fact that an object is defined without any parameters.

\begin{verbatim}
IamZero zero;
zero.print();
\end{verbatim}
9.1. What is an object?

You can define a default constructor yourself, but the previous example had a defaulted default constructor: it acted like it had a constructor

```cpp
IamZero() {};
```

Bear this in mind as you study the following code:

```cpp
Point p1(1.,2.), p2;
cout << "p1 to origin " << p1.length() << endl;
p2 = p1.scale(2.);
cout << "p2 to origin " << p2.length() << endl;
```

With the `Point` class (and its constructor) as given above, this will give an error message during compilation. The reason is that

```cpp
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:

```cpp
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:

```cpp
Point() = default;
Point( double x, double y )
: x(x), y(y) {};
```

**Remark 6** 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?

```cpp
class MyClass {
public:
    MyClass() { cout << "Construct!" << endl; };
};

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. Classes and objects

9.1.8 Accessors

You may have noticed the keywords public and private. We made the data members private, and the methods public. Thinking back to structures, you could say that their data members were (by default) public. Why don’t we do that here?

Struct data is public:

```c
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) 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.

**Public versus private**

- 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 inadvertant 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.

**Access gone wrong** We make a class for points on the unit circle

You don’t want to be able to change just one of x, y!

```c
class UnitCirclePoint {
    private:
        float x,y;
    public:
        UnitCirclePoint(float x) {
            setx(x);
        }
        void setx(float newx) {
            x = newx; y = sqrt(1-x*x);
        }
}
```

In general: enforce predicates on the members.

9.1.9 Examples
9.2. Inclusion relations between classes

Classes for abstract objects  Objects can model fairly abstract things:

```
class Stream {
private:
    int last_result(0);
public:
    int next() {
        return last_result++; }
};

int main() {
    Stream ints;
    cout << "Next: " << ints.next() << endl;
    cout << "Next: " << ints.next() << endl;
    cout << "Next: " << ints.next() << endl;
}
```

Output

```
[object] stream:
Next: 0
Next: 1
Next: 2
```

Exercise 9.7.

• 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`.

Exercise 9.8.  If you are doing the prime project (chapter 52), now is a good time to do exercise in section 52.7.

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. Classes and objects

9.2.1 Literal and figurative has-a

Sometimes a class can behave as if it includes an object of another class, while not actually doing so. For instance, a line segment can be defined from two points

```cpp
class Segment { 
    private: 
        Point starting_point, ending_point; 
    } 
... 
int main() { 
    Segment somesegment; 
    Point somepoint = somesegment.get_the_end_point();
}
```
or from one point, and a distance and angle:

```cpp
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.

**Multiple constructors**

```cpp
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:

**Constructors in has-a case**

```cpp
class Segment { 
    private: 
        // up to you how to implement! 
    public: 
        Segment(Point start, float length, float angle) 
        { .... } 
        Segment(Point start, Point end) { ... }
```
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).

Examples for base and derived cases

- Base case: employee. Has: salary, employee number.
  Special case: manager. Has in addition: underlings.
- Base case: shape in drawing program. Has: extent, area, drawing routine.
  Special case: square et cetera; has specific drawing routine.

```cpp
class General {
protected: // note!
  int g;
public:
  void general_method() {};
};

class Special : public General {
public:
  void special_method() { ... g ... };
};
```

How do you define a derived class?

- You need to indicate what the base class is:

  ```cpp
  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 defined for the base class is available as a method 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. Example:

```cpp
class General {
public:
  General( double x, double y ) {};
};
class Special : public General {
public:
  Special( double x ) : General(x,x+1) {};
};
```

Access levels  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.
9.3. Inheritance

Exercise 9.10. If you are doing the geometry project, you can now do the exercises in section 53.4. See the discussion in section ??.

9.3.1 Methods of base and derived classes

Overriding methods

- 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:

```cpp
class Base {
public:
    virtual f() { ... };
};
class Deriv : public Base {
public:
    virtual f() override { ... };
};
```

Override and base method

```cpp
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

Base vs derived methods

- Method defined in base class: can be used in any derived class.
- Method defined 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
A class is called *abstract class* if it has virtual methods; pure virtual if all methods are virtual. You can not make abstract objects.

**Abstract classes**  Special syntax for *abstract method*:

```cpp
class Base {
public:
  virtual void f() = 0;
};
class Deriv : public Base {
public:
  virtual void f() { ... };
};
```

Example: using virtual class

```cpp
class VirtualVector {
private:
public:
  virtual void setlinear(float) = 0;
  virtual float operator[](int) = 0;
};
```

**Implementation**

```cpp
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 9.11.** Advanced! Write a small ‘integrator’ library for Ordinary Differential Equations (ODEs). 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 is pure virtual; then write `ExplicitIntegrator` and `ImplicitIntegrator` classes deriving from this.

```cpp
double stepsize = .01;
auto integrate_linear = ForwardIntegrator( [] (double x) { return x*x; }, stepsize );
```

You can hardcode the function to be integrated, or try to pass a function pointer.
9.3.3 Friend classes

Friend classes A friend class can access private data and methods even if there is no inheritance relationship.

```cpp
/* forward definition: */
class A;

class B {
  friend class A;
  private:
    int i;
};
class A {
  public:
    void f(B b) { b.i; }
};
```

9.3.4 Multiple inheritance

More

- 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 9.12. This is a good time to do the exercises in section 53.2.

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:

```cpp
class HasVector {
  private:
    vector<int> values;
  public:
    HasVector( vector<int> initvalues )
      : values( initvalues ) {};

    HasVector( vector<int> init, int size )
      : values( vector<int>(size) ) {
        int loc=0;
    }
};
```

Victor Eijkhout
9. Classes and objects

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
class has_int {
    private:
        int mine(1);
    public:
        has_int(int v) {
            cout << "set: " << v << endl;
            mine = v;
        }
        has_int( has_int &h ) {
            auto v = h.mine;
        }
}
```
9.4. More about constructors

```cpp
cout << "copy: " << v << endl;
mine = v; 
void printme() { cout
   "I have: " << mine << endl; }
}
```

The following code shows that the data got copied over:

```cpp
has_int an_int(5);
has_int other_int(an_int);
an_int.printme();
other_int.printme();
```

Output

```
[object] copyscalar:
set: 5
copy: 5
I have: 5
I have: 5
```

Copying is recursive  Class with a vector:

```cpp
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) << endl; }
};
```

Copying is recursive, so the copy has its own vector:

```cpp
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:
9. Classes and objects

Code:
```cpp
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor" << endl;
    }
    ~SomeObject() {
        cout << "calling the destructor" << endl;
    }
};
```

Output

```
[object] destructor:
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

Exercise 9.13. Write a class

```cpp
class HasInt {
private:
    int mydata;
public:
    HasInt(int v) { /* initialize */ };
    ...}
```

used as

```cpp`
{
    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 ****
```

Destructors and exceptions  The destructor is called when you throw an exception:
9.5 Advanced topics

The remainder of this section is advanced material. Make sure you have studied section 15.3.

9.5.1 Class prototypes

For purposes of organizing your code, you may sometimes not want to include the full code of a method in the class definition. It is in fact possible to write a class definition, giving only the prototype of a class method, and specify the full method later or elsewhere.

```
class Point {
private:
    float x, y;
public:
    Point(float x, float y);
    float distance();
};
```

- 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.)

9.5.2 Returning by reference

Direct alteration of internals Return a reference to a private member:

```
class Point {
private:
    double x, y;
    Point &Point() {
    : x(x), y(y) {
        float Point::distance() {
            return sqrt(x*x + y*y);
        }
    }
```
9. Classes and objects

```cpp
public:
    double &x_component() { return x; };
};

int main() {
    Point v;
    v.x_component() = 3.1;
}
```

Only define this if really needed.

Reference to internals Returning a reference saves you on copying. Prevent unwanted changes by using a ‘const reference’.

```cpp
class Grid {
    private:
        vector<Point> thepoints;
    public:
        const vector<Point> &points() { return thepoints; };
    };  
    int main() {
        Grid grid;
        cout << grid.points()[0];
        // grid.points()[0] = whatever ILLEGAL
    }
```

9.5.3 Accessor functions

It is a good idea to make the data in an object private, to control 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 by whom.

Accessor functions:

```cpp
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:

```cpp
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 nonpositive.

Having two accessors can be a little clumsy. Is it possible to use the same accessor for getting and setting?
9.5. Advanced topics

Setting members through accessor  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() << endl;
  myobject.xvalue() = 3.;
  cout << "Object member updated :" << myobject.xvalue() << endl;
}
```

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 changable!

9.5.4 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*.

9.5.5 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*.

The syntax for this is

Operator overloading

```
<returntype> operator<op>(<argument>) {
  <definition>
}
```

For instance:
9. Classes and objects

**Exercise 9.14.** 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 9.15.** If you know about templates, you can do the exercises in section 22.3.

**9.5.5.1 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:**
```cpp
class IntPrintFunctor {
public:
    void operator()(int x) {
        cout << x << endl;
    }
};
/* ... */
IntPrintFunctor intprint;
intprint(5);
```

**Output**
```
[object] functor:
5
```

**Exercise 9.16.** 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:**
```cpp
IntPrintTimes printx2(2);
printx2(1);
for ( auto i : {5,6,7,8} )
    printx2(i);
```

**Output**
```
[object] functor2:
2
10
12
14
16
```
9.5. Advanced topics

9.5.5.2 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 simply is not defined with your class as second operand.

Here we solve this in two steps:

String an object

1. Define a function that yields a string representing the object, and
2. redefine the less-less operator to use this.

```cpp
#include <sstream>
using std::stringstream;
#include <string>
using std::string;

/* ... */
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;
}
/* ... */
Point pl(1.,2.);
cout << "pl " << pl
   << " has length " << pl.length() << endl;
```

(See section 11.3 for the `stringstream` header.)

9.5.6 ‘this’ pointer

‘this’ pointer to the current object Inside an object, a pointer to the object is available as `this`:

```cpp
class Myclass {
private: 
    int myint;
public: 
    Myclass(int myint) {
        this->myint = myint; // ‘this’ redundant!
    }
};
```

‘this’ use You don’t often need the `this` pointer. Example: you need to call a function inside a method that needs the object as argument:

```cpp
/* forward definition: */
class someclass;

/* ... */
void somefunction(const someclass &c) {
    /* ... */
}
```

Victor Eijkhout
There is another interesting idiom that uses the ‘this’ pointer. Define a simple class:

```cpp
class number {
    private:
        float x;
    public:
        number(float x) : x(x) {};
        float value() { return x; };
}
```

Defining a method:

```cpp
number addcopy(float y) {
    x += y;
    return *this;
}
```

both alters the object, and returns a copy.

Changing the method to return a reference:

```cpp
number& add(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:

```cpp
Code:
number mynumber(1.0);
mynumber.add(.5);
cout << mynumber.value() << endl;
mynumber.multiply(2.).add(1.).multiply(3.);
cout << mynumber.value() << endl;
```

### 9.5.7 Static members

**Static class members** A static member acts as if it’s shared between all objects.

(Note: C++17 syntax)
9.6. Review question

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.
9. Classes and objects
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.

\textit{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 10.8.2.

10.1 Some simple examples

10.1.1 Vector creation

To use vectors, you first need the `vector` header from the Standard Template Library (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.

\textbf{Short vectors}  Short vectors can be created by enumerating their elements:

```cpp
#include <vector>
using std::vector;

int main() {
    vector<int> evens{0, 2, 4, 6, 8};
    vector<float> halves = {0.5, 1.5, 2.5};
    cout << evens.at(0) << endl;
    return 0;
}
```

---

1. The term ‘array’ is used informally here. There is an `array` keyword, which is briefly discussed in section 10.8.4.
Exercise 10.1.

1. Take the above snippet, supply the missing header lines, 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.

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.)

Vector constant initialization There is a syntax for initializing 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:

```
{ vector<int> numbers{5,6,7,8,9,10};
  cout << numbers.at(3) << endl;
}
{ vector<int> numbers = {5,6,7,8,9,10};
  numbers.at(3) = 21;
  cout << numbers.at(3) << endl;
}
```

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`. 

Introduction to Scientific Programming
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 \texttt{at} method:

\begin{verbatim}
vector<int> numbers = {1,4};
numbers.at(0) += 3;
numbers.at(1) = 8;
cout << numbers.at(0) << "," << numbers.at(1) << endl;
\end{verbatim}

\textbf{Output} \[\text{array] assignatfun: 4,8\]

2. There is also a short-hand notation (which is the same as in C):

\begin{verbatim}
vector<int> numbers = {1,4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << "," << numbers[1] << endl;
\end{verbatim}

\textbf{Output} \[\text{array] assignbracket: 4,8\]

Indexing starts at zero. Consequently, a vector declared as

\texttt{vector<int> ints(N)}

has elements 0,...,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

\begin{verbatim}
vector<float> x(25);
x.at(2) = 3.14;
float y = y.at(2);
\end{verbatim}

- The same holds for indexing with square brackets.

\begin{verbatim}
vector<float> x(25);
x[2] = 3.14;
float y = y[2];
\end{verbatim}

- 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?

\begin{verbatim}
vector<float> x(6); // size 6, index ranges 0..5
x.at(6) = 5.; // oops!
i = -2;
x[i] = 3; // also oops, but different.
\end{verbatim}
10. Arrays

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.

**Vector index out of bounds**  Indexing out of bounds can go undetected for a while:

```cpp
vector<float> v(10, 2);
for (int i = 5; i < 6; i--)
    cout << "element " << i << " is " << v[i] << endl;
```

```
array] segmentation:
element -2905 is 0
element -2906 is 0
element -2907 is 0
element -2908 is 1.41531e-43
element -2909 is 1.8519e+28
element -2910 is 7.31695e+28
element -2911 is 2.56297e+08
element -2912 is 2.65628e+20
/bin/sh: line 1: 92816
Segmentation fault: 11 ./
```

**Review 10.2.** The following codes are not correct in some sense. How will this manifest itself?

```cpp
vector<int> a(5);
a[6] = 1.;
```

```cpp
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.

**A philosophical point**

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 mechanisms.

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’.

**Range over elements** You can write a range-based `for` loop, which considers the elements as a collection.

```cpp
for ( float e : my_data )
    // statement about element e
for ( auto e : my_data )
    // same, with type deduced by compiler
```

(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 lists* can also be used as a list denotation:

**Range over vector denotation**

```cpp
for ( auto i : {2,3,5,7,9} )
cout << i << ",";
cout << endl;
```

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.
10. Arrays

**Indexing the elements** You can write an *indexed for* loop, which uses an index variable that ranges from the first to the last element.

```cpp
for (int i = 0; i < numbers.size(); i++)
    // statement about index i
```

Example: find the maximum element in the vector, and where it occurs.

**Code:**

```cpp
int tmp_idx = 0;
int tmp_max = numbers.at(tmp_idx);
for (int i = 0; i < numbers.size(); i++)
    if (v > tmp_max) {
        tmp_max = v;
        tmp_idx = i;
    }
```

```cpp
cout << "Max: " << tmp_max << " at index: " << tmp_idx << endl;
```

**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:

```cpp
vector<int> numbers = {1, -4, 2, -6, 5};
```

**Hint:**

```cpp
#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 over elements by reference** Range-based loop indexing makes a copy of the vector element. If you want to alter the vector, use a reference:

```cpp
for (auto &e : my_vector)
    e = ....
```
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 parametrized 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.

Vector copy 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) << endl;
```

```
Output
[array] vectorcopy:
3.5,7
```

(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 52.16.

In a while loop, if you need an index, you need to maintain that index explicitly. There are then certain common idioms.

Example of increment indexing

```
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 << endl;
```

```
Output
[loop] plusplus:
The first even number
appears at index 4
```

Exercise 10.7. Exercise: modify the preceeding code so that after the while loop index is the number of leading odd elements.

Victor Eijkhout
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.

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?

Your first encounter with templates `vector` is a ‘templated class’: `vector<X>` is a vector-of-X.

Code behaves as if there is a class definition for each type:

```cpp
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.

Dynamic vector extension Extend a vector’s size with `push_back`:
### 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.
### 10.4.1 Pass vector to function

The mechanisms of parameters passing (section 7.4) apply to vectors too.

*C difference:* There is a big difference here between C++ vectors and C arrays!

**Vector as function argument** You can pass a vector to a function:

```cpp
double slope(vector<double> v) {
    return v.at(1)/v.at(0);
};
```

Vectors, like any argument, are passed by value, so the vector is actually copied into the function.

*C difference:* In C the array is not copied: you pass the address by value. Not the contents.

**Vector pass by value example**

**Code:**

```cpp
void set0(vector<float> v, float x) {
    v.at(0) = x;
} /* ... */
vector<float> v1;
v1.at(0) = 3.5;
set0(v1,4.6);
cout << v1.at(0) << endl;
```

- Vector is copied
- ‘Original’ in the calling environment not affected
- Cost of copying?

**Output**

[array] vectorpassnot:

```
3.5
```

**Exercise 10.9.** Revisit exercise 10.8 and introduce a function for computing the $L_2$ norm.

**Vector pass by reference** If you want to alter the vector, you have to pass by reference:

**Code:**

```cpp
void set0(vector<float> &v, float x) {
    v.at(0) = x;
} /* ... */
vector<float> v1;
v1.at(0) = 3.5;
set0(v1,4.6);
cout << v1.at(0) << endl;
```

- Parameter vector becomes alias to vector in calling environment
  ⇒ argument can be affected.

**Output**

[array] vectorpassref:

```
4.6
```
10.4. Vectors and functions

- No copying cost
- What if you want to avoid copying cost, but need not alter the argument?

**Vector pass by const reference**  Passing a vector that does not need to be altered:

```cpp
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.

### 10.4.2 Vector as function return

**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:

```cpp
vector<int> make_vector(int n) {
    vector<int> x(n);
    x.at(0) = n;
    return x;
}
```

```cpp
/* ... */
vector<int> x1 = make_vector(10);
// "auto" also possible!
    cout << "x1 size: " << x1.size() << endl;
    cout << "zero element check: " << x1.at(0) << endl;
```

**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:

```cpp
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):
Find arguments for/against that approach.

(Note: C++ has sorting functions built in.)

(See section 24.8.1 for the random function.)

**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?

## 10.5 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
10.5. Vectors in classes

- The constructor first creates \texttt{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 \textit{member initializer} list:

\textbf{Create the contained vector} Use initializers for creating the contained vector:

```cpp
class named_field {
private:
  string name;
  vector<double> values;
public:
  named_field(string name, int nelements) :
    name(name),
    values(vector<double>(n)) {}
};
```

Less desirable method is creating in the constructor:

```cpp
named_field(string name, int nelements) {
  values = vector<double>(n);
};
```

10.5.1 Timing

Different ways of accessing a vector can have drastically different timing cost.

\textbf{Filling in vector elements} You can push elements into a vector:

```cpp
vector<int> flex;
for (int i=0; i<LENGTH; i++)
  flex.push_back(i);
```

If you allocate the vector statically, you can assign with \texttt{at}:

```cpp
vector<int> stat(LENGTH);
for (int i=0; i<LENGTH; i++)
  stat.at(i) = i;
```

\textbf{Filling in vector elements} With subscript:

```cpp
vector<int> stat(LENGTH);
for (int i=0; i<LENGTH; i++)
  stat[i] = i;
```

You can also use \texttt{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:

<table>
<thead>
<tr>
<th>Timing the ways of filling a vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible time: 2.445</td>
</tr>
<tr>
<td>Static at time: 1.177</td>
</tr>
<tr>
<td>Static assign time: 0.334</td>
</tr>
<tr>
<td>Static assign time to new: 0.467</td>
</tr>
</tbody>
</table>

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.

10.6 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.

```cpp
class namedvector {
private:
    string name;
    vector<int> values;
public:
    namedvector(int n, string name="unnamed")
        : name(name), values(vector<int>{n}) {};
    string rendered() {
        string render{name};
        render += ":";
        for (auto v : values) 
            render += "\" + atoi(v) + \",\";
        return render;
    }
    /\* ... */
};
```

Unfortunately this means you may have to recreate some methods:

```cpp
int &at(int i) {
    return values.at(i);
}
```
10.7 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.

10.7.1 Matrix as vector of vectors

**Multi-dimensional vectors** Multi-dimensional is harder with vectors:

```cpp
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:**

```cpp
// alternative:
vector<vector<float>> rows(10);
for (auto &row : rows )
    row = vector<float>(20);
```

**Matrix class**

```cpp
class matrix {
    vector<vector<double>> elements;
private:
    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.
10. Arrays

**Code:**

```cpp
A.set(3.);
cout << "Sum of elements: "
    << A.totalsum() << endl;
```

**Output**

[array] matrixsum:

```
Sum of elements: 30
```

You can base this off the file `matrix.cxx`.

**Exercise 10.15.** Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

### 10.7.2 A better matrix class

You can make a ‘pretend’ matrix by storing a long enough `vector` in an object:

```cpp
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;
```

### 10.8 Advanced topics

#### 10.8.1 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 (see section 14.1.1 for details) a pointer to a vector element. Mirroring the index-loop convention of

\[
\text{for (int } i=0; i<\text{hi}; i++)
\text{ element = vec.at(i);}
\]

you can iterate:

\[
\text{for (auto } elt_ptr=\text{vec.begin(); } elt_ptr!=\text{vec.end(); } ++elt_ptr)
\text{ element = *elt_ptr;}
\]

Some remarks:
- This is one of the very few places where you need the asterisk in C++ for dereferencing; section 17.2. (However, the thing you are dereferencing is an iterator, not a pointer.)
- 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.

Another illustration of pointer arithmetic on iterators is getting the last element of a vector:

\[
\begin{align*}
\text{Code:} & \quad \text{Output} \\
\text{vector<int> mydata(5,2);} & \quad [\text{array}] \text{ vectorend:} \\
\text{mydata.push_back(35);} & \quad 6 \\
\text{cout }\ll:\text{mydata.size()}\ll\text{endl;} & \quad 35 \\
\text{cout }\ll:\text{mydata[mydata.size()-1]} & \quad \ll\text{endl;} \\
\end{align*}
\]

\[
\begin{align*}
\text{Code:} & \quad \text{Output} \\
\text{vector<int> mydata(5,2);} & \quad [\text{array}] \text{ vectorenditerator:} \\
\text{mydata.push_back(35);} & \quad 6 \\
\text{cout }\ll:\text{mydata.size()}\ll\text{endl;} & \quad 35 \\
\text{cout }\ll:*\text{(--mydata.end()) }\ll\text{endl;} & \\
\end{align*}
\]

### 10.8.2 Old-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.

\[
\begin{align*}
\text{Code:} & \quad \text{Output} \\
\{ & \quad [\text{array}] \text{ staticinit:} \\
\text{int } \text{numbers[]} = \{5,4,3,2,1\}; & \quad 2 \\
\text{cout }\ll:\text{numbers[3]}\ll\text{endl;} & \quad 21 \\
\} & \\
\text{int } \text{numbers[5]}{5,4,3,2,1}; & \\
\text{numbers[3]} = 21; & \\
\text{cout }\ll:\text{numbers[3]}\ll\text{endl;} & \\
\}
\end{align*}
\]
Arrays

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.

Range-based indexing works the same as with vectors:

```cpp
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)" << endl;
```

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?

```cpp
int a[5];
a[6] = 1.1;

int a[5];
a.at(6) = 1.1;
```

10.8.2.1 C-style arrays and subprograms

Arrays can be passed to a subprogram, but the bound is unknown there.

```cpp
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] << endl;
```

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.

10.8.2.2 Multi-dimensional arrays

Multi-dimensional arrays can be declared and used with a simple extension of the prior syntax:
10.8. Advanced topics

```cpp
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:

```cpp
void print12( int ar[][6] ) {
    cout << "Array[1][2]: " << ar[1][2] << endl;
    return;
}

// ... */
int array[5][6];
array[1][2] = 3;
print12(array);
```

C/C++ row major

| (1,1) | (1,2) | (2,1) | (3,1) |

Physical:

| (1,1) | (1,2) | (2,1) | (3,1) |

10.8.2.3 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:

```cpp
void print06( int ar[][6] ) {
    cout << "Array[0][6]: " << ar[0][6] << endl;
    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.

10.8.3 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.
• The heap is for entities that transcend scope, for instance because they are pointed-to by a pointer. The space of such objects can be returned to the free store at any time, so the heap can suffer from fragmentation.

The concepts of stack and heap do not appear in the C++ standard. However, the following description generally applies:

• Objects that obey scope are allocated on the stack, so that their memory is automatically freed control leaves the scope.
• 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 leaks in languages such as C. This danger is greatly lessened in C++.

While in C the only way to create dynamic objects is by a call to malloc, in C++ a vector obeys scope, and therefore lives on the stack. If you wonder if this may lead to stack overflow, rest assured: only the descriptor, the part of the vector that remembers the size, is on the stack, while the actual data is on the heap. However, it is no longer subject to memory leaking, since the heap storage is deallocated when the vector object goes out of scope.

If you want heap memory that transcends scope you can use the smart pointer mechanism, which also guarantees against memory leaks. See chapter 16.

10.8.4 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.

10.8.5 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;
```

In C++ you can write
10.9. 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 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

\[ p_{r,c} = \binom{r}{c} = \frac{r!}{c!(r-c)!} \]
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 \( n \times n \) 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 6.12, 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’
10. Arrays
Chapter 11

Strings

11.1 Characters

Characters and ints

- Type `char`;
- represents ‘7-bit ASCII’: printable and (some) unprintable characters.
- Single quotes: `char c = ‘a’`

Char / int equivalence Equivalent to (short) integer:

```
char ex = ‘x’;
int x_num = ex, y_num = ex+1;
char why = y_num;
cout << "x is at position " << x_num
   << endl;
cout << "one further lies " << why
   << endl;
```

Output

```
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 · · · 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
11. Strings

**String declaration**

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

// .. and now you can use 'string'
```

(Do not use the C legacy mechanisms.)

**String creation** A string variable contains a string of characters.

```cpp
string txt;
```

You can initialize the string variable or assign it dynamically:

```cpp
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?

**Quotes in strings**

You can escape a quote, or indicate that the whole string is to be taken literally:

```cpp
Code:
string one("a b c"),
two("a \"b\" c"),
three(R"\"a \"b \"c\")
cout << one << endl;
cout << two << endl;
cout << three << endl;
```

**Output**

```
[string] quote:
a b c
"a "b" c
"a ""b ""c
```

**Concatenation** Strings can be concatenated:

```cpp
Code:
string my_string, space(" ");
my_string = "foo";
my_string += space + "bar";
cout << my_string << ": " << my_string.size() << endl;
```

**Output**

```
[string] stringadd:
foo bar: 7
```

**String indexing** You can query the size:

```cpp
Code:
string five_text("fiver");
cout << five_text.size() << endl;
```

**Output**

```
[string] stringsize:
5
```
11.2. Basic string stuff

Code:
```cpp
type digits("0123456789");
cout "char three: "
<< digits[2] << endl;
cout "char four: "
<< digits.at(3) << endl;
```

Output

```
[string] stringsub:
char three: 2
char four: 3
```

Ranging over a string Same as ranging over vectors.

Range-based for:

Code:
```cpp
cout "By character: ";
for (char c: abcd)
cout c << " ";
cout endl;
```

Output

```
[string] stringrange:
By character: a b c
```

Ranging by index:

Code:
```cpp
type abc = "abc";
cout "By character: ";
for (int ic=0; ic<abc.size(); ic++)
cout abc[ic] << " ";
cout endl;
```

Output

```
[string] stringindex:
By character: a b c
```

Range with reference Range-based for makes a copy of the element
You can also get a reference:

Code:
```cpp
for (char &c: abc)
c = c + 1;
cout "Shifted: " << abc << endl;
```

Output

```
[string] stringrangeset:
Shifted: bcd
```

Iterating over a string

```cpp
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
11. Strings

**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?

**More vector methods** Other methods for the vector class apply: *insert, empty, erase, push_back*, et cetera.

**Code:**

```cpp
string five_chars;
for (int i=0; i<5; i++)
    five_chars.push_back( ' ');
cout << five_chars.size() << endl;
```

**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:

```cpp
#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. C++11 has a mechanism for raw string literals.

In its simplest form:

```cpp
cout << R"(string with }
{ \weird\ stuf)" << "\n";
```

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 multicharacter delimiter:

```cpp
cout << R"(limit{"string with }
{ \weird\ stuff})"limit" << "\n";
```

11.4.2 Conversation to/from 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.
- The Boost library has a lexical cast.
11. Strings

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 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 will have a new and improved mechanism that is as powerful as printf, but considerably more elegant.

Formatted output

- cout uses default formatting
- Possible: pad a number, use limited precision, format as hex, etc
- Many of these output modifiers need

```
#include <iomanip>
```

Normally, output of numbers takes up precisely the space that it needs:

```
for (int i=1; i<200000000; i*=10)
    cout << "Number: " << i << endl;
```

Output

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
```

157
12. Input/output

**Reserve space**  You can specify the number of positions, and the output is right aligned in that space by default:

**Code:**

```cpp
#include <iomanip>
using std::setw;
/* ... */
cout << "Width is 6:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
    << setw(6) << i << " endl;
// 'setw' applies only once:
cout << "Width is 6:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
    << setw(6) << i << " endl;
```

**Output**

```
[io] width:
Width is 6:
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

**Padding character**  Normally, padding is done with spaces, but you can specify other characters:

**Code:**

```cpp
#include <iomanip>
using std::setfill;
using std::setw;
/* ... */
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
    << setfill('.')
    << setw(6) << i << " endl;
```

**Output**

```
[io] formatpad:
Number: ..1
Number: ..10
Number: ..100
Number: ..1000
Number: .1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
```

Note: single quotes denote characters, double quotes denote strings.

**Left alignment**  Instead of right alignment you can do left:

**Code:**

```cpp
#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 << " endl;
```

**Output**

```
[io] formatleft:
Number: 1....
Number: 10....
Number: 100....
Number: 1000....
Number: 10000.
Number: 100000.
Number: 1000000.
Number: 10000000.
```
Finally, you can print in different number bases than 10:

```cpp
#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 << endl;
}
```

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
```

**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
```

**Hexadecimal** Hex output is useful for addresses (chapter 17.2):
12. Input/output

```cpp
int i;
cout << "address of i, decimal: " << (long)&i << endl;
cout << "address of i, hex : " << std::hex << &i << endl;
```

Output

[pointer] coutpoint:
- address of i, decimal: 140732703427524
- address of i, hex : 0x7ffee2cbcbc4

Back to decimal:

```cpp
cout << hex << i << dec << j;
```

There is no standard modifier for outputting as binary.

### 12.1.1 Floating point output

**Floating point precision** Use `setprecision` to set the number of digits before and after decimal point:

```cpp
#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 << endl;
    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.

**Fixed point precision** Fixed precision applies to fractional part:

```cpp
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
    cout << setprecision(4) << x << endl;
    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)
Aligned fixed point output  Combine width and precision:

**Code:**

```cpp
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4) << x
         << endl;
    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:**

```cpp
string quasifix(double);
int main() {
    for (auto x : { 1.5, 12.32, 123.456, 1234.5678 })
        cout << quasifix(x) << endl;
}
```

**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.

Scientific notation  Combining width and precision:

**Code:**

```cpp
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4)
         << x << endl;
    x *= 10;
}
cout << endl;
```

**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

```cpp
ios::fmtflags old_settings = cout.flags();

cout.flags(old_settings);

int old_precision = cout.precision();
```
cout.precision(old_precision);

12.2 File output

Text output to file  The `iostream` is just one example of a `stream`: general mechanism for converting entities to exportable form. In particular: file output works the same as screen output.

Use:

```cpp
#include <fstream>
using std::ofstream;

ofstream file_out;
file_out.open("fio_example.out");
file_out << number << endl;
file_out.close();
```

Output

```
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’.

Binary output  Binary output: write your data byte-by-byte from memory to file. (Why is that better than a printable representation?)

```cpp
ofstream file_out;
file_out.open("fio_binary.out", ios::binary);
file_out.write((char*)(&number),4);
```

Output

```
fiobin: echo 25 | ./fiobin; od fio_binary.out
A number please: Written.
0000000 000031 000000 000004
```

12.3 Output your own classes

See section 9.5.5.2.

You have used statements like:

```cpp
cout << "My value is: " << myvalue << endl;
```

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.
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 \n. However, \n 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 \n 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 \n may slow down your program because of the flush it performs. More efficiently, you would add a newline character to the output directly:

```
sostream << "Value: " << x << '\n';
ostream << "Total " << nerrors << " reported\n";
```

In other words, use cout for regular output, cerr for logging output, and use \n instead of \n.
12.5 Input

The `cin` command can be used to read integers and floating point formats.

**Code:**

```cpp
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 \n      echo $n | ./cinfloat \n    ; 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.

**Better terminal input** 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:

```cpp
#include <sstream>
using std::cout;
using std::cin;
using std::endl;
/* ... */
std::string saymany;
int howmany;

cout << "How many times? ";
gtline( cin, saymany );
stringstream saidmany(saymany);
saidmany >> howmany;
```

You can not use `cin` and `getline` in the same program.


12.5.1 File input

**File input streams** Input file stream, method `open`, then use `getline` to read one line at a time:

```cpp
#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" << endl;
}

**End of file test** 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 that 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

Test, mostly for file streams: `is_eof is_open`
12. Input/output
Chapter 13

Lambda functions

The mechanism of lambda expressions (added in C++11) makes dynamic definition of functions possible.

A simple example You can define a function and apply it:

\[
\begin{align*}
\text{double } f(x) &\{ \text{ return } 2*x; \} \\
y &= f(3.7);
\end{align*}
\]

or you can apply the function recipe directly:

\[
\begin{align*}
y &= [] (\text{double } x) \rightarrow \text{double } \{ \text{ return } 2*x; \} (3.7);
\end{align*}
\]

This example is of course fairly pointless, but it illustrates the syntax of a lambda function:

\[
\begin{align*}
\{ \text{capture} \} (\text{inputs}) &\rightarrow \text{outtype } \{ \text{definition} \}; \\
\{ \text{capture} \} (\text{inputs}) &\{ \text{definition} \};
\end{align*}
\]

- 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 functions are sometimes called closures, but this term has a technical meaning in programming language theory, that only partly coincides with C++ lambda functions.

For a slightly more useful example, we can assign the lambda function to a variable, and repeatedly apply it.

Assign lambda to variable

\[
\begin{align*}
\text{auto } f &= [] (\text{double } x) \rightarrow \text{double } \{ \text{ return } 2*x; \}; \\
y &= f(3.7); \\
z &= f(4.9);
\end{align*}
\]

- This is a variable declaration.
- Uses auto for technical reasons.
13. Lambda functions

- See different approach below.

**Exercise 13.1.** Do exercise 54.8 of the zero finding project.

## 13.1 Passing lambdas around

Above, when we assigned a lambda function to a variable, we used `auto` for the type. The reason for this is that each lambda function gets its own unique type, that is dynamically generated. But that makes it hard to pass that variable to a function: what type do we declare the parameter to be?

The way out is to use the `functional` header:

```cpp
#include <functional>
using std::function;
```

With this, you can declare parameters by their signature:

```cpp
double find_zero(
    function<double(double)>&& f,
    function<double(double)>&& fprime
) {
```

**Exercise 13.2.** Do exercise 54.9 of the zero-finding project.

### 13.1.1 Lambda members of classes

The fact that lambda function have a dynamically generated type also makes it hard to store a lambda in an object. Again we use `std::function`.

**Lambda in object** A set of integers, with a test on which ones can be admitted:

```cpp
#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;
        };
};
```

**Illustration**
13.2 Captures

A non-trivial use of lambdas uses the capture to fix one argument of a function. Let’s say we want a function that computes exponentials for some fixed exponent value. We take the `cmath` function

```cpp
pow( x, exponent );
```

and fix the exponent:

```cpp
int exponent=5;
auto powerfive =
    [exponent] (float x) -> float {
        return pow(x,exponent);
    };
```

Now `powerfunction` is a function of one argument, which computes that argument to a fixed power.

**Exercise 13.3.** Do exercises 54.10 and 54.11 of the zero-finding project.

13.3 Capture by reference

Normally, captured variables are copied by value.
13. Lambda functions

Code:

```cpp
int one = 1;
auto one_more = [
    one
] (int input) -> void {
    cout << input + one << endl;
};
one_more(5);
one_more(6);
one_more(7);
```

Output

```cpp
[func] lambdatavalue:
lambdacapture.cxx:67:9: error
    : expected expression
auto [=] (float x) -> double { /* stuff */ };
^
lambdacapture.cxx:67:14: error: expected unqualified-id
data [=] (float x) ->
double { /* stuff */ };
^
lambdacapture.cxx:67:14: error: expected ')'
lambdacapture.cxx:67:12: note: to match this '{'
auto [=] (float x) ->
double { /* stuff */ };
^
6 errors generated.
make[1]: *** [lambdacapture.o ] Error 1
```

Attempting to change the captured variable doesn’t even compile: missing snippet lambdamutate

If you do want to alter the captured parameter, pass it by reference:

```
```
13.3. Capture by reference

**Code:**

```cpp
one = 1;
auto more_and_more = [&one] (int input) -> void {
    cout << input+one << endl;
    one++;
};
more_and_more(5);
more_and_more(6);
more_and_more(7);
cout << "one is now: " << one << endl;
```

**Output**

```
[func] lambdareference:
lambdacapture.cxx:67:9: error : expected expression
auto [=] (float x ) ->
double { /* stuff */ };  
^  
lambdacapture.cxx:67:14: error: expected unqualified-id
auto [=] (float x ) ->
double { /* stuff */ };  
^  
lambdacapture.cxx:67:14: error: expected ')'  
lambdacapture.cxx:67:12: note : to match this '('
auto [=] (float x ) ->
double { /* stuff */ };  
^  
lambdacapture.cxx:69:10: error: expected expression
auto [=] (float x ) ->
double { /* stuff */ };  
^  
lambdacapture.cxx:69:14: error: expected unqualified-id
auto [=] (float x ) ->
double { /* stuff */ };  
^  
lambdacapture.cxx:69:14: error: expected ')'  
lambdacapture.cxx:69:12: note : to match this '('
auto [=] (float x ) ->
double { /* stuff */ };  
^  
6 errors generated.
make[1]: *** [lambdacapture.o ] Error 1
```

**Capture a reduction variable**  This mechanism is useful

```cpp
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;
```
13. Lambda functions

13.4 More

13.4.1 Making lambda stateful

Code:
```cpp
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 << endl;
```

Output

```
[stl] counteach:
number of even: 3
```

Lambda functions are normally stateless:

Code:
```cpp
float x = 2, y = 3;
auto f = [x] (float &y) -> void { int xx = f(y); 
cout << y << "\n"; 
f(y); 
}();
```

Output

```
[func] nonmutable:
 ld: warning: directory not found for option '-L/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/lib'
7
11
```

but with the `mutable` keyword you can make them stateful:

Code:
```cpp
float x = 2, y = 3;
auto f = [x] (float &y) mutable -> void { int xx = f(y); 
cout << y << "\n"; 
f(y); 
}();
```

Output

```
[func] yesmutable:
7
15
```

13.4.2 Generic lambdas

The `auto` keyword can be used, almost abused, for generic lambdas:

```cpp
|| auto compare = [] (auto a, auto b) { return a<b; };
```

Here the return type is clear, but the input types are generic.

13.4.3 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.
13.4. More

13.4.4 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.
13. Lambda functions
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.

**Iterator basics**

- Iterable 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.

```cpp
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
```

**Remark 11** The star operator for finding the value that an iterator ‘points to’ is very similar to the C dereference star. However, it is not the same: it is really a definition of the unary-star operator.

14.1.1 Iterators

The container class has a subclass *iterator* that can be used to iterate through all elements of a container. This was discussed in section 10.8.1.

However, an *iterator* can be used outside of strictly iterating. References to *begin* and *end* are common.

**Begin and end iterator** Use independent of looping:

Code:

```c++
vector<int> v(1,3,5,7);
auto pointer = v.begin();
cout << "we start at "
<< *pointer << endl;
pointer++;
cout << "after increment: "
<< *pointer << endl;
pointer = v.end();
cout << "end is not a valid element: "
<< *pointer << endl;
pointer--;
cout << "last element: "
<< *pointer << endl;
```

Output [stl] iter:

we start at 1
after increment: 3
end is not a valid element: 0
last element: 7

(Note: the auto actually stands for vector::iterator)

Note that the star notation is a unary star operator, not a pointer dereference:

```c++
vector<int> vec(11,22,33,44,55,66);
auto second = vec.begin(); second++;
cout << "Dereference second: "
<< *second << endl;
// DOES NOT COMPILE
// the iterator is not a type-star:
// int *subarray = second;
```

14.1.2 Simple iterating, the hard way

Auto iterators

```c++
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:

```c++
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 iterable
(vector, map, your own classes!)

Iterating backward Reverse iteration can not be done with range-based syntax.
Use general syntax with reverse iterator: rbegin, rend.

14.1.3 Forming sub-arrays

Iterators can be used to construct a vector. This can for instance be used to create a subvector:
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, with `copy`, erasing with `erase`, and inserting with `insert`.

**Copy range** Copy a range at an iterator:

**Code:**

```
vector<int> counts{1,2,3,4};
vector<int> copied(5);
copy( counts.begin(),counts.end(),
copied.begin()+1 );
```

**Output**

```
[iter] copy:
  1..4
  0..3
```

(No bound checking, so be careful!)

**Erase at/between iterators** Erase from start to before-end:

```
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); second++;
auto before = vec.end(); before--;
// vector<int> sub(second,before);
```

**Code:**
```
vector<int> counts {1, 2, 3, 4, 5, 6};
vector<int>::iterator second = counts.begin() + 1;
auto fourth = second + 2; // easier than 'iter'
counts.erase (second, fourth);
```

```
Output
[iter] erase2:
1, 4
```

(Also single element without end iterator.)

**Insert at iterator** 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);
```

```
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.

**Reconstruct index**

**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 << " we find " << *it << endl;
    it++;
}
```

```
Output
[loop] distance:
At distance 0 we find 1
At distance 1 we find 3
At distance 2 we find 5
At distance 3 we find 7
At distance 4 we find 9
```

**Iterator arithmetic**

```
auto first = myarray.begin();
first += 2;
auto last = myarray.end();
last -= 2;
myarray.erase(first, last);
```
Exercise 14.1. Use the above vector methods to return, given a `std::vector<float>`, the integer index of its maximum element.

14.2 Algorithms

14.2.1 Any/all/each

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`.

- 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.

**For each, very simple example** Apply something to each array element:

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```cpp
define vectors int s(2,3,4,5,7,8,13,14,15); for_each( s.begin(), s.end(),[i](int i) { cout << i << ”n”; }); ``` | `[iter] each: 2 3 4 5 7 8 13 14 15` |

**For any** See if any element satisfies a boolean test:

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```cpp
define vectors int s(2,3,4,5,7,8,13,14,15); bool there was an 8 = any_of( s.begin(), s.end(),[i](int i) return i==8; ); cout << ”There was an 8: ” << boolalpha << there was an 8; ``` | `[iter] each: 2 3 4 5 7 8 13 14 15` |

**For each, with capture** Capture by reference, to update with the array elements.

Code:

```cpp
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

<table>
<thead>
<tr>
<th>iter</th>
<th>each:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Use lambda to find any of

Here is an example using `any_of` to find whether there is any even element in a vector:

Code:

```cpp
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" << endl;
else 
    cout << "none were even" << endl;
```

Output

<table>
<thead>
<tr>
<th>range</th>
<th>anyof:</th>
</tr>
</thead>
<tbody>
<tr>
<td>there was an even</td>
<td></td>
</tr>
</tbody>
</table>

An example of `for_each`:

Code:

```cpp
#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 << endl;
    } );
```

Output

<table>
<thead>
<tr>
<th>stl</th>
<th>printeach:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Exercise 14.2. 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!

See section 14.1.1 for iterators, in particular section 14.2.1 for algorithms with iterators, and chapter 13 for the use of lambda expressions.

14.2.2 Reduction

Numerical reductions can be applied using iterators in `accumulate` in the `numeric` header:
14.2. Algorithms

**Reduction operation** Default is sum reduction:

```cpp
vector<int> v(1,3,5,7);
auto first = v.begin();
auto last = v.end();
auto sum = accumulate(first,last,0);
```

Output

```
[stl] accumulate:
sum: 16
```

Other arithmetic operators are found in functional: `plus`, `minus`, `multiplies`, `divides`, `modulus` (only for integers). The unary `negate` can of course not be used for reductions.

As an example of an explicitly specified reduction operator:

```cpp
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.

**Reduction with supplied operator** Supply multiply operator:

```cpp
vector<int> v(1,3,5,7);
auto first = v.begin();
auto last = v.end();
first++; last--;
auto product = accumulate
  (first,last,2,
   (first,last,2, multiplies<>()));
```

Output

```
[stl] product:
product: 30
```

**Max reduction** 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.

Example: maximum relative deviation from a quantity:

```cpp
std::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 );
  });
```

14.2.3 Sorting

The `algorithm` header also has a function `sort`.

Victor Eijkhout

```
With iterators you can easily apply this to things such as vectors:

```cpp
sort( myvec.begin(), myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```cpp
sort( myvec.begin(), myvec.end(), [] (int i, int j) { return i>j; } );
```

With iterators you can also do things like sorting a part of the vector:

```cpp
vector<int> v{3,1,2,4,5,7,9,11,12,8,10};
cout << "Original vector: " << vector_as_string(v) << endl;
auto v_std(v);
std::sort( v_std.begin(), v_std.begin()+5 );
cout << "Five elements sorts: " << vector_as_string(v_std) << endl;
```

**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,
```

## 14.3 Ranges

The C++20 standard contains a `ranges` header, which generalizes iterable objects into as-it-were streams, that can be connected with pipes.

We need to introduced two new concepts.

A **range** is an iterable 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:

```cpp
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.

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:
```cpp
vector<int> v{1,2,3,4,5,6};
cout << "Original vector: " << vector_as_string(v) << endl;
auto times_two = v | transform([](int i) { return 2*i; });
cout << "Times two: " << vector_as_string(times_two | ranges::to_vector) << endl;
auto over_five = times_two | filter([](int i) { return i>5; });
cout << "Over five: " << vector_as_string(over_five | ranges::to_vector) << endl;
```

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:
```cpp
vector<int> v{1,2,3,4,5,6};
cout << "Original vector: " << vector_as_string(v) << endl;
auto times_two_over_five = v | transform([](int i) { return 2*i; }) | filter([](int i) { return i>5; }) |
( times_two_over_five | ranges::to_vector )
cout << endl;
```

Output

```
[range] ft2:
Original vector: 1, 2, 3, 4, 5, 6,
Times two over five: 6, 8, 10, 12,
```

Exercise 14.3. 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`

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:

```cpp
auto v = std::views::reverse(vec);
auto v = vec | std::views::reverse;
auto v = vec | std::views::reverse | /* more adaptors */;
```

### 14.3.1 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.

```cpp
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 7.4.2. Make sure you study that material first.

Passing a variable to a routine passes the value; in the routine, the variable is local.

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

If you do want to make the change visible in the calling environment, use a reference:

```c
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.

**Reference: change argument** A reference makes the function parameter a synonym of the argument.

```c
void f(int &i) { i += 1; }
int main() {
    int i = 2;
    f(i); // makes it 3
}
```

**Reference: save on copying**

```c
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:

```c
void f(BigDude &thing) { .... ;
```

Prevent changes:

```c
void f(const BigDude &thing) { .... ;
```

15.3 Reference to class members

Here is the naive way of returning a class member:

```c
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:

```c
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`:
15.3. Reference to class members

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:

```cpp
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:
```cpp
class MyClass {
private:
  int data;
public:
  MyClass(int i) { data = i; }
  int operator[](int i) const { return data[i]; }
};
```

Output:
```cpp
[func] rhsref:
ld: warning: directory not found for option '-L'/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/lib
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 9.5.5. 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:
```cpp
class Vector10 {
private:
  int array[10];
public:
  int operator[](int i) { return array[i]; }
};
```
15.5. rvalue references

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];
}
```

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];
}
```

15.5 rvalue references

See the chapter about obscure stuff; section 26.4.3.
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.

Recursive data structures  Naive code:

```cpp
class Node {
private:
    int value;
    Node tail;
    /* ... */
};
```

This does not work: would take infinite memory.

Indirect inclusion: only ‘point’ to the tail:

```cpp
class Node {
private:
    int value;
    PointToNode tail;
    /* ... */
};
```

This chapter will explain C++ smart pointers, and give some uses for them.

16.1  The ‘arrow’ notation

Members from pointer

- If x is object with member y:
  ```cpp
  x.y
  ```
- If xx is pointer to object with member y:
  ```cpp
  xx->y
  ```
- In class methods this is a pointer to the object, so:
16. Pointers

```cpp
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?

Use of a shared pointer  Same as C-pointer syntax:

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output [pointer] pointx:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HasX xobj(5);</code></td>
<td>5</td>
</tr>
<tr>
<td><code>cout &lt;&lt; xobj.value() &lt;&lt; endl;</code></td>
<td>6</td>
</tr>
<tr>
<td><code>xobj.set(6);</code></td>
<td>5</td>
</tr>
<tr>
<td><code>cout &lt;&lt; xobj.value() &lt;&lt; endl;</code></td>
<td>6</td>
</tr>
<tr>
<td><code>auto xptr = make_shared&lt;HasX&gt;(5);</code></td>
<td>5</td>
</tr>
<tr>
<td><code>cout &lt;&lt; xptr-&gt;value() &lt;&lt; endl;</code></td>
<td>6</td>
</tr>
<tr>
<td><code>xptr-&gt;set(6);</code></td>
<td>6</td>
</tr>
<tr>
<td><code>cout &lt;&lt; xptr-&gt;value() &lt;&lt; endl;</code></td>
<td></td>
</tr>
</tbody>
</table>

Creating a pointer  Allocation and pointer in one:

```cpp
shared_ptr<Obj> X =
    make_shared<Obj>( /* constructor args */ );
// or simpler:
auto X = make_shared<Obj>( /* args */ );
```

Simple example  Simple class that stores one number:

```cpp
class HasX {
private:
    double x;
public:
    HasX( double x ) : x(x) {};
    auto get() { return x; };
    void set(double xx) { x = xx; };
};
```

Headers for smart pointers  Using smart pointers requires at the top of your file:

```cpp
#include <memory>
using std::shared_ptr;
```
16.2. Making a shared pointer

```cpp
using std::make_shared;
// not in this lecture:
using std::unique_ptr;
using std::make_unique;
```

What's the point of pointers? Pointers make it possible for two variables to own the same object.

**Code:**
```cpp
auto xptr = make_shared<HasX>(5);
auto yptr = xptr;
cout << xptr->get() << endl;
yptr->set(6);
cout << xptr->get() << endl;
```

**Output**

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Twopoint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Pointers to arrays The constructor syntax is a little involved for vectors:

```cpp
auto x = make_shared<vector<double>>(vector<double>{1.1, 2.2});
```

16.2.1 Pointers and arrays

Linked lists The prototypical example use of pointers is in linked lists. Consider a class `Node` with
- a data value to store, and
- a pointer to another `Node`, or `nullptr` if none.

**Constructor sets the data value:**
```cpp
class Node {
private:
  int datavalue{0};
  shared_ptr<Node> tail_ptr{nullptr};
public:
  Node() {}
  Node(int value) : datavalue(value) {};
  int value() { return datavalue; }
}
```

**Set next / test if there is a next:**
```cpp
bool has_next() { return tail_ptr!=nullptr; }
```

List usage Example use:

**Code:**
```cpp
auto first = make_shared<Node>(23),
    second = make_shared<Node>(45);
first->set_tail(second);
cout << "List length: " << first->list_length() << endl;
first->print();
```

**Output**

```
List <<23,45>> has length 2
```
16. Pointers

**Linked lists and recursion**  Many operations on linked lists can be done recursively:

```cpp
int Node::list_length() {
    if (!has_next()) return 1;
    else return 1 + tail_ptr->list_length();
}
```

**Exercise 16.1.**  Write a method `set_tail` that sets the tail of a node.

```cpp
Node one;
one.set_tail(two); // what is the type of 'two'?
cout << one.list_length() << endl; // prints 2
```

**Exercise 16.2.**  Write a recursive `append` method that appends a node to the end of a list:

```cpp
auto first = make_shared<Node>(23),
    second = make_shared<Node>(45),
    third = make_shared<Node>(32);
first->append(second);
first->append(third);
first->print();
```

**Output**

```
[tree] append:
Append 23 & 45 gives <<23,45>>
Append 32 gives <<23,45,32>>
```

**Exercise 16.3.**  Write a recursive `insert` method that inserts a node in a list, such that the list stays sorted:

```cpp
auto first = make_shared<Node>(23),
    second = make_shared<Node>(45),
    third = make_shared<Node>(32);
first->insert(second);
first->insert(third);
first->print();
```

**Output**

```
[tree] insert:
Insert 45 on 23 gives <<23,45>>
Insert 32 gives <<23,32,45>>
```

Assume that the new node always comes somewhere after the head node.

**Exercise 16.4.**  For a more sophisticated approach to linked lists, do the exercises in section 67.1.2.

**Exercise 16.5.**  If you are doing the prime numbers project (chapter 52) you can now do exercise 52.9.2.

16.2.2  **Smart pointers versus address pointers**

**Pointers don’t go with addresses**

The oldstyle `&y` address pointer can not be made smart:
16.3 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.

```cpp
// 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.

**Illustration** We need a class with constructor and destructor tracing:

```cpp
class thing {
public:
    thing() { cout << ".. calling constructor\n"; };
    ~thing() { cout << ".. calling destructor\n"; };
};
```

**Illustration 1: pointer overwrite** Let’s create a pointer and overwrite it:

```cpp
auto p = shared_ptr<HasY>( &y );
p->y = 3;
cout << "Pointer's y: "
    << p->y << endl;
```

gives:
```
address(56325,0x7fff977cc380) malloc: *** error for object 0x7ffeeb9caf08: pointer being freed was not allocated
```

Smart pointers are much better than old style pointers

```cpp
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

```cpp
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 pointers; section 16.5.4.
16. Pointers

Illustration 2: pointer copy

Code:
```cpp
cout << "set pointer2" << endl;
auto thing_ptr2 = make_shared<thing>();
cout << "set pointer3 by copy" << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2" << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3" << endl;
thing_ptr3 = nullptr;
```

Output
```
[pointer] ptr2:
set pointer2
.. calling constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

Output
```
[pointer] ptr3:
set pointer3 by copy
overwrite pointer3
.. 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.

- The object counts how many pointers there are:
- ‘reference counting’
- A pointed-to object is deallocated if no one points to it.

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.

Getting the underlying pointer

```cpp
X->y;
// is the same as
X.get()->y;
// is the same as
(*X.get()).y;
```
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:

```cpp
class A {}
class B : public A {}
```

Just like you could assign a B object to an A variable:

```cpp
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:

```cpp
auto a_ptr = shared_pointer<A>( make_shared<B>() );
```

Note: this is better than

```cpp
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:

```cpp
auto Y = make_shared<HasY>(5);
cout << Y->y << endl;
Y.get()->y = 6;
cout << (*Y.get()).y << endl;
```
### 16. Pointers

```cpp
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

```cpp
    return this->shared_from_this();
```

if you have defined your node class to inherit from what probably looks like magic:

```cpp
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.

```cpp
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

```cpp
auto sp = shared_ptr<Obj>( new Obj );
```

creates first the object, then the ‘control block’ that counts owners. On the other hand,

```cpp
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

```cpp
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.

---

*Introduction to Scientific Programming*
16.5. Advanced topics

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:

```cpp
define f(int);
define 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` know what it points to, use `std::any`, which is usually smart enough to call destructors when needed.

```cpp
std::any a = 1;
cout << a.type().name() << ": 
    << std::any_cast<int>(a) << endl;
a = 3.14;
cout << a.type().name() << ": 
    << std::any_cast<double>(a) << endl;
a = true;
cout << a.type().name() << ": 
    << std::any_cast<bool>(a) << endl;
try{
a = 1;
cout << std::any_cast<float>(a) << endl;
} catch (const std::bad_any_cast & e) {
cout << e.what() << endl;
}
```

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:
16. Pointers

It is possible to initialize that double:

```cpp
// shared pointer to initialized double
auto array = make_shared<double>(50);
double *ptr = array.get();
cout << ptr[0] << endl;
```

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 7.4.
- 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 67.1.2.
- 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 off: 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.

Memory addresses 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:

```cpp
int i;
cout << "address of i, decimal: " << (long)&i << endl;
cout << "address of i, hex : " << std::hex << &i << endl;
```

Output

```
[pointer] coutpoint:
address of i, decimal:
140732703427524
address of i, hex : 0x7ffee2cbcbc4
```

Same in C Using purely C:
17. C-style pointers and arrays

```c
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 7.4.2. 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.

### Address types
The type of ‘& i’ is `int*`, pronounced ‘int-star’, or more formally: ‘pointer-to-int’.

You can create variables of this type:

```c
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:

```c
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.

### Dereferencing
Using `*addr` ‘dereferences’ the pointer: gives the thing it points to; the value of what is in the memory location.

```c
int i;
int* addr = &i;
    i = 5;
cout << *addr << endl;
    i = 6;
cout << *addr << endl;
```

Output
[pointer] cintpointer:
5
6
17.3 Arrays and pointers

Illustration

```
int x = 6;       "x"
    6
946
int y = x;       "x"    "y"
    6    6
946 958
```

- `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.

**Star stuff**  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 10.8.2 you saw the treatment of static arrays in C++. Examples such as:
C-style pointers and arrays

```c
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] << endl;
```

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:

```c
void array_set_star( double *ar, int idx, double val) {
    ar[idx] = val;
};
/* ... */
array_set_star(array,2,4.2);```

Array and pointer equivalence

Array and memory locations are largely the same:

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>double array[5] = {11,22,33,44,55};</strong></td>
<td></td>
</tr>
<tr>
<td>*<em>double <em>addr_of_second = &amp;array[1];</em></em></td>
<td></td>
</tr>
<tr>
<td>*<em>cout &lt;&lt; <em>addr_of_second &lt;&lt; endl;</em></em></td>
<td></td>
</tr>
<tr>
<td><strong>array[1] = 7.77;</strong></td>
<td></td>
</tr>
<tr>
<td>*<em>cout &lt;&lt; <em>addr_of_second &lt;&lt; endl;</em></em></td>
<td></td>
</tr>
<tr>
<td><strong>Output [pointer] arrayaddr:</strong></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>7.77</td>
<td></td>
</tr>
</tbody>
</table>

Array passing to function

When an array is passed to a function, it behaves as an address:

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>void set_array( double <em>x, int size) {</em></em></td>
<td></td>
</tr>
<tr>
<td><strong>for (int i=0; i&lt;size; i++)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>x[i] = 1.41;</strong></td>
<td></td>
</tr>
<tr>
<td><strong>};</strong></td>
<td></td>
</tr>
<tr>
<td><em><em>/</em> ... <em>/</em></em></td>
<td></td>
</tr>
<tr>
<td><strong>double array[5] = {11,22,33,44,55};</strong></td>
<td></td>
</tr>
<tr>
<td><strong>set_array(array,5);</strong></td>
<td></td>
</tr>
<tr>
<td><strong>cout &lt;&lt; array[0] &lt;&lt; &quot;....&quot; &lt;&lt; array[4] &lt;&lt; endl;</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Output [pointer] arraypass:</strong></td>
<td></td>
</tr>
<tr>
<td>1.41....1.41</td>
<td></td>
</tr>
</tbody>
</table>

Note that these arrays don’t know their size, so you need to pass it.

Dynamic allocation

You can dynamically reserve memory with `new`, which gives a something-star:

```c
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 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

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++ pass by reference C++ style functions that alter their arguments:

void inc(int &i) {
i += 1;
}
int main() {
  int i=1;
  inc(i);
C-style pointers and arrays

```cpp
cout << i << endl;
return 0;
}
```

C-style pass by reference In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```cpp
void inc(int *i) {
    *i += 1;
}

int main() {
    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:

```cpp
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 10.8.2 you learned how to create arrays that are local to a scope:

**Problem with static arrays** Create an array with size depending on something:

```cpp
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:
### Allocation, C vs C++

C allocates in bytes:

```c
double *array;
array = (double*) malloc( 25*sizeof(double) );
```

C++ allocates an array:

```c
double *array;
array = new double[25];
```

Don’t forget:

```c
free(array); // C  
delete array; // C++
```

### Declaration and allocation

Now dynamic allocation:

```c
double *array;
if (something) {
    array = new double[25];
} else {
    array = new double[26];
}
```

Don’t forget:

```c
delete array;
```

### Memory leak 1

```c
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
- ⇒ memory leak.

### Memory leaks

```c
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!
17. C-style pointers and arrays

**De-allocation** Memory allocated with `malloc`/`new` does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```c
free(array);
delete(array);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

**Stop using C!** 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:

**Allocation in C**

```c
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.

**Allocation in a function**

```c
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 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.

```cpp
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:

```cpp
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 memory leaking: there is memory that you have reserved, but you have lost the ability to access it.

In this example:

```cpp
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. C-style pointers and arrays

17.8 Const pointers

A pointer can be constant in two ways:

1. It points to a block of memory, and you cannot 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:

```
int value = 5;
int *ptr = &value;
*ptr += 1;
cout << "value: " << value << endl;
cout << "*ptr: " << *ptr << endl;
ptr += 1;
cout << "random memory: " << *ptr << endl;
```

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.

Constant arguments Function arguments marked `const` can not be altered by the function code. The following segment gives a compilation error:

```cpp
void f(const int i) {
    i++;  // Compilation error
}
```

18.2 Const references

A more sophisticated use of `const` is the `const reference`:

```cpp
void f( const int &i ) { .... }
```

This may look strange. After all, references, and the pass-by-reference mechanism, were introduced in section 7.4 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:
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:

```cpp
class has_int {
private:
    int mine(1);
public:
    has_int(int v) {
        cout << "set: " << v << endl;
        mine = v;
    }
    has_int(has_int &h) {
        auto v = h.mine;
        cout << "copy: " << v << endl;
        mine = v;
    }
    void printme() { cout << "I have: " << mine << endl; }
};
```

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:

```cpp
void f_with_copy(has_int other) {
    cout << "function with copy" << endl;
}
void f_with_ref(const has_int &other) {
    cout << "function with ref" << endl;
}/* ... */
cout << "Calling f with copy..." << endl;
f_with_copy(an_int);
cout << "Calling f with ref..." << endl;
f_with_ref(an_int);
```
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

\[
\text{\texttt{for ( auto } v : \text{some\_vector})}
\]

copies the vector elements to the \textit{v} variable, whereas

\[
\text{\texttt{for ( auto} &v : \text{some\_vector})}
\]

makes a reference. To get the benefits of references (no copy cost) while avoiding the pitfalls (inadvertant changes), you can also use a const-reference here:

\[
\text{\texttt{for ( const auto} &v : \text{some\_vector})}
\]

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 \textit{const}. While this is in no way required, it contributes to a clean programming style:

<table>
<thead>
<tr>
<th>Intention checking</th>
<th>Using \texttt{const} will catch mismatches between the prototype and definition of the method. For instance,</th>
</tr>
</thead>
</table>
| \texttt{class Things {}
|  \texttt{private:}
|  \texttt{int var;}
|  \texttt{public:}
|  \texttt{f(int &ivar, int c) const {}
|  \texttt{  var += c;} // typo: should be \texttt{\textquote{ivar}}
|  \texttt{}}}
| |

Here, the use of \textit{var} was a typo, should have been \textit{ivar}. Since the method is marked \texttt{const}, the compiler will generate an error.

<table>
<thead>
<tr>
<th>No side-effects</th>
<th>It encourages a functional style, in the sense that it makes side-effects impossible:</th>
</tr>
</thead>
</table>
| \texttt{class Things {}
|  \texttt{private:}
|  \texttt{int i;}
|  \texttt{public:}
|  \texttt{int get() const { return i; }}
|  \texttt{int inc() { return i++; }} // side-effect!
|  \texttt{void addto(int &thru) const { thru += i; }}
| }

18.4 Overloading on const
**Const polymorphism**  A const method and its non-const variant are different enough that you can use this for overloading.

**Code:**

```cpp
class has_array {
private:
    vector<float> values;
public:
    has_array(int i, float v) :
        values(vector<float>({i,v})) {};
    auto at(int i) {
        cout << "var at" << endl;
        return values.at(i); }
    const auto at(int i) const {
        cout << "const at" << endl;
        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 i; float v;
    cin >> i; cin >> v;
    has_array fives(i,v);
    cout << fives.sum() << endl;
    fives.at(0) = 2;
    cout << fives.sum() << endl;
}
```

**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`.

```cpp
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() << endl;
    }
};
```
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:

```cpp
auto get_thing_ptr() const {
    return thing_ptr;
}
/* ... */
has_thing container(5);
container.print();
container.get_thing_ptr() =
    make_shared<thing>(6);
container.print();
```

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:

```cpp
// 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();
```

If you want to prevent the pointed object from being changed, you can declare the pointer as a `shared_ptr<const thing>`:

```cpp
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 crnt() const {
    cout << const_thing->value() << endl;
}
/* ... */
has_thing container(1,2);
// Error: does not compile
container.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:

```cpp
int *p;
const int *p;
int const * p; // this is the same
int * const p; // as this
```
18. Const

For the interpretation, it is often said to read the declaration ‘from right to left’. So:

```c
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:

```c
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,

```c
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:

```c
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,

```c
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:

```c
// 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:

```c
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; };
};
```

Introduction to Scientific Programming
18.7. Compile-time constants

Attempt at caching the computation:

```cpp
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?

```cpp
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 { 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`.)

```cpp
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 { 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 cannot be changed:

```cpp
const int i=5;
// DOES NOT COMPILE:
i += 2;
```

**constexpr if** The combination `if constexpr` is useful with templates:

```cpp
template <typename T>
auto get_value(T t) {
  if constexpr (std::is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

**Constant functions** To declare a function to be constant, use `constexpr`. The standard example is:

```cpp
constexpr double pi() {
  return 4.0 * atan(1.0); }
```

but also

```cpp
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

Prototypes 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.

![Diagram](image)

**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 tying it all together. We will work towards this in a number of steps.

### 19.2 Prototypes for functions

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

```c
int tester(float);
```

#### Prototypes and forward declarations, 1

A first use of prototypes is forward declaration.

Some people like defining functions after the main:

```c
int f(int);
int main() {
  f(5);
}
int f(int i) {
  return i;
}
```

versus before:

```c
int f(int i) {
  return i;
}
int main() {
  f(5);
}
```

#### Prototypes and forward declarations, 2

You also need forward declaration for mutually recursive functions:

```c
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

Prototypes are useful if you spread your program over multiple files. You would put your functions in one file and the main program in another.

```c
// file: def.cxx
int tester(float x) {
  ....
}
```

```c
// 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:
19.2. Prototypes for functions

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

Compiling and linking  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.

Dealing with multiple files  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 prototype every time you need the function is to have a header file:

Prototypes and header files
19. Prototypes and header files

Header file contains only prototype:

```c
// file: def.h
int tester(float);
```

The header file gets included both in the definitions file and the main program:

```c
// file: def.hxx
#include "def.h"
int tester(float x) {
    ....
}
```

```c
// file: main.hxx
#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 prototype 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

```c
// header.h
int somefunction(int);
```

and

```c
#include "header.h"
int somefunction(float x) { .... }
```

19.2.3 C and C++ headers

You have seen the following syntaxes for including header files:

```c
#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 Prototypes for class methods

### Class prototypes

**Header file:**

```cpp
class something {
    private:
        int i;
    public:
        double dosomething( int i, char c );
};
```

**Implementation file:**

```cpp
double something::dosomething( int i, char c ) {
    // do something with i, c
};
```

### Data members in proto

**Header file:**

```cpp
class something {
    private:
        int localvar;
    public:
        double somedo(vector);
};
```

**Implementation file:**

```cpp
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 prototype.

- 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

The use of templates (see chapter 22) often make 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,
19. Prototypes and header files

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:

\[
\text{template } \text{typename } T \text{ class foo}<T> \{ \\
\};
\]

and it will only be used (instantiated) with float and double, then adding the following lines after the class definition makes the file separately compilable:

\[
\text{template class foo}<\text{float}>; \\
\text{template class foo}<\text{double}>;
\]

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:

\[
\text{int processnumber; } \\
\text{void f() \{ } \\
\text{... processnumber ... } \\
\}; \\
\text{int main() \{ } \\
\text{processnumber = // some system call } \\
\};
\]

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:

\[
\text{extern int processnumber;}
\]

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:
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.) This largely dispenses with the need for header files included through C Preprocessor (CPP).
19. Prototypes and header files
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

```cpp
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 having

```cpp
using namespace std;
```

somewhere high up in your file, and write

```cpp
vector<int> bunch_of_ints;
```

in the rest of the file.

More safe:

```cpp
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.

### Defining a namespace

You can make your own namespace by writing

```cpp
namespace a_namespace {
    // definitions
    class an_object {
    };
}
```

so that you can write
20. Namespaces

### Namespace usage

```cpp
namespace a_namespace::an_object myobject();
```

or

```cpp
using namespace a_namespace;
an_object myobject();
```

or

```cpp
using a_namespace::an_object;
an_object myobject();
```

or

```cpp
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:

```cpp
#include "geolib.h"
using namespace geometry;
```

The header would contain the normal function and class headers, but now inside a named namespace:

```cpp
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:

```cpp
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 ) {
```
20.2. Best practices

In this course we advocated pulling in functions explicitly:

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

It is also possible to use

```cpp
#include <iostream>
using namespace std;
```

The problem with this is that it may pull in unwanted functions. For instance:

**Why not ‘using namespace std’?**

This compiles, but should not:  

```cpp
#include <iostream>
using namespace std;

def swop(int i, int j) {};

int main() {
  int i=1, j=2;
  swap(i, j);
  cout << i << endl;
  return 0;
}
```

This gives an error:

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

def swop(int i, int j) {};

int main() {
  int i=1, j=2;
  swap(i, j);
  cout << i << endl;
  return 0;
}
```

Even if you use `using namespace std`, 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.
20. Namespaces
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. Preprocessor

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 \texttt{-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 \texttt{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 \texttt{makefiles} and the \texttt{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.

```cpp
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 \texttt{#define} to define a preprocessor macro:

```cpp
#define N 100000  
void dosomething() {  
  for (int i=0; i<N; i++) .... 
}
```

Introduction to Scientific Programming
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 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 Parametrized macros

Instead of simple text substitution, you can have *parametrized 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:
21. Preprocessor

```c
#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:

```c
#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:

```c
#if 0
  bunch of code that needs to be disabled
#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,

```c
#define N
#include N 100
#endif
```

Why would a macro already be defined? Well you can do that on the compile line:

```c
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` (which, however, is not a language standard) that has the same effect:

// this is foo.h
#pragma once

// the things you want to include only once

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
21. Preprocessor
Chapter 22

Templates

Sometimes you want a function or a class based on more than one different datatypes. For instance, in chapter 10 you saw how you could create an array of ints as `vector<int>` and of doubles as `vector<double>`. Here you will learn the mechanism for that.

**Templated type name** 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

**Example: function** 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:
22. Templates

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:

**Templated vector** The STL contains in effect

```cpp
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
}
```

22.2.1 Specific implementation

```cpp
template<typename T>
void f(T);

template<>  
void f(char c) { /* code with c */ };  

template<>  
void f(double d) { /* code with d */ };  
```

22.3 Example

Let’s start with a simple class for polynomials:

```cpp
class polynomial {
private:
    vector<double> coefficients;
public:
    polynomial(vector<double> c) : coefficients(c) {};
```
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:

$$5x^2 + 4x + 3 \equiv ((5) \cdot x + 4) \cdot x + 3$$

This class is used as:

```cpp
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";
}
```

You may know from mathematics courses that you can also have polynomials based on square matrices. In that case, all matrix 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

```cpp
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 define the `polynomial` class as templated over the scalar type:

```cpp
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. For scalars:
22. Templates

```cpp
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:

```cpp
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";
}
```

22.4 Concepts

Coming in the C++20 standard.

Templates can be too generic. For instance, one could write a templated \( \text{gcd} \) function

```cpp
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:

```cpp
template<typename T>
concept bool Integral() {
    return std::is_integral<T>::value;
}
```

used as:

```cpp
template <typename T>
requires Integral<T>()
T gcd( T a, T b ) { /* ... */ }
```

or

```cpp
template <Integral T>
T gcd( T a, T b ) { /* ... */ }
```

Abbreviated function templates:

```cpp
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: behaviour 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:

```cpp
vector<float> a(10);
for (int i=0; i<=10; i++)
a.at(i) = x; // runtime error
```

or undefined behaviour:

```cpp
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:

```cpp
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. Error handling

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:

**Using assertions** Check on valid input parameters:

```cpp
#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:

```cpp
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:

```cpp
#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 6.12.

**Example**

```cpp
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`.  

242 Introduction to Scientific Programming
23.2. Mechanisms to support error handling and debugging

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.

### Exception structure

**Code with problem:**

```cpp
if ( /* some problem */ )
    throw(5);
/* or: throw("error"); */
```

**try**

```cpp
try {
    /* code that can go wrong */
    catch (...){ // literally three dots!
    /* code to deal with the problem */
}
```

**catch**

```cpp
/* 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:**

```cpp
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:**

```cpp
vector<float> x(5);
for (int i=0; i<10; i++) {
    try {
        x.at(i) = 3.14;
    } catch (...) { 
        cout << "Exception indexing at: " << i << endl;
        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. Error handling

23.2.2.3 Throw your own exceptions

**Exception throwing**  Throwing an exception is one way of signalling an error or unexpected behaviour:

```cpp
void do_something() {
    if ( oops )
        throw(5);
}
```

**Catching an exception**  It now becomes possible to detect this unexpected behaviour by catching the exception:

```cpp
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 52.12.

You can throw integers to indicate an error code, a string with an actual error message. You could even make an error class:

**Exception classes**

```cpp
class MyError {
public:
    int error_no; string error_msg;
MyError( int i, string msg )
    : error_no(i), error_msg(msg) {};
}

throw( MyError(27, "oops");
```

```cpp
try {
    // something
} catch ( MyError & m ) {
    cout << "My error with code=" << m.error_no
        << " msg=" << m.error_msg << endl;
}
```

You can use exception inheritance!

**Multiple catches**  You can multiple `catch` statements to catch different types of errors:

```cpp
try {
    // something
} catch ( int i ) {
    // handle int exception
} catch ( std::string c ) {
    // handle string exception
}
```
23.2. Mechanisms to support error handling and debugging

**Catch any exception** Catch exceptions without specifying the type:

```cpp
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`

```cpp
#include <cfenv>
using std::isnan;
```

and again throw an exception.

Exceptions in constructors A function try block will catch exceptions, including in member initializer lists of constructors.

```cpp
f::f( int i )
try : fbase(i) {
    // constructor body
} catch ( ... ) { // handle exception
}
```

More about exceptions

- Functions can define what exceptions they throw:

```cpp
void func() throw( MyError, std::string );
void func() 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`
23. Error handling

 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, as well as the use of uninitial-ized 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

```cpp
#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.

```cpp
#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`.

Imaginary unit number i through literals `i`, `if` (float), `il` (long):

```cpp
using namespace std::complex_literals;
std::complex<double> c = 1.0 + il;
```

Support:

```cpp
std::complex<T> conj(const std::complex<T>& z);
std::complex<T> exp(const std::complex<T>& z);
```
24. Standard Template Library

24.2 Containers

Vectors (section 10.3) and strings (chapter 11) are special cases of a STL container. Methods such as push_back and insert apply to all 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:

```cpp
#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:

```cpp
#include <map>
using std::make_pair;
ages.insert(make_pair("Alice",29));
ages["Bob"] = 32;
```

You can range over a map:

```cpp
for ( auto person : ages )
    cout << person.first << " has age " << person.second << endl;
```

A more elegant solution uses structured bindings (section 24.4):

```cpp
for ( auto [person,age] : ages )
    cout << person " has age " << age " endl;
```

Searching for a key gives either the iterator of the key/value pair, or the end iterator if not found:

```cpp
for ( auto k : {4,5} ) {
    auto wherek = intcount.find(k);
    if (wherek==intcount.end())
        cout << "could not find key" << k << endl;
    else {
        auto [kk,vk] = *wherek;
        cout << "found key: " << kk << " has value " << vk << endl;
    }
}
```

Exercise 24.1. If you’re doing the prime number project, you can now do the exercises in section 52.7.2.
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:

```
vector<string> names ("Victor", "aDam", "DoD")
auto cap = regex("[A-Z][a-z]+")
for (auto n : names) {
    auto match = regex_match(n, cap);
    if (match) cout << ": yes"
    else cout << ": no"
    cout << endl;
}
```

(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 preceeding the match; with `smatch::prefix().size()` you get the number of characters preceeding the match, that is, the location of the match.

```
[regexp] search:
Found <<the>>
Found <<o[^o]+o>> at 12 as <<own fo>> preceeded by <<The quick br>>
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
24. Standard Template Library

24.3.1 Regular expression syntax


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 bindings

Remember how in section 7.4.2 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 `std::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`.)

### C++11 style tuples

```cpp
#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:

```cpp
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:

```cpp
tuple<bool, float>
maybe_root2(float x) {
    if (x<0)
```
24.5. Union-like stuff: tuples, optionals, variants

```cpp
return (false,-1);
else
    return (true,sqrt(x));
```

Catching a returned tuple  The calling code is particularly elegant:

```cpp
code:
auto [succeed,y] = maybe_root2(x);
if (succeed)
    cout << "Root of " << x
    << " is " << y << endl;
else
    cout << "Sorry, " << x
    << " is negative" << endl;
```

Output
[stl] tuple:
Root of 2 is 1.41421
Sorry, -2 is negative

This is known as structured binding.

An interesting use of structured bindings is iterating over a map (section 24.2.1):

```cpp
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:

```cpp
bool RootOrError(float &x) {
if (x<0)
    return false;
else
    x = sqrt(x);
return true;
}
```

```cpp
/* ... */
for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
        cout << "Root is " << x << endl;
else
    cout << "could not take root of " << x << endl;
```

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:

```cpp
tuple<bool,float> RootAndValid(float x) {
    if (x<0)
        return (false,x);
```

Victor Eijkhout
24. Standard Template Library

```cpp
else
    return {true, sqrt(x)};
};
/* ... */
for ( auto x : {2.f, -2.f} )
    if ( auto [ok, root] = RootAndValid(x) ; ok )
        cout << "Root is " << root << endl;
    else
        cout << "could not take root of " << x << endl;
```

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`.

```cpp
#include <optional>
using std::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`.

```cpp
optional<float> f {
    if (something) return 3.14;
    else return {};
}
```

- You can test whether the optional quantity has a quantity with the method `has_value`, in which case you can extract the quantity with `value`:

```cpp
auto maybe_x = f();
if (f.has_value)
    // do something with f.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 52.15.

**Exercise 24.3.** The eight queens problem (chapter 55) can be elegantly solved using `std::optional`. See also section 55.3 for a Test-Driven Development (TDD) approach.

24.5.3 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` does not suffer from these limitations. The function `get_if` can retrieve a value by type.

Let’s start with a variant of int, double, string:

```cpp
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:

```cpp
union_ids = 3.5;
switch ( union_ids.index() ) {
    case 1:
        cout << "Double case: " << std::get<double>(union_ids) << endl;
}
```

Getting the wrong variant leads to a `bad_variant_access` exception:

```cpp
union_ids = 17;
try {
    cout << "Using option " << union_ids.index() << ": " << get<int>(union_ids) << endl;
} catch (bad_variant_access b) {
    cout << "Exception getting as double while index=" << union_ids.index() << endl;
}
```

It is safer to use `get_if` which gives a pointer if successful, and false if not:

```cpp
union_ids = "Hello world";
if ( auto union_int = get_if<int>(&union_ids) ; union_int )
    cout << "Int: " << *union_int << endl;
else if ( auto union_string = get_if<string>(&union_ids) ; union_string )
    cout << "String: " << *union_string << endl;
```

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:**

```cpp
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 [a,b,c] = coefficients;
    auto result = compute_roots(coefficients);
```

**Output**

```plaintext
[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`.

### 24.5.3.1 The same function on all variants

Suppose you have a variant of some classes, which all support an identically prototyped method:

```cpp
class x_type {
    public: r_type method() { ... };
};
```

Victor Eijkhout 253
24. Standard Template Library

```cpp
class y_type {
public:  // y_type method()
    ...
};
```

It is not directly possible to call this method on a variant:

```cpp
variant< x_type, y_type> xy;
// WRONG xy.method();
```

For a specific example:

```cpp
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 `std::visit`, coming from the `variant` header. This is used to apply an object (defined below) to the variant object:

```cpp
Code:
```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:

```cpp
class sizer {
public:
    int operator()( double_object d ) {
        return static_cast<int>( d.value() );
    }
    int operator()( string_object s ) {
        return s.value().size();
    }
};
```

24.5.4 Any

While `variant` can be any of a number of prespecified 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`:

```cpp
std::any a(12);
std::any_cast<int>(a); // succeeds
std::any_cast<string>(a); // fails
```
24.6 Limits

There used to be a header file *limits.h* that contained macros such as `MAX_INT`. While this is still available, the STL offers a better solution in the `numeric_limits` header.

**Templated functions for limits**  Use header file *limits.h*:

```cpp
#include <limits>
using std::numeric_limits;

cout << numeric_limits<long>::max();
```

**Limits of floating point values**

- 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:

```cpp
cout << "Single lowest "
    << numeric_limits<float>::lowest()
    << " and epsilon "
    << numeric_limits<float>::epsilon()
    << endl;
cout << "Double lowest "
    << numeric_limits<double>::lowest()
    << " and epsilon "
    << numeric_limits<double>::epsilon()
    << endl;
```

**Output**

```
[stl] eps:
Single lowest -3.40282e+38 and epsilon 1.19209e-07
Double lowest -1.79769e+308 and epsilon 2.22045e-16
```

**Some limit values**
Exercise 24.5. Write a program to discover what the maximal \( n \) is so that \( n! \), that is, \( n \)-factorial, can be represented in an \texttt{int}, \texttt{long}, or \texttt{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.

There are tests for detecting whether a number is \texttt{Inf} or \texttt{NaN}. However, using these may slow a computation down.

Detection of \texttt{Inf} and \texttt{NaN} The functions \texttt{isinf} and \texttt{isnan} are defined for the floating point types (\texttt{float}, \texttt{double}, \texttt{long double}), returning a \texttt{bool}.

```c
#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
```

24.7 Common numbers

```c
#include <numbers>
static constexpr float pi = std::numbers::pi;
```

24.8 Random numbers

The STL has a \textit{random number generator} that is more general and more flexible than the C version (section 24.8.1), 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:

```cpp
std::default_random_engine generator;
```

This generator will start at the same value everytime. You can seed it:

```cpp
std::random_device r;
std::default_random_engine generator( r() );
```

Next, you need to declare the distribution. For instance, a uniform distribution between given bounds:

```cpp
std::uniform_real_distribution<float> distribution(0.,1.);
```

A roll of the dice would result from:

```cpp
std::uniform_int_distribution<int> distribution(1,6);
```

---

### Random floats

```cpp
// 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) << endl;
```

### Dice throw

```cpp
// 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
```

### Poisson distribution

Another distribution is the Poisson distribution:

```cpp
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```
24.8.1 C random function

There is an easy (but not terribly great) random number generator that works the same as in C.

```cpp
#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.
- 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

  ```cpp
  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:

```c
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.

### 24.9 Time

**Header**

```c
#include <chrono>
```

Several clocks. Here we use

```c
using myclock = std::chrono::high_resolution_clock;
```

**Resolution:**

```c
auto num = myclock::period::num,
    den = myclock::period::den;
auto tick = static_cast<double>(num)/static_cast<double>(den);
```

**Timing:**

```c
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:

```c
auto deadline = myclock.now() + std::chrono::seconds(10);
```

### 24.10 File system

As of the C++17 standard, there is a file system header, `filesystem`, which includes things like a directory walker.

```c
#include <filesystem>
```

### 24.11 Regular expressions

Victor Eijkhout
24. Standard Template Library

Example

Code:

```cpp
vector<string> names { "Victor", "aDam", "DoD" };  
auto cap = regex("^[a-z][a-z]+$");  
for ( auto n : names )  
    auto match = regex_match( n, cap );  
    cout << n;  
    if (match) cout << ": yes";  
    else cout << ": no" ;  
    cout << endl;  
```

Output

```cpp
[regexp] regexp: 
Looks like a name: libc++abi.dylib: terminating 
with uncaught exception of type std::__1::regex_error: 
Unknown error type 
made[2]: *** [run_regexp] Abort 
trap: 6
```

24.12 Enum classes

The C-style `enum` keyword introduced global names, so

```cpp
enum colors { red,yellow,green };  
cout << red << "," << yellow << "," << green << endl;  
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:

```cpp
enum class colors { red,yellow,green };  
cout << static_cast<int>( colors::red ) << "," << static_cast<int>( colors::yellow ) << "," << static_cast<int>( colors::green ) << endl;
```

Even if such a class inherits from an integral type, you still need to cast it occasionally:

```cpp
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 ) << endl;
```

If you only want a namespace-d enum:

```cpp
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 );
```

Introduction to Scientific Programming
Chapter 25

Concurrency

Concurrency is a difficult subject. For a good introduction watch https://www.youtube.com/watch?v=F6Ipn7gC0sY from which a lot of this is taken.

25.1 Thread creation

Use header

```cpp
#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.

```cpp
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 1.00016 sec

25.2 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:
25. Concurrency

```cpp
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 << endl;
```

Output

[thread] race:
Three runs of <<race>>;
printing first lines only:
-----
Main: 1
Thread: 56
Final value: 376753
Runtime: 1.00547 sec
-----
Main: 1
Thread: 11
Final value: 496720
Runtime: 1.00309 sec
-----
Main: 1
Thread: 57
Final value: 480175
Runtime: 1.00361 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`.

```cpp
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 << endl;
```

Output

[thread] atomic:
Three runs of <<atomic>>;
printing first lines only:
-----
Main: 1
Thread: 13
Final value: 423433
Runtime: 1.01244 sec
-----
Main: 1
Thread: 2
Final value: 401307
Runtime: 1.00498 sec
-----
Main: 1
Thread: 58
Final value: 444905
Runtime: 1.01652 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 26

Obscure stuff

26.1 Auto

This is not actually obscure, but it intersects many other topics, so we put it here for now.

26.1.1 Declarations

Sometimes the type of a variable is obvious:

```cpp
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:

```cpp
auto myvar = new std::vector< std::shared_ptr< myclass >>
    ( 20, new myclass(1.3) );
```

Type deduction in functions Return type can be deduced in C++17:

```cpp
auto equal( int i, int j ) { return i==j; }
```

Type deduction in methods Return type of methods can be deduced in C++17:

```cpp
class A {
private:  float data;
public:
    A( float i ) : data(i) {};
    auto &access() {
        return data;
    }
    void print() {
        cout << "data: " << data << endl;
    }
};
```
26. Obscure stuff

### Auto and references, 1
auto discards references and such:

**Code:**
```cpp
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

**Output**

Output

[auto] plainget:
```
data: 5.7
```

### Auto and references, 2
Combine auto and references:

**Code:**
```cpp
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

**Output**

Output

[auto] refget:
```
data: 6.7
```

### Auto and references, 3
For good measure:

```cpp
A my_a(5.7);
const auto &get_data = my_a.access();
get_data += 1; // WRONG does not compile
my_a.print();
```

26.1.2 Auto and function definitions

The return type of a function can be given with a trailing return type definition:

```cpp
auto f(int i) -> double { /* stuff */};
```

This notation is more common for lambdas, chapter 13.

26.1.3 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

```cpp
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:

```cpp
try {
    auto v = some_object.get_stuff();
} catch (...) {};
f(v);
```

because the try block is a scope. It also doesn’t work to write

```cpp
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);
```

### 26.2 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 iterable that has a number of conditions satisfied.

The class needs to have:

- a method `begin` with prototype
  
  ```
  iterableClass iterableClass::begin()
  ```
  
  That gives an object in the initial state, which we will call the ‘iterator object’; likewise
- a method `end`
  
  ```
  iterableClass iterableClass::end()
  ```
  
  that gives an object in the final state; furthermore you need
- an increment operator
  
  ```
  void iterableClass::operator++()
  ```
  
  that advances the iterator object to the next state;
- a test
  
  ```
  bool iterableClass::operator!=(const iterableClass&)
  ```
  
  to determine whether the iteration can continue; finally
- a dereference operator
  
  ```
  iterableClass::operator*()
  ```
  
  that takes the iterator object and returns its state.

### Simple illustration

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 {

```
26. Obscure stuff

```cpp
// basic data
private:
    int first, last;
public:
    bag(int first, int last) : first(first), last(last) {};
```

**Internal state** 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:

```cpp
private:
    int seek(0);
```

**Initial/final state** The begin method gives a bag with the seek parameter initialized:

```cpp
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.

**Termination test** The termination test method is called on the iterator, comparing it to the end object:

```cpp
bool operator!=(const bag &test) const {
    return seek <= test.last;
}
```

**Dereference** Finally, we need the increment method and the dereference. Both access the seek member:

```cpp
void operator++() { seek++; }
int operator*() { return seek; }
```

**Use case**

We can iterate over our own class:
26.2. Iterating over classes

Code:

```cpp
bag digits(0,9);

bool find3{false};
for ( auto seek : digits )
    find3 = find3 | (seek==3);
cout << "found 3: " << boolalpha << find3 << endl;

bool find15{false};
for ( auto seek : digits )
    find15 = find15 | (seek==15);
cout << "found 15: " << boolalpha << find15 << endl;
```

(for this particular case, use `std::any_of`)

With algorithms

Code:

```cpp
bool find8 = any_of(digits.begin(),digits.end(),
    [=] (int i) { return i==8; } );
cout << "found 8: " << boolalpha << find8 << endl;
```

Output

```
[loop] bagfind:
found 3: true
found 15: false
```

If we add a method `has` to the class:

```cpp
bool has(int tst) {
    for (auto seek : *this )
        if (seek==tst) return true;
    return false;
}
```

we can call this:

```cpp
cout << "f3: " << digits.has(3) << endl;
cout << "f15: " << digits.has(15) << endl;
```

Of course, we could have written this function without the range-based iteration, but this implementation is particularly elegant.

**Exercise 26.1.** You can now do exercise 52.20, 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 26.2.** Find a way to make `has` a `const` method.
26. Obscure stuff

26.3  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:

```c
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’:

```c
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:

```c
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 15 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.


26.3.1  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`.

```c
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
cout << "overflow: " << overflow << endl;
size_t bignumber = static_cast<size_t>(hundredk)*hundredk;
cout << "bignumber: " << bignumber << endl;
```
However, if the conversion is possible, the result may still not be ‘correct’.

```
Code:
long int hundredg = 100000000000;
cout << "long number: " << hundredg << endl;
int overflow;
overflow = static_cast<int>(hundredg);
cout << "assigned to int: " << overflow << endl;
```

```
Output
[cast] int long:
```

There are no runtime tests on static casting.
Static casts are a good way of casting back void pointers to what they were originally.

### 26.3.2 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!" << endl;
    };
};
class Erived : public Base {
public:
    virtual void print() {
        cout << "Construct erived!" << endl;
    };
};
```

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" << endl;
    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:

```cpp
Base *object = new Derived();
f(object);
Base *nobject = new Derived();
f(nobject);
```

On the other hand, a `static_cast` would not do the job:

```cpp
void g(Base *obj) {
    Derived *der = static_cast<Derived*>(obj);
    der->print();
};
/* ... */
Base *object = new Derived();
g(object);
Base *nobject = new Derived();
g(nobject);
```

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.

### 26.3.3 Const cast

With `const_cast` you can add or remove `const` from a variable. This is the only cast that can do this.

### 26.3.4 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:

```cpp
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.

### 26.3.5 A word about void pointers

A traditional use for casts in C was the treatment of `void pointers`. 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.
26.4. lvalue vs rvalue

Here you can declare your own monitor routine that will be called internally: the library makes a callback to your code. Since the library cannot 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):

```c
KSPSetMonitor( ksp,
    mycontext) (KSP k, int ,PetscReal r) -> int {
    my_monitor_function(k,r,mycontext); } );
```

26.4. lvalue vs rvalue

The terms ‘lvalue’ and ‘rvalue’ sometimes appear in compiler error messages.

```c
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:

```c
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.

### 26.4.1 Conversion

Most lvalues can quickly be converted to rvalues:

```cpp
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:

```cpp
int i;
int *a = &i;
&i = 5; // wrong
```

### 26.4.2 References

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```cpp
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

```cpp
const std::string &s = std::string();
```

works, since s can not be modified any further.

### 26.4.3 Rvalue references

A new feature of C++ is intended to minimize the amount of data copying through move semantics.

Consider a copy assignment operator
26.5 Move semantics

```cpp
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

```cpp
BigThing 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:

```cpp
BigThing& operator=( BigThing &&other ) {
    swap( /* other into me */ );
    return *this;
}
```

26.5 Move semantics

With an overloaded operator, such as addition, on matrices (or any other big object):

```cpp
Matrix operator+(Matrix &a, Matrix &b);
```

the actual addition will involve a copy:

```cpp
Matrix c = a+b;
```

Use a move constructor:

```cpp
class Matrix {
private:
    Representation rep;
public:
    Matrix(Matrix &&a) {
        rep = a.rep;
        a.rep = {};
    }
};
```

26.6 Graphics

C++ has no built-in graphics facilities, so you have to use external libraries such as OpenFrameworks, https://openframeworks.cc.
26. Obscure stuff

26.7 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 << endl

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.

26.7.1 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.

26.7.2 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 14.1.1) is largely obviated.
- Lambdas. See chapter 13.
- Variadic templates.
- Unique pointer.
  ``` c++
  unique_ptr<int> iptr( new int(5) );
  ```
  This fixes problems with auto\_ptr.
- constexpr
  ``` c++
  constexpr int get_value() {
    return 5*3;
  }
  ```

26.7.3 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:
  ``` c++
  auto f() {
    SomeType something;
    return something;
  }
  ```
26.7. Standards timeline

• Generic lambdas (section 13.4.2)

\[
\texttt{const auto count} = \texttt{std::count(begin(vec),end(vec), [] ( const auto i ) { return i<3; })};
\]

Also more sophisticated capture expressions.

• Unique pointer

\[
\texttt{auto iptr( make_unique<int>>(5) )};
\]

This makes \texttt{new} and \texttt{delete} completely superfluous.

• constexpr

\[
\texttt{constexpr int get_value()} {
\texttt{int val = 5;}
\texttt{int val2 = 3;}
\texttt{return val*val2}
\}
\]

26.7.4 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.

26.7.5 C++20

• \texttt{modules}: these offer a better interface specification than using \texttt{header files}.
• \texttt{coroutines}, another form of parallelism.
• \texttt{concepts} including in the standard library via ranges; section 22.4.
• \texttt{spaceship operator} including in the standard library
• broad use of normal C++ for direct compile-time programming, without resorting to template metaprogramming (see last trip reports)
• \texttt{ranges}
• \texttt{calendars} and \texttt{time zones}
• \texttt{text formatting}
• \texttt{span}. See section 10.8.5.
• \texttt{numbers}. Section 24.7.
• Safe integer/unsigned comparison; section 4.6.2.

Here is a summary with examples: https://oleksandrkl.github.io/2021/04/02/cpp-20-overview.html.
26. Obscure stuff
Chapter 27

Graphics

The C++ language and standard library do not have graphics components. However, the following projects exist.

https://www.sfml-dev.org/
Chapter 28

C++ for C programmers

28.1 I/O

There is little employ for \texttt{printf} and \texttt{scanf}. Use \texttt{cout} (and \texttt{cerr}) and \texttt{cin} instead. There is also the \texttt{fmtlib} library.

Chapter 12.

28.2 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 \texttt{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 \texttt{span}; section 10.8.5.

28.2.1 Vectors from C arrays

Suppose you have to interface to a C code that uses \texttt{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 \texttt{range constructor}:

\[
\texttt{vector<double> x( pointer_to_first, pointer_after_last );}
\]

Such vectors can still be used dynamically, but this may give a memory leak and other possibly unwanted behavior:
28. C++ for C programmers

```cpp
float *x;
x = (float*)malloc(length*sizeof(float));
/* ... */
vector<float> xvector(x, x+length);
cout << "xvector has size: " << xvector.size()
    << endl;
    xvector.push_back(5);
cout << "Push back was successful" << endl;
cout << "pushed element: "
    << xvector.at(length) << endl;
cout << "original array: "
    << x[length] << endl;
```

**Output**

```text
[array] cvector:
xvector has size: 53
Push back was successful
pushed element: 5
original array: 0
```

### 28.3 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 10.8.3. (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

```cpp
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:

```cpp
vector<double> x(10);
```

declares the variable `x`, allocates the dynamic storage, and initializes it.

### 28.4 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.

### 28.5 Pointers

Many of the uses for C pointers, 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 7.4.
• 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 pointers`, and they can still be used for ‘non-owning’ occurrences of pointers.

28.5.1 Parameter passing

No longer by address: now true references! Section 7.4.

28.6 Objects

Objects are structures with functions attached to them. Chapter 9.

28.7 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter 20.

28.8 Templates

If you find yourself writing the same function for a number of types, you’ll love templates. Chapter 22.

28.9 Obscure stuff

28.9.1 Lambda

Function expressions. Chapter 13.

28.9.2 Const

Functions and arguments can be declared const. This helps the compiler. Section 18.1.

28.9.3 Lvalue and rvalue

Section 26.4.
28. C++ for C programmers
Chapter 29

C++ review questions

29.1 Arithmetic

1. Given
   ```
   int n;
   ```
   write code that uses elementary mathematical operators to compute $n^3$. 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;
   ```

29.2 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.

29.3 Functions

**Exercise 29.1.** The following code snippet computes in a loop the recurrence

\[ v_{i+1} = av_{i} + b, \quad v_{0} \text{ given.} \]

Write a recursive function

\[
\textbf{float} \quad v = \textbf{value}_n(n,a,b,v0);
\]
29. C++ review questions

that computes the value \( v_n \) for \( n \geq 0 \).

29.4 Vectors

**Exercise 29.2.** 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.

```cpp
#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
}
```

29.5 Vectors

**Exercise 29.3.** Take another look at exercise 29.1. 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?

29.6 Objects

**Exercise 29.4.** 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

```cpp
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 29.5.** 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.

```cpp
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 );
}
```
29. C++ review questions
PART III

FORTRAN
Chapter 30

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 idiosyncracies 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 our book, we will teach you 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 our exposition of Fortran can stand on its own, we will in places point out explicitly differences with C++.

For secondary reading, this is a good course on modern Fortran: http://www.pcc.qub.ac.uk/tec/courses/f77tof90/stu-notes/f90studentMIF_1.html

30.1 Source format

Fortran started in the era when programs were stored on punch cards. 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.
30. Basics of Fortran

30.2 Compiling Fortran

For Fortran programs, the compiler is gfortran for the GNU compiler, and ifort for Intel.

The minimal Fortran program is:

```
Program SomeProgram
   ! stuff goes here
End Program SomeProgram
```

Exercise 30.1. Add the line

```
print *, "Hello world!"
```

to the empty program, and compile and run it.

Fortran ignores case. Keywords such as Program can just as well be written as PrOgRaM.

A program optionally has a stop statement, which can return a message to the OS.

```
Program SomeProgram
   stop 'the code stops here'
End Program SomeProgram
```

Output

```
[basicf] stop:
STOP the code stops here
```

30.3 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 16 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.

30.3.1 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.
30.3.2 Statements

Let’s say a word about layout. Fortran has a ‘one line, one statement’ principle.

- 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:

  \[
  x = \text{very } & \\
  \text{long } & \\
  \text{expression}
  \]

30.3.3 Comments

Fortran knows only single-line *comments*, indicated by an exclamation point:

\[
\| x = 1 ! \text{set } x \text{ to one}
\]

Everything from the exclamation point onwards is ignored.

Maybe not entirely obvious: you can have a comment after a continuation character:

\[
\| x = f(a) \& ! \text{term1} \\
+ g(b) \! \text{term2}
\]

30.4 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:

\[
\| \text{Program YourProgram} \\
\text{implicit none} \\
! \text{variable declaration} \\
! \text{executable code} \\
\text{End Program YourProgram}
\]

A variable declaration looks like:

\[
\| \text{type [ , attributes ] :: name1 [ , name2, ... ]}
\]

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 30.4.1.
• the optional attributes are things such as dimension, allocatable, intent, parameter etcetera.
• name is something you come up with. This has to start with a letter.

The built-in data types of Fortran:

**Data types**

- Numeric: Integer, Real, Complex (section 30.6).
- Logical: Logical.
- Character: Character. Strings are realized as arrays of characters.
- Derived types: Type; chapter 36.

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 being a synonym for the value, rather than a variable you can assign to, with the parameter keyword.

\[
\text{real, parameter :: } \pi = 3.141592
\]

In chapter 39 you will see that parameters are often used for defining the size of an array.

Further specifications for numerical precision are discussed in section 30.4.1.2. Strings are discussed in chapter 35.

### 30.4.1 Declarations

#### 30.4.1.1 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

\[
\text{implicit none}
\]

immediately after any program or subprogram header. Now every variable needs to be given a type explicitly in a declaration.

#### 30.4.1.2 Variable ‘kind’s

Fortran has several mechanisms for indicating the precision of a numerical type.

\[
\text{integer(2) :: } i2 \\
\text{integer(4) :: } i4 \\
\text{integer(8) :: } i8 \\
\text{real(4) :: } r4 \\
\text{real(8) :: } r8 \\
\text{real(16) :: } r16 \\
\text{complex(8) :: } c8 \\
\text{complex(16) :: } c16 \\
\text{complex*32 :: } c32
\]
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.

**Variable kind and range** If you want to know what the kind and range of a type is, use kind and huge:

```plaintext
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
```

### 30.4.1.3 Kind selection

First of all, you can 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.

Likewise, you can specify the precision of a constant. Writing `3.14` will usually be a single precision real.

**Single/double precision constants**

```plaintext
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.

**Numerical precision** 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 $10^{+50}$ and $10^{+300}$ respectively.
### 30. Basics of Fortran

#### Storage size

**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.

#### Force a constant to be `real(8)`:

<table>
<thead>
<tr>
<th>Double precision constants</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>real(8) :: x,y</code></td>
</tr>
<tr>
<td><code>x = 3.14d0</code></td>
</tr>
<tr>
<td><code>y = 6.022e-23</code></td>
</tr>
</tbody>
</table>

- Use a compiler flag such as `-r8` to force all reals to be 8-byte.
- Write `3.14d0`
- `x = real(3.14, kind=8)`

#### 30.4.2 Range

You can use the function `huge` to query the maximum value of a type.

<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Integer :: idef</code></td>
</tr>
<tr>
<td><code>Real :: rdef</code></td>
</tr>
<tr>
<td><code>Real(8) :: rdouble</code></td>
</tr>
<tr>
<td><code>print 10,&quot;integer is kind&quot;,kind(idef)</code></td>
</tr>
<tr>
<td><code>print 10,&quot;integer max is&quot;,huge(idef)</code></td>
</tr>
<tr>
<td><code>print 10,&quot;real is kind&quot;,kind(rdef)</code></td>
</tr>
<tr>
<td><code>print 10,&quot;real max is&quot;,huge(rdef)</code></td>
</tr>
<tr>
<td><code>print 15,&quot;real8 is kind&quot;,kind(rdouble)</code></td>
</tr>
<tr>
<td><code>print 15,&quot;real8 max is&quot;,huge(rdouble)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output [typeof] def:</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer is kind 4</td>
</tr>
<tr>
<td>integer max is 2147483647</td>
</tr>
<tr>
<td>real is kind 4</td>
</tr>
<tr>
<td>real max is 0.3403E+39</td>
</tr>
<tr>
<td>real8 is kind 8</td>
</tr>
<tr>
<td>real8 max is 0.1798+309</td>
</tr>
</tbody>
</table>

With [ISO bindings](https://www.iso.org/obp/ui/index.html) there is a more systematic approach:

<table>
<thead>
<tr>
<th>Integer(kind=Int8) :: i8</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Integer(kind=Int16) :: i16</code></td>
</tr>
<tr>
<td><code>Integer(kind=Int32) :: i32</code></td>
</tr>
<tr>
<td><code>Integer(kind=Int64) :: i64</code></td>
</tr>
</tbody>
</table>
30.4.3 Initialization

Variables can be initialized in their declaration:

```fortran
integer :: i=2
real(4) :: x = 1.5
```

That this is done at compile time, leading to a common error:

```fortran
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.

30.5 Quick comparison Fortran vs C++

30.5.1 Statements

**Fortran statements** It’s all much like C++:

- Assignments:

```fortran
x = y
x = 2*y / (a+b)
```
30. Basics of Fortran

- I/O: next
- conditionals and loops: later
- function calls: later

30.5.2 Input/Output, or I/O as we say

Simple I/O

- Input:
  \[ \text{READ} *, n \]
- Output:
  \[ \text{PRINT} *, n \]

There is also \text{Write}.

The ‘star’ indicates that default formatting is used.
Other syntax for read/write with files and formats.

30.5.3 Expressions

Arithmetic expressions

- Pretty much as in C++
- Exception: \( r^a \) for power \( r^a \).
- Modulus is a function: \text{MOD}(7, 3).

Boolean expressions

- Long form:
  \[ \&\& \ .\neg \ .\| \]
  \[ \lt \ .\le \ .\eq \ .\ne \ .\ge \ .\gt \]
  \[ \text{true} \ .\false \]
- Short form:
  \[ < \leq \neq > \geq \]

Conversion and casting

Conversion is done through functions.

- \text{INT}: truncation; \text{NINT}: rounding
- \text{REAL, FLOAT, SNGL, DBLE}
- \text{CMPLX, CONJG, AIMAG}

\[ \text{http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-conv.htm} \]

Complex

Complex numbers exist; section 30.6.

Strings

Strings are delimited by single or double quotes.

For more, see chapter 35.
30.6 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**  Complex constants are written as a pair of reals in parentheses. There are some basic operations.

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
</table>
| `Complex` :: &
  `fourtynfivedegrees` = (1.,1.), &
  `number`, `rotated`

`Real` :: `x`, `y`  
`print *,"fourtynfivedegrees"
`x` = 3. ; `y` = 1.; `number` = `cmplx`(x,y)
`rotated` = `number` * `fourtynfivedegrees`
`print '("Rotated number has Re",f5.2," Im",f5.2)'

<table>
<thead>
<tr>
<th>Complexf08</th>
<th>complexf08</th>
</tr>
</thead>
<tbody>
<tr>
<td>45 degrees:</td>
<td>(1.00000000,1.00000000)</td>
</tr>
<tr>
<td>Rotated number has Re= 2.00 Im= 4.00</td>
<td></td>
</tr>
</tbody>
</table>

The imaginary root \( i \) is not predefined. Use

\[ ||\text{Complex, parameter} :: i = (0,1) \]

In Fortran2008, `Complex` is a derived type, and the real/imaginary parts can be extracted as

\[ ||\text{print *},\text{rotated%re},\text{rotated%im} \]

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
</table>
| `print *,"fourtynfivedegrees"
`x` = 3. ; `y` = 1.; `number` = `cmplx`(x,y)
`rotated` = `number` * `fourtynfivedegrees`
`print '("Rotated number has Re",f5.2," Im",f5.2)
`rotated%re`, `rotated%im` | `complexf08`: |
| 45 degrees: | (1.00000000,1.00000000) |
| Rotated number has Re= 2.00 Im= 4.00 |

**Exercise 30.2.**  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.

30.7 Review questions
Exercise 30.3. What is the output for this fragment, assuming \( i, j \) are integers?

```fortran
integer :: idiv
!! ...
i = 3 ; j = 2 ; idiv = i/j
print *,idiv
```

Exercise 30.4. What is the output for this fragment, assuming \( i, j \) are integers?

```fortran
real :: fdiv
!! ...
i = 3 ; j = 2 ; fdiv = i/j
print *,fdiv
```

Exercise 30.5. In declarations

```fortran
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 30.6. 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 30.7. In the following code, if \( \text{value} \) is nonzero, what do expect about the output?

```fortran
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 31

Conditionals

31.1 Forms of the conditional statement

The Fortran conditional statement uses the *if* keyword:

<table>
<thead>
<tr>
<th>Conditionals</th>
<th>Single line conditional:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if</strong> (test) <em>statement</em></td>
<td></td>
</tr>
</tbody>
</table>

The full if-statement is:

```fortran
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:

```fortran
checkx: if ( ... some test on x ... ) then
checky: if ( ... some test on y ... ) then
  ... code ...
end if checky
else checkx
  ... code ...
end if checkx
```

31.2 Operators

Comparison and logical operators
### 31. Conditionals

<table>
<thead>
<tr>
<th>Operator</th>
<th>old style</th>
<th>meaning</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>.eq.</td>
<td>equals</td>
<td>$x = y - 1$</td>
</tr>
<tr>
<td>/=</td>
<td>.ne.</td>
<td>not equals</td>
<td>$x * x \neq 5$</td>
</tr>
<tr>
<td>&gt;</td>
<td>.gt.</td>
<td>greater</td>
<td>$y &gt; x - 1$</td>
</tr>
<tr>
<td>&gt;=</td>
<td>.ge.</td>
<td>greater or equal</td>
<td>$\sqrt{y} \geq 7$</td>
</tr>
<tr>
<td>&lt;</td>
<td>.lt.</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>.le.</td>
<td>less or equal</td>
<td></td>
</tr>
<tr>
<td>.and.</td>
<td></td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>.or.</td>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td>.eqv.</td>
<td></td>
<td>equiv (iff, not XOR)</td>
<td></td>
</tr>
<tr>
<td>.neqv.</td>
<td></td>
<td>not equiv (XOR)</td>
<td></td>
</tr>
</tbody>
</table>

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

**Exercise 31.1.** Read in three grades: Algebra, Biology, Chemistry, each on a scale 1 · · · 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.

### 31.3 Select statement

The Fortran equivalent of the C++ `case` statement is `select`. It takes single values or ranges; works for integers and character strings.

**Select statement** Test single values or ranges, integers or characters:

```fortran
Select Case (i)
  Case (-1)
    print *, "Negative"
  Case (5)
    print *, "Five!"
  Case (0)
    print *, "Zero."
  Case (1:4,6:) ! can not have (1:)
    print *, "Positive"
end Select
```

Compiler does checking on overlapping cases!

Case values need to be constant expressions.

### 31.4 Boolean variables

The Fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`
Exercise 31.2. Print a boolean variable. What does the output look like in the true and false case?

Exercise 31.3. What is a conceptual difference between the C++ switch and the Fortran Select statement?
Chapter 32

Loop constructs

32.1 Loop types

Fortran has the usual indexed and ‘while’ loops. There are variants of the basic loop, and both use the \texttt{do} keyword. The simplest loop has a loop variable, an upper bound, and a lower bound.

\begin{verbatim}
Indexed Do loops

integer :: i

\texttt{do i=1,10}
\hspace{1cm} ! code with \texttt{i}
\texttt{end do}

You can include a step size (which can be negative) as a third parameter:

By steps of 3:

\texttt{do i=1,10,3}
\hspace{1cm} ! code with \texttt{i}
\texttt{end do}

Counting down:

\texttt{do i=10,1,-1}
\hspace{1cm} ! code with \texttt{i}
\texttt{end do}
\end{verbatim}

A couple of remarks about the loop variable:

- The loop variable is defined outside the loop, so it will have a value after the loop terminates.
- Non-integer loop variables are allowed, but be careful about the termination test. Because of floating point error you may run an iteration more or less than you were expecting.

Non-integer loop variables Used to be allowed-but-dangerous

Now deleted language feature:
32. Loop constructs

```
Code:
real(4) :: r
!
do r=.1,1.,.1
   print *,r
end do
```

```
Output
[loopf] loopr:
loopr.F90:16:5:
   16 | do r=.1,1.,.1
   | 1
   Warning: Deleted feature: Loop variable at (1) must be integer
loopr.F90:16:7:
   16 | do r=.1,1.,.1
   | 1
   Warning: Deleted feature: Start expression in DO loop at (1) must be integer
loopr.F90:16:10:
   16 | do r=.1,1.,.1
   | 1
   Warning: Deleted feature: End expression in DO loop at (1) must be integer
loopr.F90:16:13:
   16 | do r=.1,1.,.1
   | 1
   Warning: Deleted feature: Step expression in DO loop at (1) must be integer
   0.100000001
   0.200000003
   0.300000012
   0.400000006
   0.500000000
   0.600000024
   0.700000048
   0.800000072
   0.900000095
   1.00000012
```

Semantic fine points

- 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.

While loop  The while loop has a pre-test:

```
| do while (i<1000)
  | print *,i
  | i = i*2
| end do
```
32.2. Interruptions of the control flow

You can label loops with a construct name, which improves readability, but see also below.

```
outer: do i=1,10
   inner: do j=1,10
      ! something
   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.

**Remark 17** There is a legacy mechanism of label-terminated loops. Please do not use them.

### 32.2 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.

#### Exit and cycle

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++.

#### Labeled loops

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
```

Cycle and exit can apply to multiple levels, if the do-statements are labeled.

```
outer: do i = 1,10
   inner: do j = 1,10
```

Victor Eijkhout
32. Loop constructs

```fortran
if (i+j>15) exit outer
if (i==j) cycle inner
end do inner
end do outer
```

**Remark 18** 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.

### 32.3 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:

**Implied do loops** Normally, each print statement is on a new line; use an implied loop to print on one line.

```fortran
print *, (2*i, i=1,20)
```

You can iterate multiple expressions:

```fortran
print *, (2*i, 2*i+1, i=1,20)
```

These loops can be nested:

```fortran
print *, ( (i*j, i=1,20), j=1,20 )
```

Also useful for **Read**.

This construct is especially useful for printing arrays.

**Exercise 32.1.** Use the implied do-loop mechanism to print a triangle:

```fortran
1
2 2
3 3 3
4 4 4 4
```

up to a number that is input.

### 32.4 Review questions

**Exercise 32.2.** What is the output of:

```fortran
do i=1,11,3
   print *,i
end do
```
What is the output of:

\[
\begin{align*}
\text{do } & i=1,3,11 \\
\text{print } & *,i \\
\text{end do}
\end{align*}
\]
32. Loop constructs
Chapter 33

Scope

33.1 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.

33.1.1 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:

```fortran
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:

```fortran
Subroutine Foo()
  call Bar()
  call Bar()
End Subroutine Foo

Subroutine Bar()
  Integer :: i
  ! ‘i’ is created every time Bar is called
End Subroutine Bar
```

33.1.1.1 Variables in a module

Variables in a module (section 37.2) have a lifetime that is independent of the calling hierarchy of program units: they are static variables.
33. Scope

33.1.1.2 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 behaviour 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.

33.1.2 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 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. This is a keyword, but you won’t see it used initially.

If you structure your code in a single file, this is the recommended structure:

### Procedures in contains clause

```fortran
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
- In a `Module`; see section 37.2.
- Externally: the procedure is not internal to a `Program` or `Module`. In this case it’s safest to declare it through an `Interface` specification; section 42.1.

### 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:

```fortran
subroutine foo( <parameters> )
   <variable declarations>
   <executable statements>
end subroutine foo
```
Exit from a procedure can happen two ways:
1. the flow of control reaches the end of the procedure body, or
2. execution is finished by an explicit `return` statement.

A subroutine is invoked with a `call` statement:

```
call foo()
```

The `return` statement is optional in the first case. The `return` statement is different from C++ in that it does not indicate the return result of a function.

**Exercise 34.1.** Rewrite the above subroutine `foo` without a `return` statement.

```
subroutine foo()
  print *, "foo"
  if (something) return
  print *, "bar"
end subroutine foo
```

Output:
```
[funcf] printone:
  5
```

**Subroutine with argument**

**Code:**

```
program printing
  implicit none
  call printint(5)
contains
  subroutine printint(invalue)
    implicit none
    integer :: invalue
    print *, invalue
  end subroutine printint
end program printing
```

**Output:**
```
[funcf] printone:
  5
```

**Subroutine can change argument**

```
program printing
  implicit none
  call printint(5)
contains
  subroutine printint(invalue)
    implicit none
    integer :: invalue
    print *, invalue
    end subroutine printint
end program printing
```

```
[funcf] printone:
  5
```
34.2 Return results

Parameters are always ‘by reference’!

Recursive functions in Fortran need to be explicitly declared as such, with the recursive keyword.

**Recursion** Declare function as **Recursive Function**

```
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
```

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++

### Function definition and usage

- **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:** \( y = f(x) \)

### Function example

```fortran
program plussing
  implicit none
  integer :: i
  i = plusone(5)
  print *, i
contains
  integer function plusone(invalue)
    implicit none
    integer :: invalue
    plusone = invalue + 1 ! note!
  end function plusone
end program plussing
```

A function is not invoked with **call**, but rather through being used in an expression:

\[
\text{|| if } (\text{test}(3.0) \land \text{something}\_\text{else}) ... \text{||}
\]

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 37.2 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; negative input should be rejected. Fill in the missing lines in this code fragment:
34.2. Return results

**Code:**

```fortran
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.

```fortran
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

#### Why a ‘contains’ clause?

**Program NoContains**

```fortran
program NoContains
implicit none
end Program NoContains
```

**Program ContainsScope**

```fortran
program ContainsScope
implicit none
contains
  subroutine DoWhat(i)
    implicit none
    integer :: i
    i = 5
  end subroutine DoWhat
end Program ContainsScope
```

Warning only, crashes.  
Error, does not compile
34. Procedures

### Why a ‘contains’ clause, take 2

**Code:**

```fortran
Program NoContainTwo
  implicit none
  integer :: i=5
  call DoWhat(i)
end Program

subroutine DoWhat(x)
  implicit none
  real :: x
  print *,x
end subroutine DoWhat
```

**Output**

```
[funcf] nocontainatype:
nocantain2.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

**Procedure arguments** Arguments are declared in procedure body:

```fortran
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
```

Declaring the ‘intent’ is optional, but highly advisable.

**Parameter passing**

- 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.

**Fortran nomenclature** 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 arguments**:

```
  call f(x) ! 'x' is actual argument
```
34.3. Arguments

**Intent checking**  Compiler checks your intent against your implementation. This code is not legal:

```fortran
subroutine ArgIn(x)
  implicit none
  real, intent(in) :: x
  x = 5 ! compiler complains
end subroutine ArgIn
```

Why intent checking? Self-protection: if you state the intended behaviour of a routine, the compiler can detect programming mistakes.

Allow compiler optimizations:

```fortran
x = f()
call ArgOut(x)
print *, x
```

Call to \( f \) removed

```fortran
do i=1,1000
  x = ! something
  y1 = .... x ....
call ArgIn(x)
y2 = ! same expression as y1
end do
```

\( y2 \) 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 \texttt{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.

```fortran
! 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 arguments. You can mix positional and keyword arguments, but if you give one argument by keyword, all subsequent ones also need their keyword.

**Keyword arguments**

- Use the name of the formal parameter as keyword.
- Keyword arguments have to come last.
34. Procedures

### Code:

```fortran
contains
  subroutine say_xy(x,y)
    implicit none
    integer, intent(in) :: x,y
    print *,"x=",x," y=",y
  end subroutine say_xy
```

### Output

<table>
<thead>
<tr>
<th>funcf</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

A relation notion is that of **optional arguments**. 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`.

### Optional arguments

- 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 procedures**. This is as opposed to **module procedures**.

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

```fortran
|| 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.`
34.5. Local variable *save*-ing

```
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:

**Saved values**  Local variable is initialized only once, second time it uses its retained value.

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```integer function maxof2(i,j)
  implicit none
  integer,intent(in) :: i,j
  integer :: max=0
  if (i>max) max = i
  if (j>max) max = j
  maxof2 = max
end function maxof2``` | [funcf] save:
| Comparing: 1 3
  3 |
| Comparing: -2 -4
  3 |
34. Procedures
Chapter 35
String handling

35.1 String denotations
A string can be enclosed in single or double quotes. That makes it easier to have the other type in the string.

\[
\text{print } *, 'This string was in single quotes'
\]
\[
\text{print } *, 'This string in single quotes contains a single ' ' quote'
\]
\[
\text{print } *, "This string was in double quotes"
\]
\[
\text{print } *, "This string in double quotes contains a double " " quote"
\]

35.2 Characters

35.3 Strings
The length of a Fortran string is specified with the \texttt{len} keyword when the string is created:

\[
\text{character (len=50) :: mystring}
\]
\[
\text{mystring } = \text{ "short string"}
\]

The \texttt{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.

\begin{table}
\begin{tabular}{|l|l|}
\hline
\texttt{code:} & \texttt{output:} \\
\hline
\texttt{character (len=12) :: strvar} & \texttt{strlen:} \\
\texttt{!! ...} & \texttt{12} \\
\texttt{strvar } = \texttt{"word"} & \texttt{4} \\
\texttt{print *, len(strvar), len(trim(strvar))} & \texttt{strlen:} \\
\hline
\end{tabular}
\end{table}

To get the more intuitive length of a string, that is, the location of the last non-blank character, you need to \texttt{trim} the string.

\textbf{Intrinsic functions:} \texttt{LEN} (string), \texttt{INDEX} (substring,string), \texttt{CHAR} (int), \texttt{ICHAR} (char), \texttt{TRIM} (string)
35. String handling

Output
[stringf] concat:

Code:

character(len=10) :: firstname,lastname
without trimming: Victor
character(len=15) :: shortname,fullname
Eijkh
!! ...
with trimming: Victor
firstname = "Victor"; lastname = "Eijkhout"
Eijkhout
shortname = firstname // lastname
print *,"without trimming: ",shortname
fullname = trim(firstname) // " " // trim(lastname)
print *,"with trimming: ",fullname

35.4

Strings versus character arrays

35.5

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.

35.5.1

Format letters

x : one space
x5 : five spaces
a5 : strings, five positions wide
i3 : integer, three positions wide
f5.2 : fixed point, 5 positions wide, 2 positions for fractional part
e13.8 : floating point, 13 positions wide, 8 positions for fractional part
! f90:
b : binary
o : octal
z : hex

322

Introduction to Scientific Programming


35.5.2 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 *, i, j
print '(2i4)', i, j
print '(2i2)', i, j
```

```
OUTPUT
[iof] ii:
12 34
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.
```

To be precise, the format specifier is inside single quotes and parentheses, and consists of comma-separated specifications for a single item:

- ‘an’ specifies a string of \( n \) characters. If the actual string is longer, it is truncated in the output.
- ‘in’ specifies an integer of up to \( n \) digits. If the actual number takes more digits, it is rendered with asterisks. To use the minimum number of digits, use \( i0 \).
- ‘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.
- ‘en’ Exponent representation.
- Strings can go into the format:
  ```
  print '("Result:",3f5.3)', x, y, z
  ```
- ‘x’ for a space, ‘/’ for newline

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
35. String handling

Format repetitions

```
print '( 3i4 )', i1, i2, i3
print '( 3(i2,":\",f7.4) )', i1, r1, i2, r2, i3, r2
```

Repeats and line breaks

- 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 35.1. Use formatted I/O to print the number 0 · · · 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
```
Chapter 36

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.

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 declaration** Type name / End Type name block.

Member declarations inside the block:

```fortran
 type mytype
  integer :: number
  character :: name
  real(4) :: value
end type mytype
```

**Creating a type structure** Declare a type variable in the main program:

```fortran
Type (mytype) :: struct1, struct2
```

Initialize with type name:

```fortran
struct1 = mytype(1, 'my_name', 3.7)
```

Copying:

```fortran
struct2 = struct1
```

**Member access** Access structure members with %

(compare C++ dot-notation)

```fortran
Type (mytype) :: typed_struct
typed_struct%member = ....
```
36. Structures, eh, types

Example

```
| type point |
| real :: x, y |
| end type point |
```

```
| type(point) :: p1, p2 |
| p1 = point(2.5, 3.7) |
| p2 = p1 |
| print *, p2%x, p2%y |
```

Type definitions can go in the main program up top (or use a module; see later)

You can have arrays of types:

```
| type(my_struct) :: data |
| type(my_struct), dimension(1) :: data_array |
```

Structures as procedure argument  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 36.1.  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 `π`? So

```
(1,1) ⇒ 0.25
```

You can base this off the file `point.F90`. 
Exercise 36.2. 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 36.3. In the previous exercise 36.2:

Bonus points for using a module,
double bonus points for using an object-oriented solution.
36. Structures, eh, types
Chapter 37

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.

Module definition  Modules look like a program, but without main (only ‘stuff to be used elsewhere’):

```fortran
Module definitions
  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 definitions
```

Note also the numeric constant.

Module use  Module imported through `use` statement; comes before `implicit none`

Code:

```fortran
Program size
  use definitions
  implicit none
  type(point) :: p1, p2
  p1 = point(2.5, 3.7)
  p2 = p1
  print *, p2%x, p2%y
  print *, "length:", length(p2)
  print *, 2*pi
end Program size
```

Output

```
[structf] typermol:
  2.50000000 3.70000005
  length: 4.46542263
  6.2831854820251465
```
37. Modules

Exercise 37.1. Take exercise 36.2 and put all type definitions and all functions in a module.

37.1 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 19 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.

37.2 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.

F77 note: Modules are much cleaner than common blocks. Do not use those.

Any routines come after the contains clause.

A module is made available with the use keyword, which needs to go before the implicit none.

Module use

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 37.2. Write a module PointMod that defines a type Point and a function distance to make this code work:

```fortran
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:

```fortran
ifort -g -c pointmod.F90
ifort -g -c pointmain.F90
ifort -g -o pointmain pointmod.o pointmain.o
```

37.3 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.

Separate compilation of modules 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 modules, which can even further facilitate separate compilation.

37.4 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 behaviour 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.
37.5 Polymorphism

module somemodule

INTERFACE swap
MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
END INTERFACE

contains
subroutine swapreal
...
end subroutine swapreal
subroutine swapint
...
end subroutine swapint

37.6 Operator overloading

MODULE operator_overloading
IMPLICIT NONE
...
INTERFACE OPERATOR (+)
MODULE PROCEDURE concat
END INTERFACE

including the assignment operator:

INTERFACE ASSIGNMENT (=)
subroutine_interface_body
END INTERFACE

This mechanism can also be used for dot-operators:

INTERFACE OPERATOR (.DIST.)
MODULE PROCEDURE calcdist
END INTERFACE
Chapter 38

Classes and objects

38.1 Classes

Classes and objects  Fortran classes are based on type objects. Similarities and differences with C++.
- Same % syntax for specifying data members and methods.
- Data and functions declared separately.
- Object itself as extra parameter.

All will become clear . . .

Object is type with methods  You define a type as before, with its data members, but now the type has a contains for the methods:

```fortran
Module multmod
  type Scalar
    real(4) :: value
    contains
      procedure, public :: &
      printme, scaled
  end type Scalar
  contains ! methods
end module multmod
```

Object methods  Method call similar to C++
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)`.

**Method definition**  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
```

**Methods have object as argument**  You define functions that accept the type as first argument, but instead of declaring the argument as `Type`, you define it as `class`.

The members of the class object have to be accessed through the `%` operator.

```
subroutine set(p,xu,yu)
  implicit none
  class(point) :: p
  real(8), intent(in) :: xu,yu
  p%x = xu; p%y = yu
end subroutine set
```

In summary:
38.1. Classes

Class organization

- You’re pretty much forced to use Module
- A class is a Type with a contains clause followed by procedure declaration
- Actual methods go in the contains part of the module
- ⇒ First argument of method is the object itself.

Point program

```
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
```

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td>in the object</td>
<td>in the ‘type’</td>
</tr>
<tr>
<td>Methods</td>
<td>in the object</td>
<td>interface: in the type</td>
</tr>
<tr>
<td>Constructor</td>
<td>default or explicit</td>
<td>none</td>
</tr>
<tr>
<td>object itself</td>
<td>‘this’</td>
<td>first argument</td>
</tr>
<tr>
<td>Class members</td>
<td>global variable</td>
<td>accessed through first arg</td>
</tr>
<tr>
<td>Object’s methods</td>
<td>period</td>
<td>percent</td>
</tr>
</tbody>
</table>

Similarities and differences

Exercise 38.1. 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.

Exercise 38.2. 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?

38.1.1 Final procedures: destructors

The Fortran equivalent of destructors is a final procedure, designated by the final keyword.
38. Classes and objects

```fortran
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

38.2 Inheritance

More OOP Inheritance:
``` type, extends(baseclas) :: derived_class

Pure virtual:
``` type, abstract

Further reading
http://fortranwiki.org/fortran/show/Object-oriented+programming

Use modules! 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
    !! ...
```

Introduction to Scientific Programming
38.3 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
```

**Example class** 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:

**Methods just for testing**

```
subroutine set(v, x)
  implicit none
  class(ScalarField), intent(in) :: 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.

**Operator definition** Interface block:

```
interface operator (+)
  module procedure add
end interface operator (+)
```

**Exercise 38.3.** Extend the above example program so that the type stores an array instead of a scalar.

**Code:**

```fortran
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** field:

```
[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.

Similarly, we can redefine the assignment operator; see [https://dannyvanpoucke.be/oop-fortran-tut5-en/](https://dannyvanpoucke.be/oop-fortran-tut5-en). This comes with some complications regarding shallow copy and deep copy.
Chapter 39

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.

39.1 Static arrays

The preferred way for specifying an array size is:

```
Fortran dimension Preferred way of creating arrays through dimension keyword:

\[ \text{real}(8), \text{dimension}(100) :: x, y \]

One-dimensional arrays of size 100.

• Older mechanism works too:

\[ \text{integer} :: i(10, 20) \]

Two-dimensional array of size 10 × 20.

These arrays are statically defined, and only live inside their program unit (subroutine, function, module).

Such an array, with size explicitly indicated, is called a static array or automatic array. (See section 39.4 for dynamic arrays.)

1-based Indexing 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 most other languages.)

Note the use of parameter: compile-time constant

Size needs to be known to the compiler.
39. Arrays

**Lower bound** Unlike C++, Fortran can specify the lower bound explicitly:

```fortran
real, dimension(-1:7) :: x
do i=-1,7
    ... x(i) ...
end do
```

Safer:

```fortran
real, dimension(-1:7) :: array
integer :: idx
!! ...

do idx=lbound(array,1), ubound(array,1)
    array(idx) = 1+idx/10.
    print *, array(idx)
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.

### 39.1.1 Initialization

There are various syntaxes for *array initialization*, including the use of *implicit do-loops*:

**Array initialization** Different syntaxes:

- Explicit:
  ```fortran
  real, dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
  ```

- Implicit do-loop:
  ```fortran
  real5 = [ (1.01*i, i=1, size(real5,1)) ]
  ```

- Legacy syntax
  ```fortran
  real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
  ```

(Maybe for systems that do not have the [] characters?)

### 39.1.2 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:

```fortran
real*8, dimension(10) :: x, y
x = y
```

This obviously requires the arrays to have the same size. You can assign subarrays, or *array sections*, as long as they have the same shape. This uses a colon syntax.
### Array sections

- `:` to get all indices,
- `:n` to get indices up to `n`,
- `n:` to get indices `n` and up,
- `m:n` indices in range `m, ..., n`.

### Use of sections

**Code:**

```fortran
real(8), dimension(5) :: x = [1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0]
!! ...

x(2:5) = x(1:4)

print '(f5.3)', x
```

**Output**

[arrayf] sectionassign:

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.100</td>
</tr>
<tr>
<td>0.200</td>
</tr>
<tr>
<td>0.300</td>
</tr>
<tr>
<td>0.400</td>
</tr>
</tbody>
</table>

**Notes:**

- `1.0d0` explicit double precision: avoid loss of precision
- Format syntax will be discussed later:
  - float number, 5 positions, 3 after decimal point.

### Exercise 39.1

Code out the above array assignment 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 and 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 cannot trivially be converted to a simple loop.

### Exercise 39.2

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 sophisticated than assigning to a whole array or a section of it. For instance, you can use a stride:

**Strided sections** Copy a contiguous array to a strided subset of another:

**Code:**

```fortran
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:

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
You can even do arithmetic on array sections, for instance adding them together.

**Exercise 39.3.** Code \( \forall i: y_i = (x_i + x_{i+1})/2 \):

\[
\begin{array}{c}
 i  \\
\hline
 i  \\
\hline
 x_i  \\
 x_{i+1}  \\
\hline
 y_i  \\
\end{array}
\]

- 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.

### 39.1.3 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.

**Index arrays**

\[
\begin{align*}
\text{integer, dimension}(4) & : \ i = [2, 4, 6, 8] \\
\text{real}(4), \text{dimension}(10) & : \ x \\
\text{print} \ & *, x(i)
\end{align*}
\]

### 39.2 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:

**Multi-dimension arrays**

\[
\begin{align*}
\text{real}(8), \text{dimension}(20,30) & : \ array \\
array(i,j) & = 5./2
\end{align*}
\]

A useful function is **reshape**.
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 39.1.

To traverse the elements as they are stored in memory, you would need the following code:

```fortran
  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 39.4.** Can you describe in words how memory elements are access if you would write

```fortran
  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.

**Array sections in multi-D**

```fortran
  real(8), dimension(10) :: a,b
  a(1:9) = b(2:10)
```

or
39. Arrays

You can also use strides.

**Array printing** Fill array by rows:

\[
1 \quad 2 \quad ... \quad N \\
N + 1 \quad ... \\
... \\
MN
\]

**Code:**

```fortran
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
11.00000000 16.00000000
7.00000000 12.00000000
3.00000000 8.00000000
13.00000000 18.00000000
9.00000000 4.00000000
14.00000000
5.00000000 19.00000000
10.00000000
20.00000000
```

39.2.1 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

  ```fortran
  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:

  ```fortran
  array_of_lower = Lbound(symm)
  upper_in_dim_2 = Ubound(symm,2)
  ```

**Code:**

```fortran
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
```
39.3 Arrays to subroutines

- 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(5), y(5,4)
size(x)
size(y,2)
```

39.2.2 Reshaping

**RESHAPE**

```
array = RESHAPE( list, shape )
```

Example:

```
square = reshape( (/i, i=1,16)/), (/4,4/) )
```

**SPREAD**

```
array = SPREAD( old, dim, copies )
```

39.3 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:

**Pass array: main program**  Passing array as one symbol:

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(8), dimension(:) :: x(N) &amp;</td>
<td>[arrayf] arraypass1d:</td>
</tr>
<tr>
<td>real(8), dimension(:) :: y(0:N-1) &amp;</td>
<td>Sum of one-based array:</td>
</tr>
<tr>
<td></td>
<td>55.000</td>
</tr>
<tr>
<td>sx = arraysum(x)</td>
<td>Sum of zero-based array:</td>
</tr>
<tr>
<td>sy = arraysum(y)</td>
<td>55.000</td>
</tr>
<tr>
<td>print '(&quot;Sum of one-based array:&quot;,&quot;/4x,f6.3&quot;)', sx</td>
<td></td>
</tr>
<tr>
<td>print '(&quot;Sum of zero-based array:&quot;,&quot;/4x,f6.3&quot;)', sy</td>
<td></td>
</tr>
</tbody>
</table>

**Pass array: subprogram**  Note declaration as dimension(:) actual size is queried

```
real(8) function arraysum(x)
implicit none
real(8), intent(in), dimension(:) :: x
tmp = 0.
do i=1, size(x)
tmp = tmp+x(i)
end do
```
Arrays

arraysum = tmp
end function arraysum

The array inside the subroutine is known as a assumed-shape array or automatic array.

39.4 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.

Array allocation

```
real(8), dimension(:), allocatable :: x, y
n = 100
allocate(x(n), y(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)
```

39.4.1 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):

```
result = some_subroutine( some_array )
```

Introduction to Scientific Programming
39.5 Array output

The simple statement

```fortran
print *, A
```

will output the element of A in memory order; see section 39.2.

For a more sophisticated approach, the main thing to know is that each call to a format statement starts on a new line. Thus

```fortran
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 32.3:

```fortran
print '(10f7.3)', (A(i), i=1, size(A))
```

or for multiple dimensions:

```fortran
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:

39.6 Operating on an array

39.6.1 Arithmetic operations

Between arrays of the same shape:

```fortran
A = B+C  
D = D+E
```

(where the multiplication is by element).

39.6.2 Intrinsic functions

The following intrinsic functions are available for arrays:

```fortran
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
```
39. Arrays

Array intrinsics

• **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** returns 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

Multi-dimensional intrinsics

• Functions such as **Sum** operate on a whole array by default.
• To restrict such a function to one subdimension add a keyword parameter **DIM**:
  \[ s = \text{Sum}(A, \text{DIM}=1) \]
  where the keyword is optional.
• Likewise, the operation can be restricted to a **MASK**:
  \[ s = \text{Sum}(A, \text{MASK}=B) \]

Exercise 39.5. 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 **Functions** that compute these norms.


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?
39.6. Operating on an array

Print the maximum absolute distance between this and the reference result.

3. Use the \texttt{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?

39.6.3 Restricting with \texttt{where}

If an array operation should not apply to all elements, you can specify the ones it applies to with a \texttt{where} statement.

**Operate where**

\[
\| \texttt{where ( A<0 ) B = 0 } \|
\]

Full form:

\[
\begin{align*}
\text{\texttt{WHERE} ( logical argument )} \\
\text{sequence of array statements} \\
\text{\texttt{ELSEWHERE}} \\
\text{sequence of array statements} \\
\text{\texttt{END WHERE}}
\end{align*}
\]

39.6.4 Global condition tests

Reduction of a test on all array elements: \texttt{all}

\[
\begin{align*}
\text{REAL} (8), \text{dimension} (N,N) & : : A \\
\text{LOGICAL} & : : \text{positive, positive\_row(N), positive\_col(N)} \\
\text{positive} & = \text{\texttt{ALL}} ( A>0 ) \\
\text{positive\_row} & = \text{\texttt{ALL}} ( A>0, 1 ) \\
\text{positive\_col} & = \text{\texttt{ALL}} ( A>0, 2 )
\end{align*}
\]

**Exercise 39.7.** 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 $A_{\text{big}}$ and $A_{\text{small}}$ 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 $A_{\text{big}}$ and $A_{\text{small}}$, and test whether the sum is close enough to $A$.

Similar to \texttt{all}, there is a function \texttt{any} that tests if any array element satisfies the test.

\[
\| \texttt{if ( Any ( Abs (A-B) > } \|
\]
39. Arrays

39.7 Array operations

39.7.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

39.7.1.1 Slicing

If your loop assigns to an array from another array, you can use section notation:

\[
\begin{align*}
  a(:) &= b(:) \\
  c(1:n) &= d(2:n+1)
\end{align*}
\]

39.7.1.2 ‘forall’ keyword

The *forall* keyword also indicates an array assignment:

\[
\begin{align*}
  \text{forall} & (i=1:n) \\
  a(i) &= b(i) \\
  c(i) &= d(i+1) \\
  \text{end forall}
\end{align*}
\]

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

\[
\begin{align*}
  A &= [1,2,3,4,5] \\
  \text{do } i=1,4 \\
  & \quad A(i+1) = A(i) \\
  \text{end do} \\
  \text{print } '(5(i2x))', A
\end{align*}
\]

\[
\begin{align*}
  A &= [1,2,3,4,5] \\
  \text{do } i=4,1,-1 \\
  & \quad A(i+1) = A(i) \\
  \text{end do} \\
  \text{print } '(5(i2x))', A
\end{align*}
\]

Can you predict the output? Now consider the following:

<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
  A &= [1,2,3,4,5] \\
  \text{forall } (i=1:4) \\
  & \quad A(i+1) = A(i) \\
  \text{end forall} \\
  \text{print } '(5(i2x))', A
\end{align*}
\] |

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>[arrayf] forallf:</td>
</tr>
<tr>
<td>1 1 2 3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
  A &= [1,2,3,4,5] \\
  \text{do } i=4,1,-1 \\
  & \quad A(i+1) = A(i) \\
  \text{end do} \\
  \text{print } '(5(i2x))', A
\end{align*}
\] |

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>[arrayf] forallb:</td>
</tr>
<tr>
<td>1 1 2 3 4</td>
</tr>
</tbody>
</table>
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.

### 39.7.1.3 Do concurrent

#### 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:

```plaintext
do concurrent (i=1:n)
a(i) = b(i)
c(i) = d(i+1)
end do
```

(Do not use *for all*)

### 39.7.2 Loops without dependencies

Here are some illustrations of simple array copying with the above mechanisms.

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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 39.8.** Create arrays $A$, $C$ of length $2N$, and $B$ of length $N$. Now implement

$$B_i = \frac{(A_{2i} + A_{2i+1})}{2}, \quad i = 1, \ldots, N$$
and
\[ C_i = A_{i/2}, \quad i = 1, \ldots, 2N \]
using all four mechanisms. Make sure you get the same result every time.

### 39.7.3 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.

```plaintext
Original 1 2 3 4 5 6 7 8 9 10
Recursive 1 2 4 8 16 32 64 128 256 512
```

The slicing version of this:

```plaintext
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:

```plaintext
forall (i=2:n) counting(i) = 2*counting(i-1)
```

```plaintext
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:

```plaintext
do concurrent (i=2:n) counting(i) = 2*counting(i-1)
```

```plaintext
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.

### 39.8 Review questions
**Exercise 39.9.** Let the following declarations be given, and assume that all arrays are properly initialized:

\[
\begin{align*}
  \textit{real} & \quad :: \quad x \\
  \textit{real, dimension}(10) & \quad :: \quad a, \ b \\
  \textit{real, dimension}(10,10) & \quad :: \quad c, \ d
\end{align*}
\]

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?
39. Arrays
Chapter 40

Pointers

Pointers in C/C++ are based on memory addresses; Fortran pointers on the other hand, are more abstract.

40.1 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.

Dereferencing Fortran pointers are often automatically dereferenced: if you print a pointer you print the variable it references, not some representation of the pointer.

<table>
<thead>
<tr>
<th>Code:</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>real, target :: x</code></td>
<td><code>1.20000005</code></td>
</tr>
<tr>
<td><code>real, pointer :: point_at_real</code></td>
<td></td>
</tr>
<tr>
<td><code>x = 1.2</code></td>
<td></td>
</tr>
<tr>
<td><code>point_at_real =&gt; x</code></td>
<td></td>
</tr>
<tr>
<td><code>print *,point_at_real</code></td>
<td></td>
</tr>
</tbody>
</table>

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.
Setting the pointer

- You have to declare that a variable is pointable:
  \[
  \text{real, target :: x}
  \]
- Declare a pointer:
  \[
  \text{real, pointer :: point_at_real}
  \]
- Set the pointer with \(\Rightarrow\) notation (New! Note!):
  \[
  \text{point_at_real} \Rightarrow x
  \]

Using the pointer Now using \text{point_at_real} is the same as using \text{x}.

\[
\text{print *,point_at_real ! will print the value of x}
\]

Pointers can not just point at anything: the thing pointed at needs to be declared as \text{target}

\[
\text{real, target :: x}
\]

and you use the \(\Rightarrow\) operator to let a pointer point at a target:

\[
\text{point_at_real} \Rightarrow x
\]

If you use a pointer, for instance to print it

\[
\text{print *,point_at_real}
\]

it behaves as if you were using the value of what it points at.

**Pointer example**

**Code:**

\[
\text{real, target :: x, y} \\
\text{real, pointer :: that_real}
\]

\[
x = 1.2 \\
y = 2.4 \\
\text{that_real} \Rightarrow x \\
\text{print *,that_real} \\
\text{that_real} \Rightarrow y \\
\text{print *,that_real} \\
y = x \\
\text{print *,that_real}
\]

**Output**

\[
[\text{pointerf}] \text{ realp:}
\]

\[
1.20000005 \\
2.40000010 \\
1.20000005
\]

1. \text{that_real} points at \text{x}, so the value of \text{x} is printed.
2. \text{that_real} is reset to point at \text{y}, so its value is printed.
3. The value of \text{y} is changed, and since \text{that_real} still points at \text{y}, this changed value is printed.

Assign pointer from other pointer

\[
\text{real, pointer :: point_at_real, also_point}
\]
40.1. Basic pointer operations

point_at_real => x
also_point => point_at_real

Now you have two pointers that point at x.

Very important to use the =>, otherwise strange memory errors

If you have two pointers

\[ \text{real,pointer} :: \text{point_at_real,also_point} \]

you can make the target of the one to also be the target of the other:

\[ \text{also_point} => \text{point_at_real} \]

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.

Assignment subtleties  What happens if you want to write \( p2 \rightarrow p1 \) but you write \( p2= p1 \)?
The second one is legal, but has different meaning:

Assign underlying variables:  Crash because \( p2 \) pointer unassociated:

\[
\begin{align*}
\text{real,target} & : x,y \\
\text{real,pointer} & : p1,p2 \\
\hline
x & = 1.2 \\
p1 \rightarrow x \\
p2 \rightarrow y \\
p2 = p1 \quad \text{! same as } y=x \\
\text{print } *,p2 \quad \text{! same as print } y
\end{align*}
\[
\begin{align*}
\text{real,target} & : x \\
\text{real,pointer} & : p1,p2 \\
\hline
x & = 1.2 \\
p1 \rightarrow x \\
p2 = p1 \\
\text{print } *,p2
\end{align*}
\]

Exercise 40.1. Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

Code:

\[
\begin{align*}
\text{real,dimension}(10),\text{target} & : \text{array } & \\
\text{real,pointer} & : \text{biggest_element} \\
\text{print } '(10f5.2)',\text{array} \\
\text{call SetPointer(array,biggest_element)} \\
\text{print } *,"\text{Biggest element is"},\text{biggest_element} \\
\text{print } *,"\text{checking pointerhood:}";,\& \\
\text{associated(biggest_element)} \\
\text{biggest_element} = 0 \\
\text{print } '(10f5.2)',\text{array}
\end{align*}
\]

Output

[pointerf] arpointf:

\[
\begin{array}{cccccccc}
1.10 & 2.20 & 3.30 & 4.40 & 5.50 & 9.90  \\
8.80 & 7.70 & 6.60 & 0.00
\end{array}
\]

\text{Biggest element is} 9.89999962
\text{checking pointerhood: } T

\[
\begin{array}{cccccccc}
1.10 & 2.20 & 3.30 & 4.40 & 5.50 & 0.00  \\
8.80 & 7.70 & 6.60 & 0.00
\end{array}
\]

You can base this off the file \text{arpointf.F90}.  

Victor Eijkhout 357
40. Pointers

**Pointer status**

- **Nullify**: zero a pointer
- **Associated**: test whether assigned

**Code:**

```fortran
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
```

**Pointer allocation** If you want a pointer to point at something, but you don’t need a variable for that something:

**Code:**

```fortran
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++)

## 40.2 Pointers and arrays

You can set a pointer to an array element or a whole array.

```fortran
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 slices:

```fortran
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.
40.3  Example: linked lists

For pictures of linked lists, see section 67.1.2.

Linked list

- 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 40.2. 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.

40.3.1  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.

Linked list datatypes

- Node: value field, and pointer to next node.
- List: pointer to head node.

```
module 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.

Sample main  Our main program will create three nodes, and append them to the end of the list:
40. Pointers

40.3.2 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_node )
  implicit none
  ! parameters
  type(list),intent(inout) :: the_list
  type(node),intent(inout),pointer :: new_node

  ! if the list has no head node, attached the new node
  if ( .not. associated( the_list%head ) ) then
    nullify( new_node%next )
    the_list%head => new_node
  else
    ! go down the list, finding the last node
    current => the_list%head
    do while ( associated( current%next ) )
      previous => current
      current => current%next
    end do
    nullify( new_node%next )
    current%next => new_node
  end if
```

(Received solution; you can also do it recursively.)

40.3.3 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
Example: linked lists

Almost the same as before, but now keep the list sorted:

Code:

```
allocate(node_ptr); node_ptr%value = 1
call insert(the_list, node_ptr)
allocate(node_ptr); node_ptr%value = 5
call insert(the_list, node_ptr)
allocate(node_ptr); node_ptr%value = 3
call insert(the_list, node_ptr)
call print(the_list)
```

Output

```
[pointerf] listinsert:
List: [ 1,3,5, ]
```

Exercise 40.3. Copy the attach routine to insert, and modify it so that inserting a node will keep the list ordered.

You can base this off the file listfinsert.F90.

Exercise 40.4. Modify your code from exercise 40.3 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 40.5. 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 40.6. Write a length function for the linked list.
Try it both with a loop, and recursively.
Chapter 41

Input/output

41.1 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.

<table>
<thead>
<tr>
<th>I/O commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Print</strong> simple output to terminal</td>
</tr>
<tr>
<td>• <strong>Write</strong> output to terminal or file (‘unit’)</td>
</tr>
<tr>
<td>• <strong>Read</strong> input from terminal or file</td>
</tr>
<tr>
<td>• <strong>Open, Close</strong> for files and streams</td>
</tr>
<tr>
<td>• <strong>Format</strong> format specification that can be used in multiple statements.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formatted and unformatted I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Formatted: ascii output. This is good for reporting, but not for numeric data storage.</td>
</tr>
<tr>
<td>• Unformatted: binary output. Great for further processing of output data.</td>
</tr>
<tr>
<td>• Beware: binary data is machine-dependent. Use hdf5 for portable binary.</td>
</tr>
</tbody>
</table>

41.2 Print to terminal

The simplest command for outputing 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.
41. Input/output

41.2.1 Print on one line

The statement

\[
\text{print } *, \text{item1, item2, item3}
\]

will print the items on one line, as far as the line length allows.

**Implicit do loops**  Parametrized printing with an *implicit do loop*:

\[
\text{print } *, ( i \times i, i=1, n)
\]

All values will be printed on the same line.

41.2.2 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*:

\[
\text{print } *, ( A(i), i=1, n)
\]

41.3 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:

\[
\text{write } (\text{unit, format}) \text{ data}
\]

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.

41.3.1 Units

\[
\text{Open(11)}
\]

will result in a file with a name typically *fort11*.

\[
\text{Open(11, FILE=“filename“)}
\]

Many other options for error handling, new vs old file, etc.

After this:

\[
\text{Write } (11, \text{fmt}) \text{ data}
\]

Again options for errors and such.
41.4. Unformatted output

41.3.2 Other write options

\[\text{write(unit, fmt, ADVANCE="no") data}\]

will not issue a newline.

open Close

41.4 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.

Unformatted output  Indicated by lack of format specification:

\[\text{write (*) data}\]

Note: may not be portable between machines.
41. **Input/output**
Chapter 42

Leftover topics

42.1 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
end function f
```

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.

42.1.1 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.
42. Leftover topics

42.2 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)`:

```fortran
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:

```fortran
integer :: seedsize
integer, dimension(:), allocatable :: seed

call random_seed(size=seedsize)
allocate(seed(seedsize))
seed(:) = ! your integer seed here
call random_seed(put=seed)
```

42.3 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.

```fortran
integer :: clockrate, clock_start, clock_end
print *,"Ticks per second:",clockrate

call system_clock(count_rate=clockrate)
print *,"Time:",(clock_end-clock_start)/REAL(clockrate)
```
Chapter 43

Fortran review questions

43.1 Fortran versus C++

Exercise 43.1. 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.

43.2 Basics

Exercise 43.2.

- What does the \texttt{Parameter} keyword do? Give an example where you would use it.
- Why would you use a \texttt{Module}?
- What is the use of the \texttt{Intent} keyword?

43.3 Arrays

Exercise 43.3. 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:

\begin{verbatim}
Integer :: year, high, low
!! code omitted
read *,year,high,low
\end{verbatim}

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 32°F/0°C’.

43.4 Subprograms

Exercise 43.4. 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:

```fortran
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

JULIA (LARGELY YET TO BE WRITTEN)
Chapter 44

Basics of Julia

44.1 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.

**Program statements**

- A program contains statements. A simple statement is one that fits on one line. No need for a semicolon: the line end terminates it.
  
  ```julia
  a = 1
  ```

- You can multiple statements on one line by using semicolons:

  ```julia
  a = 1; b = 2
  ```

- Compound statements, such as conditionals, take up multiple lines;

  ```julia
  if a == 1
  println("It was one")
  end
  ```

- Comments are ‘Note to self’, short:

  ```julia
  c = 3 # set c to three
  ```

**Printing stuff** The basics of interacting with a program are through the output statements `print` and `println`, and input `readline`.

```julia
a = 1; b = 2
print("a and b: ")
print(a)
print(" ")
println(b)
```

or simpler:

```julia
println("a and b: $a $b")
```
44. Basics of Julia

44.2 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 place in memory.

Variable names are more liberal than in other programming languages: many Unicode characters are allowed.

44.2.1 Datatypes

In Julia, like in Python, but unlike in C++, you do not need a variable declaration: a variable is created the first time you assign something to it, and its type is the type of the value you assign.

Variables have

- a name
- a type
- a value.

The type of the variable is determined by when a value is assigned to it. This is like python, and unlike C.

**Numerical types** The main numerical types are \texttt{Int64} and \texttt{Float64}, where the ‘64’ indicates that in both cases these are 64-bit types.

These are ‘concrete types’, but they are both based on the ‘abstract type’ \texttt{Real}, which is a supertype of both; see later.

**Type of expression** Types can be determined by looking at the rhs expression, or explicitly forced:

```
Code:

\begin{verbatim}
one_int = 1
println("$one_int has type $(typeof(one_int))")
one_int = one(Int64)
println("$one_int has type $(typeof(one_int))")

one_float = 1.0
println("$one_float has type $(typeof(one_float))")
one_float = one(Float64)
println("$one_float has type $(typeof(one_float))")
\end{verbatim}

Output

[basic] scalar:

```
make[2]: julia: No such file or directory
make[2]: *** [run_scalar] Error 1
```
44.2. Variables

Subtypes and supertypes  Types have a hierarchy.
The subtype relation is denoted with the <: relation:

\[
\text{Integer <: Number}
\]

This can also be used in functions:

\[
\text{function somecompute(x::T) where T <: Real}
\]

44.2.1 Boolean values

Truth values  So far you have seen integer and real variables. There are also boolean values, type Bool, which represent truth values.

\[
\text{primitive type Bool <: Integer 8 end}
\]

There are only two values: \texttt{true} and \texttt{false}.

\[
\begin{align*}
\text{bool found} &= \texttt{false}; \\
\text{found} &= \texttt{true};
\end{align*}
\]

To create an array of booleans you can

\[
\begin{align*}
\text{alltrue} &= \texttt{trues(n)} \\
\text{allfalse} &= \texttt{falses(n)}
\end{align*}
\]

44.2.1.2 Characters and strings

Characters and strings

- Characters, of type \texttt{Char}, in single quotes
- Strings, of type \texttt{String}, in double quotes.

In addition to the regular 128 ASCII characters, Unicode characters can be entered through a backslash notation: \texttt{\gamma}. This follows LATEX naming conventions.

44.2.2 Truth values

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

Boolean expressions

- Testing: == != < > <= >=
- Not, and, or: ! && ||

Bool-int conversion  As in other languages, booleans can be implicitly interpreted as integer, with \texttt{true} corresponding to 1 and \texttt{false} to 0. However, conversion the other way is not as generous.

\[
1+\texttt{true}
\]
Logical expressions in C++ are evaluated using shortcut operators: you can write
\[
x > 0 \land \sqrt{x} < 2
\]
If \( x \) is negative, the second part will never be evaluated because the ‘and’ conjunction can already be concluded to be false. Similarly, ‘or’ conjunctions will only be evaluated until the first true clause.

The ‘true’ and ‘false’ constants could strictly speaking be stored in a single bit. C++ does not do that, but there are bit operators that you can apply to, for instance, all the bits in an integer.

### 44.3 Expressions

Division \(2/3\) gives a \texttt{Float64} number. You can cast to integer:
\[
\texttt{half} = \texttt{Int64((27+2)/2)}
\]

#### 44.3.1 Terminal input

To make a program run dynamic, you can set starting values from keyboard input. For this, use \texttt{readline}, which takes keyboard input. To assign this input to a variable you need to parse that as the desired type, using the function \texttt{parse}.
\[
\texttt{n = parse(Int64, readline(stdin))}
\]

### 44.4 Tuples

\[
(a, b) = (1, 2)
\]
even
\[
(a, b) = (b, a)
\]
Chapter 45

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 $x > 0$, 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.

45.1 Basic syntax

Conditionals are delimited by if and end.

```
if $x < 0$
  $x = 2*x; y = y/2;$
else
  $x = 3*x; y = y/3;$
end
```

Chaining conditionals (where the dots stand for omitted code):

```
if $x>0$
  ....
elseif $x<0$
  ....
else
  ....
end
```

Nested conditionals:

```
if $x>0$
  if $y>0$
    ....
  else
    ....
  end
else
  ....
end
```

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 45.1. For what values of $x$ will the left code print ‘b’?
For what values of $x$ will the right code print ‘b’?

<table>
<thead>
<tr>
<th>$x = ##$ some float value</th>
<th>$x = ##$ some float value</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $x &gt; 1$</td>
<td>if $x &gt; 1$</td>
</tr>
<tr>
<td>println(&quot;a&quot;)</td>
<td>println(&quot;a&quot;)</td>
</tr>
<tr>
<td>if $x &gt; 2$</td>
<td>elseif $x &gt; 2$</td>
</tr>
<tr>
<td>println(&quot;b&quot;)</td>
<td>println(&quot;b&quot;)</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

45.2 Advanced topics

45.2.1 Short-circuit operators

Julia logic operators have a feature called short-circuiting: a short-circuit operator stops evaluating 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`.

This mechanism allows you to write

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

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

45.2.2 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 perform in effect the same action, and only differ by an expression?

```
if (foo)
    x = 5;
else
    x = 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:
\[ x = \text{foo} ? 5 : 6; \]

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

\[ \text{foo} ? x : y = 7; \]
Chapter 46

Looping

See chapter 6 for a general introduction to loops.

46.1 Basics

Julia loops have a structure:

```julia
for var = some_range
    # statements
end
```

or

```julia
for var in some_range
    # statements
end
```

Code:

```julia
function sum_of_cubes(n)
cubesum = 0.
for num in 1:n
    cubesum += num*num*num
end
return cubesum
end
```

```julia
n = 10
sumofcubes = sum_of_cubes(n)
println("Sum of $n cubes: $sumofcubes")
```

Output

```
[loopj] loop:
make[2]: julia: No such file or directory
make[2]: *** [run_loop] Error 1
```

There is a subtlety using a loop at top level, for instance in the interpreter: not only the loop variable, but every scalar variable in the loop is then local. To access a variable from outside the loop you need to declare it `global`:
Note that we said ‘scalar’ variable. This does not hold for array variable: those are always global.

### 46.1.1 Stride

Loops by default have a stride of 1. To use something else, write

```julia
for odd=1:2:10
    f(odd)
end
```

or

```julia
for down=10:-1:1
    f(down)
end
```

**Exercise 46.1.** Read an integer value with `cin`, and print ‘Hello world’ that many times.

**Exercise 46.2.** Extend exercise 6.1: the input 17 should now give lines

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

Loops such as

```julia
for i in 1:10
    ...
end
```

have well-defined starting and ending values. However, loops can also iterate over more abstractly defined iteration spaces. You will see this later.

### 46.1.2 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.
This is called \textit{loop nest}; the \textit{row}-loop is called the \textit{outer loop} and the \textit{col}-loop the \textit{inner loop}.

For a more compact syntax:

\begin{verbatim}
for i in 1:10, j in 1:10
    ...
end
\end{verbatim}

This syntax makes it easier to \textit{break} out of multiple loops.
Chapter 47

Functions

Read the introduction of chapter 7 for a general discussion of what functions are about.

Here is a single-line function definition:

```plaintext
Code:
double(x) = 2*x
println(double(6))
```

Output

```
[funcj] simpleone:
/bin/sh: julia: command not found
```

You may have noticed that this function definition does not include parameter or return types. Those are determined by the function invocation.

Technically, Julia uses multiple dispatch (which is known as polymorphism in other languages): the same function name can correspond to different computations, as long they differ in their type!signature.

This (multiline) function prints out its argument type:

```plaintext
Code:
function verydouble(x)
    println("Arg type \\$(typeof(x)): \$x")
    2*x
end
println(verydouble(15))
println(verydouble(15.1))
println(verydouble([1,2]))
```

Output

```
[funcj] simplemulti:
/bin/sh: julia: command not found
```

You can enforce an argument type

```
myfunction( x::Float64 ) = #
```
47. Functions
Chapter 48

Arrays


48.1 Introduction

Create array of length 3, with values undefined.

```
Array{Int64}(undef, 3)
```

Create array with 2 rows and 3 columns, with values undefined.

```
Array{Integer}(undef, 2, 3)
```

Zero-based indexing:

```
mat = Array(Float64)(undef, 2, 4)
typeof(mat)
show(mat)
mat[1, 3] = 3.14
show(mat)
```

Bound checking:

```
mat = Array(Float64)(undef, 2, 4)
mat[2, 4] = 3.14
mat[0, 0] = 1.1
show(mat)
```

48.1.1 Element access

Julia uses the square bracket notation

```
ar[5] = 6
```

with bounds that are 1-based.

(Fortran-style arbitrary lower bounds can be supported: https://docs.julialang.org/en/v1/devdocs/offset-arrays/)
48. Arrays

48.1.2 Arrays in linear algebra

The keywords Vector and Matrix are synonyms for

\[
\begin{align*}
\text{Array}\{T,1\} \\
\text{Array}\{T,2\}
\end{align*}
\]

respectively.

Additionally, there are

- Symmetric: Symmetric matrix
- Hermitian: Hermitian matrix
- UpperTriangular: Upper triangular matrix
- UnitUpperTriangular: Upper triangular matrix with unit diagonal
- LowerTriangular: Lower triangular matrix
- UnitLowerTriangular: Lower triangular matrix with unit diagonal
- UpperHessenberg: Upper Hessenberg matrix
- Tridiagonal: Tridiagonal matrix
- SymTridiagonal: Symmetric tridiagonal matrix
- Bidiagonal: Upper/lower bidiagonal matrix
- Diagonal: Diagonal matrix
- UniformScaling: Uniform scaling operator

For some of these types, optimized addition and multiplication routines are available.

48.1.3 Type hierarchy

There is an untyped base type AbstractMatrix from which typed matrices are derived. You can use the function eltype to find out what the type of an abstract matrix is:

\[
\text{function}\ \text{look\_at\_matrix}(\ A::\text{AbstractMatrix}\ )
\]

\[
(\text{size},\text{type}) = (\text{size}(A,1),\text{eltype}(A))
\]

48.2 Functions

48.2.1 Built-in functions

collect: turn the elements of a collection or iterator into an array. Example:

\[
\text{integers} = \text{collect}(1:n)
\]

48.2.2 Scalar functions

Scalar functions such as abs can be applied in a pointwise manner to an array by using a dot-notation. Since this is ambiguous with taking a member of a structure, the array needs to be enclosed in parentheses:
48.3. Comprehensions

48.2.3 Vectors are dynamic

An array can be grown or shrunk after its creation. For instance, you can use the `push!` method to add elements at the end:

**Dynamic vector extension** Extend with `push!`:

```
Code:
foo = [1, 2, 3];
push!(foo, 4, 5, 6);
```

**Output**
```
[arrayj] push:
make[2]: julia: No such file or directory
make[2]: *** [run_push] Error 1
```

48.3 Comprehensions

It is possible to define an array by its elements.

```
[i^2 for i in 1:10]
Complex[i^2 for i in 1:10]
[(i, sqrt(i)) for i in 1:10]
```

Two-dimensional:
```
[(r, c) for r in 1:5, c in 1:2]
```

With a test to filter:
```
[x for x in 1:100 if x % 7 == 0]
```

Enumerating:
```
[i for i in enumerate(m)]
3x3 Array{Tuple{Int64, Int64}, 2}:
  (1, 6) (4, 5) (7, 3)
  (2, 4) (5, 0) (8, 7)
  (3, 1) (6, 7) (9, 4)
```

Zip together multiple generators:
```
for i in zip(0:10, 100:110, 200:210)
  println(i)
end
```
Arrays

You can iterators:

\[ \text{ro} = 0:2:100 \]
\[ [i \text{ for } i \in \text{ro}] \]

48.4 Multi-dimensional cases

Julia has support for multi-dimensional arrays. Array types look like

\[ \text{Array(Int64, 2)} \]

for a two-dimensional array of Int64 objects.

Array organization Arrays are organized by columns. For instance using \texttt{reshape} to make a two-dimensional array from one-dimensional one:

\begin{verbatim}
    a = reshape(1:100, (10, 10))
    for n in a
        print(n, " ")
    end
\end{verbatim}
Chapter 49

Structs

49.1 Why structures?
You have seen the basic datatypes in section 4.3.4. These are enough to program whatever you want, but it would be nice if the language had some datatypes that are more abstract, closer to the terms in which you think about your application. For instance, if you are programming something to do with geometry, you had rather talk about points than explicitly having to manipulate their coordinates.

Structures are a first way to define your own datatypes. A struct acts like a datatype for which you choose the name. A struct contains other datatypes; these can be elementary, or other structs.

```haskell
v1 = vector(1.,2.,5)
v2 = vector(3.,4.,6)
println("v2 has y=$(v2.y)")
v2 = v1
println("v2 has y=$(v2.y)")
```

The reason for using structures is to ‘raise the level of abstraction’: instead of talking about $x, y$-values, you can now talk about vectors and such. This makes your code look closer to the application you are modeling.

49.2 The basics of structures
A structure behaves like a data type: you declare variables of the structure type, and you use them in your program. The new aspect is that you first need to define the structure type. This definition can be done anywhere before you use it.

```haskell
mutable struct vector
    x::Float64
    y::Float64
    label::Int64
end
```
49.3 Structures and functions

You can use structures with functions, just as you could with elementary datatypes.

Julia here uses *multiple dispatch* (which is a form of polymorphism) where a function can receive additional definitions for operating on user-defined types.

49.3.1 Structures versus objects

Multiple dispatch on structures are Julia’s alternative to Object-Oriented Programming (OOP): rather than having methods inside a defined class, functions are defined free-standing, operating on newly defined types.

While there is not much difference between `obj.method()` and `method(obj)`, things like inheritance are harder to realize.

**Exercise 49.1.** If you are doing the prime project (chapter 52) you can now do exercise 52.8.
Chapter 50

More Julia

50.1 Assertions

Test on dynamic conditions

```julia
function twice_root( x::T )
    @assert( x>0 )
    return 2*sqrt()
end
```
PART V

EXERCISES AND PROJECTS
Chapter 51

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 judgement.

**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 in advisable. The academic XSEDE has the following guidelines: https://www.xsede.org/terminology; much longer and more extensive, the Google developer documentation style guide is also a great resource.

**Completeness** Your writeup needs to have the same elements as a good paper:
- Title and author, including EID.
- A one-paragraph abstract.
- A bibliography at the end.

**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 your findings.

**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 should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a verbatim environment, but using the listings package (used in this book) is very much recommended.

**Results and discussion** Present tables and/or graphs when appropriate, but also include verbiage to explain what conclusions can be drawn from them.

You can also discuss possible extensions of your work to cases not covered.
51. Style guide for project submissions

**Summary**  Summarize your work and findings.
Chapter 52

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.

52.1 Arithmetic

Before doing this section, make sure you study 4.5 Expressions section.4.5.

Exercise 52.1. Read two numbers and print out their modulus. Two ways:
• use the cout function to print the expression, or
• assign the expression to a variable, and print that variable.

52.2 Conditionals

Before doing this section, make sure you study 5.1 Conditionals section.5.1.

Exercise 52.2. Read two numbers and print a message stating whether the second is as divisor of the first:
52. Prime numbers

Code:
```c++
int number, divisor;
bool is_a_divisor;
/* ... */
if (/* ... */)
    {cout << "Indeed, " << divisor
     << " is a divisor of "
     << number << endl;
    }
else {
    cout << "No, " << divisor
     << " is not a divisor of "
     << number << endl;
}
```

Output

```
[primes] division:
( echo 6 ; echo 2 ) | 1
divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) | 1
divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

52.3 Looping

*Before doing this section, make sure you study 6.1 The ‘for’ loop section.*

**Exercise 52.3.** 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 52.4.** Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

**Exercise 52.5.** Read in an integer *r*. If it is prime, print a message saying so. If it is not prime, find integers *p* ≤ *q* so that *r* = *p* · *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*.

52.4 Functions

*Before doing this section, make sure you study 7 Functions chapter.*
Above you wrote several lines of code to test whether a number was prime.

**Exercise 52.6.** Write a function `test_if_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```c
int main() {
    bool isprime;
    isprime = test_if_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?

### 52.5 While loops

**Exercise 52.7.** Take your prime number testing function `test_if_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.)

### 52.6 Structures

Before doing this section, make sure you study section 15.1 References. In the C programming language, a struct functions to bundle together a number of data item. The C++ struct does this, but, like a class, it can also contain methods.

The exercises in this chapter only use data encapsulation as a preliminary to full-fledged classes with method encapsulation. You can do these exercises, or skip straight to section 52.7.

**Exercise 52.8.** Rewrite the exercise that found a predetermined number of primes, putting the `number_of_primes_found` and `last_number_tested` variables in a structure. Your main program should now look like:

```c
int main() {
    bool isprime;
    isprime = test_if_prime(13);
}
```

Hint: the variable `last_number_tested` does not appear in the main program. Where does it get
updated? Also, there is no update of number_of_primes_found in the main program. Where do you think it would happen?

52.7 Classes and objects

Before doing this section, make sure you study 9.1 What is an object? section 9.1.

In exercise 52.8 Structuresexcounter.52.8 you made a structure that contains the data for a primesequence, and you have separate functions that operate on that structure or on its members.

Exercise 52.9. Write a class primegenerator that contains:

- Methods how_many_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:

```cpp
int number = sequence.nextprime();
```

In the previous exercise you defined the primegenerator class, and you made one object of that class:

```cpp
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other.

Exercise 52.10. 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 52.9 Classes and objects exercise 52.9.

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.

Exercise 52.11. The Goldbach conjecture says that every even number \( 2n \) (starting at 4), is the sum of two primes \( p + q \):

\[
2n = p + q.
\]
52.7. Classes and objects

Equivalently, every number \( n \) is equidistant from two primes:

\[
 n = \frac{p + q}{2} \quad \text{or} \quad q - n = n - p.
\]

In particular this holds for each prime number:

\[
 \forall r \text{ prime} \exists p, q \text{ prime}: r = (p + q)/2 \text{ is prime}.
\]

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?

For each \( r \) value, when the program finds the \( p, q \) values, print the \( q, p, r \) triple and move on to the next \( r \).

Allocate an array where you record all the \( p - q \) distances that you found. Print some elementary statistics, for instance: what is the average, do the distances increase or decrease with \( p \)?

52.7.1 Exceptions

Before doing this section, make sure you study 23.2.2 Exception handling subsection.

Exercise 52.12. Revisit the prime generator class (exercise 52.9 Classes and objects subsection) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 24.6 Limits subsection.)

```cpp
try {
  do {
    auto cur = primes.nextprime();
    cout << cur << endl;
  } while (true);
} catch (string s) {
  cout << s << endl;
}
```

Output

```
[primes] genx:
9931
9941
9949
9967
9973
Reached max int
```

52.7.2 Prime number decomposition

Before doing this section, make sure you study 24.2.1 Maps: associative array subsection.

Design a class \( \texttt{Integer} \) which stores its value as its prime number decomposition. For instance,

\[
 180 = 2^2 \cdot 3^3 \cdot 5 \implies [2:2, 3:3, 5:1]
\]

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.
52. Prime numbers

**Exercise 52.13.** 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:**

```cpp
Integer i2(2);
cout << i2.as_string() << " : "
   << i2.as_int() << endl;

Integer i6(6);
cout << i6.as_string() << " : "
   << i6.as_int() << endl;
```

**Output**

```
[primes] decomposition26:
2^1 : 2
2^1 3^1 : 6
```

**Exercise 52.14.** Extend the previous exercise to having multiplicity > 1 for the prime factors.

**Code:**

```cpp
Integer i180(180);
cout << i180.as_string() << " : "
   << i180.as_int() << endl;
```

**Output**

```
[primes] decomposition180:
2^2 3^2 5^1 : 180
```

Implement addition and multiplication for Integers.

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.

52.8 Other

The following exercise requires `std::optional`, which you can learn about in section 24.5.2Optional-subsection.24.5.2.

**Exercise 52.15.** Write a function `first_factor` that optionally returns the smallest factor of a given input.

```cpp
auto factor = first_factor(number);
if (factor.has_value())
   cout << "Found factor: " << factor.value() << endl;
```

52.9 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, . . .
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
   \[7, 11, 13, 17, 19, 23, 29 \ldots\]

52.9.1 Arrays implementation

The sieve can be implemented with an array that stores all integers.

**Exercise 52.16.** 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.

52.9.2 Streams implementation

The disadvantage of using an array is that we need to allocate an array. What’s more, the size is deter-
mined 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 52.17.** Write a stream class that generates integers and use it through a pointer.

```
for (int i=0; i<7; i++)
    cout << "Next int: "
    << the_ints->next() << endl;
```

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 52.18.** 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.
Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

**Exercise 52.19.** 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.

### 52.10 Range implementation

*Before doing this section, make sure you study 26.2 Iterating over classes section.26.2.*

**Exercise 52.20.** Make a `primes` class that can be ranged:

**Code:**

```cpp
primegenerator allprimes;
for (auto p : allprimes)
    cout << p << " 
    if (p>100) break;

```

**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,
```

### 52.11 User-friendliness

Use the `cxxopts` package (section 65.2 Options processing: `cxxopts` section.65.2) to add command-line options to some primality programs.

**Exercise 52.21.** 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 53

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.

53.1 Basic functions

Exercise 53.1. 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 53.2. 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:

```cpp
const float pi = 2*acos(0.0);
float x{1.0}, y{0.0};
rotate(x, y, pi/4);
std::cout << "Rotated halfway: (" << x << "," << y << ")" << std::endl;
rotate(x, y, pi/4);
std::cout << "Rotated to the y-axis: (" << x << "," << y << ")" << std::endl;
```

Output

[geom] rotate:
Rotated halfway: (0.707107,0.707107)
Rotated to the y-axis: (0,1)

53.2 Point class

Before doing this section, make sure you study 9.1What is an object?chapter.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 53.3. Make class \texttt{Point} with a constructor
\begin{verbatim}
\texttt{Point( float xcoordinate, float ycoordinate );}
\end{verbatim}

Write the following methods:
- \texttt{distance\_to\_origin} returns a float.
- \texttt{printout} uses \texttt{cout} to display the point.
- \texttt{angle} computes the angle of vector \((x, y)\) with the \(x\)-axis.

Exercise 53.4. Extend the \texttt{Point} class of the previous exercise with a method: \texttt{distance} that computes the distance between this point and another: if \(p, q\) are \texttt{Point} objects,
\begin{verbatim}
\texttt{p.distance(q)}
\end{verbatim}
computes the distance between them.

\textbf{Hint:} remember the ‘dot’ notation for members.

Exercise 53.5. Write a method \texttt{halfway\_point} that, given two \texttt{Point} objects \(p, q\), construct the \texttt{Point} halfway, that is, \((p + q)/2:\)
\begin{verbatim}
\texttt{Point p(1,2.2), q(3.4,5.6);}
\texttt{Point h = p.halfway(q)};
\end{verbatim}

You can write this function directly, or you could write functions \texttt{Add} and \texttt{Scale} and combine these. (Later you will learn about operator overloading.)

How would you print out a \texttt{Point} to make sure you compute the halfway point correctly?

Exercise 53.6. Make a default constructor for the point class:
\begin{verbatim}
\texttt{Point() { /* default code */ }}
\end{verbatim}

which you can use as:
\begin{verbatim}
\texttt{Point p;}
\end{verbatim}

but which gives an indication that it is undefined:
\begin{verbatim}
\texttt{Point p3;}
\texttt{cout << "Uninitialized point:" << endl;}
\texttt{p3.printout();}
\texttt{cout << "Using uninitialized point:" << endl;}
\texttt{auto p4 = Point(4,5)+p3;}
\texttt{p4.printout();}
\end{verbatim}

\textbf{Hint:} see section 4.6.5Number values and undefined valuessubsection.4.6.5.
53.3. Using one class in another

Exercise 53.7. Revisit exercise 53.2 Basic functionsexcounter.53.2 using the Point class. Your code should now look like:

```c++
newpoint = point.rotate(alpha);
```

Exercise 53.8. Advanced. Can you make a Point class that can accommodate any number of space dimensions? Hint: use a vector; section 10.3 Vector are a class section.10.3. Can you make a constructor where you do not specify the space dimension explicitly?

53.3 Using one class in another

Before doing this section, make sure you study 9.2 Inclusion relations between classes section.9.2.

Exercise 53.9. Make a class LinearFunction with a constructor:

```c++
LinearFunction( Point input_p1, Point input_p2 );
```

and a member function

```c++
float evaluate_at( float x );
```

which you can use as:

```c++
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 53.10. Make a class LinearFunction with two constructors:

```c++
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

```c++
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 53.11. Revisit exercises 53.2 Basic functionsexcounter.53.2 and 53.7 Point classexcounter.53.7, introducing a Matrix class. Your code can now look like

```c++
newpoint = point.apply(rotation_matrix);
```

or

```c++
newpoint = rotation_matrix.apply(point);
```

Can you argue in favour 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.

Axi-parallel rectangle class  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 53.12.

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 topege_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 53.13. 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.

53.4 Is-a relationship

Before doing this section, make sure you study 9.3Inheritance section 9.3.

Exercise 53.14. 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 53.15. 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.
53.5 More stuff

*Before doing this section, make sure you study 15.3 Reference to class members section. 15.3.*

The `Rectangle` class stores at most one corner, but may be convenient to sometimes have an array of all four corners.

**Exercise 53.16.** Add a method

```cpp
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.
Chapter 54

Zero finding

54.1 Root finding by 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 54.1.

In section 54.2 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.

54.1.1 Simple implementation

Before doing this section, make sure you study 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 69.2.)
54. Zero finding

54.1.2 Polynomials

First of all, we need to have a way to represent polynomials. For a polynomial of degree \( d \) we need \( d + 1 \) coefficients:

\[
f(x) = c_0 x^d + \cdots + c_{d-1} x + c_d
\]

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:

```cpp
TEST_CASE("coefficients are polynomial","[1]"
) {
  auto coefficients = set_coefficients();
  REQUIRE(coefficients.size()>0);
  REQUIRE(coefficients.front()!=0.);
}
```

**Exercise 54.1.** Write a routine `set_coefficients` that constructs a vector of coefficients:

```cpp
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 `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 `proper_polynomial` you should check that

- it recognizes correct polynomials, and
- it fails for improper coefficients that do not properly define a polynomial.

**Exercise 54.2.** Write a function `proper_polynomial` as described, and write unit tests for it, both passing and failing.

Next we need polynomial evaluation. You can interpret the array of coefficients in (at least) two ways, but with equation (54.1) 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:

```cpp
// correct interpretation: 2x^2 + 1
vector<double> second{2,0,1};
REQUIRE(proper_polynomial(second));
REQUIRE(evaluate_at(second,2) == Catch::Approx(9));
// wrong interpretation: 1x^2 + 2
REQUIRE(evaluate_at(second,2) != Catch::Approx(6));
```
54.1. Root finding by bisection

(where we have left out the TEST_CASE header.)

Now we write the function that passes these tests:

**Exercise 54.3.** Write a function `evaluate_at` which computes

\[ y \leftarrow f(x). \]

and confirm that it passes the above tests.

For bonus points, look up *Horner's rule* and implement it.

With the polynomial function implemented, we can start working towards the algorithm.

### 54.1.3 Left/right search points

![Figure 54.2: Setting the initial search points](image)

Suppose \( x_- , x_+ \) are such that \( x_- < x_+ \), and \( f(x_-) \cdot f(x_+) < 0 \), 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 54.1.

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 54.4.** Make the following code work:

```cpp
if ( not is_odd(coefficients) ) {
    cout << "This program only works for odd-degree polynomials\n";
    exit(1);
}
```
You could test the above as:

```cpp
def find_outer():
    second = [2, 0, 1]  # 2x^2 + 1
    require(not is_odd(second))
    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 54.5. Write a function `find_outer` which computes $x_-, x_+$ such that

$$f(x_-) < 0 < f(x_+) \text{ or } f(x_+) < 0 < f(x_-)$$

How can you compute this test more compactly? What is a good prototype for the function? How do you 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 particular 23.2.2.3) for polynomials of even degree:

```cpp
right = left + 1;
second = [2, 0, 1]  # 2x^2 + 1
require_throws(find_initial_bounds(second, left, right));
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?

54.1.4 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:

```cpp
def find_zero():
    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 54.6. Write a function `move_bounds` and test it.

Implement some unit tests on this function.

Put it all together:

Exercise 54.7. Make this call work:

```cpp
auto zero = find_zero(coefficients, left, right);
cout << "Found root " << zero
<< " with value " << evaluate_at(coefficients, zero) << "\n";
```

Add an optional precision argument to the root finding function.

Design unit tests, including on the precision attained, and make sure your code passes them.

54.1.5 Object implementation

Revisit the exercises of section 54.1.1 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 \((1 + x)^n\)?

54.1.6 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?

54.2 Newton’s method

*Before doing this section, make sure you study lambda functions; chapter 13.*

Exercise 54.8. The Newton method (HPC book, section-17) 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:

```cpp
double f(double x) { return x*x-2; };
double fprime(double x) { return 2*x; };
```

and the algorithm:

```cpp
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 \(g\).
54. Zero finding

You can base this off the file newton.F90.

You would like the Newton loop of the previous exercise to be contained in a function. This means you have to pass the objective function and the derivative as arguments:

\[ \text{double} \ root = \text{find_zero}( f, g ); \]

Exercise 54.9. Rewrite the Newton exercise above to use a function with prototype

\[ \text{double} \ root = \text{find_zero}( f, g ); \]

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 54.10. Extend the newton exercise to compute roots in a loop:

\[
\text{for} \ ( \text{int} \ n=2; \ n<8; \ n++) \ \{ \\
\quad \text{cout} \ll \text{"sqrt(”} << \ n \ll \text{") = “} \ll \text{find_zero(} \\
\quad \quad \text{/* ... */} \\
\quad \quad \text{<< \ “\n”;} \\
\text{\} }
\]

Without lambdas, you would define a function

\[ \text{double squared_minus_n( double x, int n )} \{ \text{return} \ x*x-n \}; \]

However, the \text{find_zero} function takes a function of only a real argument. Use a capture to make \( f \) dependent on the integer parameter.

Exercise 54.11. You don’t need the gradient as an explicit function: you can approximate it as

\[ f'(x) = (f(x + h) - f(x))/h \]

for some value of \( h \).

Write a version of the root finding function

\[ \text{double find_zero( function< double(double) > f )} \]

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 \text{find_zero} you coded earlier.
Chapter 55

Eight queens

A famous exercise for recursive programming is the *eight queens* problem: Is it possible to position eight queens on a chessboard, so that no two queens ‘threaten’ each other, according to the rules of chess?

55.1 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 55.1.** 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 $n \geq 4$. 
55. Eight queens

55.2 Solving the eight queens problem, basic approach

This problem requires you to know about arrays/vectors; chapter 10. Also, see chapter 69 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

```cpp
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.

55.3 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:

```cpp
ChessBoard(int n);
```

This is a ‘generalized chess board’ of size $n \times n$, and initially it should be empty.

**Exercise 55.2.** 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:

```cpp
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 55.3. Write this method and make sure that it passes the test for an empty board.

```cpp
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:

```cpp
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.

```cpp
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 55.4. Write this method, and make sure it passes the following test for valid and invalid column numbers:

```cpp
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:

```cpp
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.

```cpp
ChessBoard empty(n);
REQUIRE(empty.feasible());
```
Eight queens

ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );

Exercise 55.5. Write the method and make sure it passes these tests.

We shouldn’t only do successfull 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 55.6. 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 55.7. 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.

55.4 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>:
The recursive solution method

- 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 55.2, this placement method has roughly the following structure:

```cpp
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

```cpp
ChessBoard next = *this;
```

makes a copy of the object you’re in.

**Remark 20** Another approach would be to make a recursive function

```cpp
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 55.8.** Write a method

```cpp
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 55.9.** Write the method

```cpp
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 55.10.** Use the constructor

```cpp
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:

```cpp
ChessBoard almost( 4, (1,3,0) );
```
Eight queens

```cpp
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:

```cpp
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:

```cpp
TEST_CASE( "there are 4x4 solutions","[10]" ) {
    ChessBoard four(4);
    auto solution = four.place_queens();
    REQUIRE( solution.has_value() );
}
```

**Exercise 55.11.** (Optional) Can you modify your code so that it counts all the possible solutions?

**Exercise 55.12.** (Optional) How does the time to solution behave as function of $n$?
Chapter 56

Infectious disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

56.1 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.

56.1.1 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].


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
Infectious disease simulation

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.

56.2 Coding

56.2.1 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 56.1. 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:

```cpp
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() << endl;
    if (joe.is_stable())
        break;
}
```

Here is a suggestion how you can model disease status. Use a single integer with the following interpretation:
56.2. Coding

- 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 21** Consider a point of programming style. Now that you’ve modeled the state of a person with an integer, you can use that as

```cpp
void infect(n) {
    if (state==0)
        state = n;
}
```

But you can also write

```cpp
bool is_susceptible() {
    return state==0;
}
void infect(n) {
    if (is_susceptible())
        state = n;
}
```

Which do you prefer and why?

56.2.2 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?
Disease ran its course by step 6
```

**Remark 22** 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 56.2.** Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:
56. Infectious disease simulation

```c
|| 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 24.8.1C random functions subsection 24.8.1), or the new STL one in section 24.8 Random numbers section 24.8.
- 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.
```

56.2.3 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 56.3.** Read in a number $0 \leq p \leq 1$ representing the probability of disease transmission upon contact.

```c
|| population.set_probability_of_transfer(probability);
```

Incorporate this into the program: in each step the direct neighbours of an infected person can now get sick themselves. Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

**Exercise 56.4.** 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?

56.2.4 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](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 56.5. 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 56.6. 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 this probability 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.

56.2.5 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.

56.3 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?
56. Infectious disease simulation

56.4 Project writeup and submission

56.4.1 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 19.2.2 Header files subsection 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.

56.4.2 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 56.2.4 Spread subsection 56.2.4 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.

56.5 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 56.7.** Code this scheme. What is the effect of varying \( dt \)?

**Exercise 56.8.** For the disease to become an epidemic, the number of newly infected has to be larger than the number of recovered. That is,

\[ \lambda S_i I_i - \gamma I_i > 0 \iff S_i > \gamma/\lambda. \]

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.
56. Infectious disease simulation
Chapter 57

Google PageRank

57.1 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 57.1. 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:

```cpp
auto homepage = make_shared<Page>("My Home Page");
cout << "Homepage has no links yet:" << endl;
cout << homepage->as_string() << endl;
```

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:

```cpp
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() << endl;
```

Exercise 57.2. Add some more links to your homepage. Write a method random_click for the Page class. Intended code:

```cpp
for (int iclick=0; iclick<20; iclick++) {
```
57. Google PageRank

57.2 Clicking around

Exercise 57.3. 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:

```cpp
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 57.4. 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 57.5. Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

```cpp
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.

57.3 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 57.6.** 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.

### 57.4 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(); |
```

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 57.7.** 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.

**Exercise 57.8.** 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?
57. Google PageRank

Compare the result to the random walk exercise above.

**Exercise 57.9.** 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 57.10.** Add a page that you will artificially made look important: add a number of pages (for instance four times the average number of links) that 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?

57.5 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 57.11.** 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 58

Redistricting

In this project you can explore ‘gerrymandering’, the strategic drawing of districts to give a minority population a majority of districts.

58.1 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.
58. Redistricting

58.2 Basic functions

58.2.1 Voters

We dispense with census districts, expressing everything in terms of voters, for which we assume a known voting behaviour. 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 58.1. Implement a `Voter` class. You could for instance let ±1 stand for A/B, and 0 for undecided.

```cpp
cout << "Voter 5 is positive:" << endl;
Voter nr5(5,+1);
cout << nr5.print() << endl;

cout << "Voter 6 is negative:" << endl;
Voter nr6(6,-1);
cout << nr6.print() << endl;

cout << "Voter 7 is weird:" << endl;
Voter nr7(7,3);
cout << nr7.print() << endl;
```

58.2.2 Populations

Exercise 58.2. 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 part 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:

```cpp
cout << "Making district with one B voter";
Voter nr5(5,+1);
District nine( nr5 );
cout << ". size: " << nine.size() << endl;
cout << ". lean: " << nine.lean() << endl;
/* ... */
cout << "Making district ABA" << endl;
District nine( vector<Voter>
  ( {1,-1},{2,+1},{3,-1} ) );
cout << ". size: " << nine.size() << endl;
cout << ". lean: " << nine.lean() << endl;
```

Output

```
[gerry] district:
Making district with one B voter
.. size: 1
.. lean: 1
Making district ABA
.. size: 3
.. lean: -1
```
58.2. Basic functions

Exercise 58.3. Implement a Population class that will initially model a whole state.

Code:

```cpp
string pns( "-++--" );
Population some(pns);
cout << "Population from string " << pns << endl;
cout << ". size: " << some.size() << endl;
cout << ". lean: " << some.lean() << endl;
Population group=some.sub(1,3);
cout << "sub population 1--3 " << endl;
cout << ". size: " << group.size() << endl;
cout << ". lean: " << group.lean() << endl;
```

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.

58.2.3 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 58.4. Write a class Districting that stores a vector of District objects. Write size and lean methods:

Code:

```cpp
cout << "Making single voter population B" << endl;
Population people( vector<Voter>{ Voter(0,+1) } );
cout << ". size: " << people.size() << endl;
cout << ". lean: " << people.lean() << endl;
Districting gerry;
cout << "Start with empty districting:" << endl;
cout << ". number of districts: " << gerry.size() << endl;
```

Output

```
[gerry] gerryempty:
Making single voter population B
.. size: 1
.. lean: 1
Start with empty districting:
.. number of districts: 0
```

Exercise 58.5. Write methods to extend a Districting:

```cpp
cout << "Add one B voter:" << endl;
gerry = gerry.extend_with_new_district( people.at(0) );
cout << ". number of districts: " << gerry.size() << endl;
cout << ". lean: " << gerry.lean() << endl;
cout << "add A A:" << endl;
gerry = gerry.extend_last_district( Voter(1,-1) );
gerry = gerry.extend_last_district( Voter(2,-1) );
cout << ". number of districts: " << gerry.size() << endl;
cout << ". lean: " << gerry.lean() << endl;
cout << "Add two B districts:" << endl;
gerry = gerry.extend_with_new_district( Voter(3,+1) );
```

Victor Eijkhout 439
58. Redistricting

```cpp
gerry = gerry.extend_with_new_district(Voter(4,+1));
cout << ".. number of districts: " << gerry.size() << endl;
cout << ".. lean: " << gerry.lean() << endl;
```

58.3 Strategy

![Diagram showing the recursive approach to redistricting.]

Figure 58.1: Multiple ways of splitting a population

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 58.1 Multiple ways of splitting a population.

- 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.
• Keep the districting that gives the strongest minority rule, over all values of \( p \).

You can now realize the above simple example:

\[
AAABB \Rightarrow AAA|B|B
\]

**Exercise 58.6.** Implement the above scheme.

```cpp
Population five("+++--");
cout << "Redistricting population: " << endl;
cout << ".. majority rule: " << five.print() << endl;
int ndistricts{3};
auto gerry = five.minority_rules(ndistricts);
cout << gerry.print() << endl;
cout << ".. minority rule: " << gerry.rule() << endl;
```

**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 58.7.** Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

### 58.4 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 58.8.** 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.
58. Redistricting

Exercise 58.9. Code a dynamic programming solution to the redistricting problem.

58.5 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 58.10. 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 58.11. Look up the definition of efficiency gap (and ‘wasted votes’), and implement it in your code.

58.6 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 59

Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

59.1 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.

59.2 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.

59.2.1 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 59.1.** Code a class `Address` with the above functionality, and test it.
59. Amazon delivery truck scheduling
<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
</table>
| Address one(1, 1.),
  two(2, 2.);
  cerr << "Distance: "
    << one.distance(two)
    << "\n"; |

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>[amazon] address:</td>
</tr>
<tr>
<td>Address</td>
</tr>
<tr>
<td>Distance: 1.41421</td>
</tr>
<tr>
<td>.. address</td>
</tr>
<tr>
<td>Address 1 should be closest to the depot. Check: 1</td>
</tr>
</tbody>
</table>

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 59.2.** 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.

### 59.2.2 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.

### 59.2.3 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

```cpp
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

**Exercise 59.3.** 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:
59.3. 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](image)

Figure 59.1: Illustration of the ‘opt2’ idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [12], 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 59.1 illustrates the ‘opt2’ idea of reversing part of a path. Figure 59.1 shows that the path 1 − 2 − 3 − 4 can be made shorter by reversing part of it, giving 1 − 4 − 3 − 2 − 1. 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:

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.
for all nodes \( m < n \) on the path \([1..N]\):
make a new route from 
\([1..m-1] + [m--n].\text{reversed} + [n+1..N] \)
if the new route is shorter, keep it

**Exercise 59.4.** 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.

**Exercise 59.5.** What is the runtime complexity of this heuristic solution?

**Exercise 59.6.** 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.

### 59.4 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 59.2 Extending the ‘opt2’ idea to multiple paths figure. 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.

**Exercise 59.7.** Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 59.3 Multiple paths test case figure. 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.
59.5 Amazon prime

In section 59.4 Multiple trucks section, 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 59.8.** 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.

59.6 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 59.9.** Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

59.7 Ethics

People sometimes criticize Amazon’s labor policies, including regarding its drivers. Can you make any observations from your simulations in this respect?
Figure 59.3: Multiple paths test case
Chapter 60

High performance linear algebra

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.

60.1 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:

```c
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 60.1.** 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:

$$
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}
$$

Then

$$
C_{11}=A_{11}B_{11} + A_{12}B_{21},
C_{12}=A_{11}B_{12} + A_{12}B_{22},
C_{21}=A_{21}B_{11} + A_{22}B_{21},
C_{22}=A_{21}B_{12} + A_{22}B_{22}
$$

(Convince yourself that this actually computes the same product $C = A \cdot B$.)
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. Historically, linear algebra software has used columnwise storage, meaning that the location of an element $(i, j)$ is computed as $j + i \cdot M$ (we will use zero-based indexing throughout this project, both for code and mathematical expressions.)

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. For this reason, linear algebra software treats each matrix like a submatrix, described by three parameters $M, N, LDA$, where ‘LDA’ stands for ‘leading dimension of $A$’ (see Basic Linear Algebra Subprograms (BLAS) [10], and Lapack [1]). This is illustrated in figure 60.1 Submatrix out of a matrix, with m,n,lda

![Figure 60.1: Submatrix out of a matrix, with m,n,lda of the submatrix indicated](image)

**Exercise 60.2.** 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 10.8.5Spansubsection.10.8.5.

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 24.5.3Variantsubsection.24.5.3), 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.

```cpp
// example values for M, LDA, N
M = 2; LDA = M+2; N = 3;
```
60.2. Matrix storage

```cpp
// 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. Read up on section 10.8.2Old-style arrayssubsection.10.8.2.)

Exercise 60.3.  Start implementing the Matrix class with a constructor

```cpp
Matrix::Matrix(int m, int lda, int n, double *data)
```

and private data members:

```cpp
private:
    int m,n,lda;
    span<double> data;
```

Write a method

```cpp
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 60.4.  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:

```cpp
#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

```cpp
auto get_double_data() {
    double *adata;
    adata = data.data();
    return adata;
}
```

Exercise 60.5.  Implement this. Use a cpp #define macro for the optimized indexing expression. (See section 21.2.3Parametrized macrossubsection.21.2.3.)

60.2.1 Submatrices

Next we need to support constructing actual submatrices. Since we will mostly aim for decomposition in 2 × 2 block form, it is enough to write four methods:

Victor Eijkhout
where, for instance, \( \text{Left}(5) \) gives the columns with \( j < 5 \).

**Exercise 60.6.** Implement these methods and test them.

### 60.3 Multiplication

You can now write a first multiplication routine, for instance with a prototype

\[
\text{void Matrix::MatMult( Matrix& other, Matrix& out );}
\]

Alternatively, you could write

\[
\text{Matrix Matrix::MatMult( Matrix& other );}
\]

but we want to keep the amount of creation/destruction of objects to a minimum.

#### 60.3.1 One level of blocking

Next, write

\[
\text{void Matrix::BlockedMatMult( Matrix& other, Matrix& out );}
\]

which uses the \( 2 \times 2 \) form above.

#### 60.3.2 Recursive blocking

The final step is to make the blocking recursive.

**Exercise 60.7.** Write a method

\[
\text{void RecursiveMatMult( Matrix& other, Matrix& out );}
\]

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.

### 60.4 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.
Exercise 60.8. Read up on cache memory, and 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 60.9. 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?

60.4.1 Parallelism (optional)

The four clauses of equation 60.1 Mathematical preliminaries equation.60.1.1 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`.

60.4.2 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:

```c
#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 60.10. 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 [9], and
- it probably involves a certain amount of assembly coding.
High performance linear algebra
Chapter 61

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 62

Cryptography

62.1 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:

```java
Integer big = 2000000000; // two billion
big *= 1000000;
bigger = big + 1;
Integer one = bigger % big;
```

**Exercise 62.1.** Code Farey sequences.

62.2 Cryptography


62.3 Blockchain

Implement a blockchain algorithm.
Chapter 63

DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

63.1 Basic functions

Refer to section 24.4.

First we set up some basic mechanisms.

**Exercise 63.1.** 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

```c
char BaseComplement(char);
```

**Exercise 63.2.** 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.

63.2 De novo shotgun assembly

One approach to generating a genenome 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 genenome.)

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).
63. DNA Sequencing

63.2.1 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.

63.2.2 De Bruijn graph assembly

63.3 ‘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.

63.3.1 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 63.3.** 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 contain 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?

63.3.2 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.

\[
\text{vvvv match location} \\
\text{antidisestablishmentarianism} \\
\text{blis} \\
\text{``bad character''}
\]

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:

\[
\text{vvvv match location} \\
\text{antidisestablishmentarianism} \\
\text{blis}
\]

in fact, we move it further, to the first match on the first character of the pattern:

\[
\text{vvvv match location} \\
\text{antidisestablishmentarianism} \\
\text{blis} \\
\text{``first character match''}
\]

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.

\[
\text{shooobeedooobeeboobah} \\
\text{edoobeeboob} \\
\text{``mismatch''} \\
\text{``other occurrence of ‘d’''}
\]

Note that this can be a considerably smaller shift than in the previous case.

\[
\text{shooobeedooobeeboobah} \\
\text{edoobeeboob} \\
\text{``match the bad character ‘d’} \\
\text{``new location''}
\]

Exercise 63.4. 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:

\[
\text{desistrust}
\]
listrest
   ^^good suffix

desistrust
   listrest
   ^^next occurrence of suffix
Chapter 64

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 behaviour 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 [11].)

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 39Arrays): 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.

64.1 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/](https://data.giss.nasa.gov/gistemp/).

**Exercise 64.1.** Start by making a listing of the available years, and an array `monthly_deviation` of size $12 \times n_{\text{years}}$, where $n_{\text{years}}$ 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.

64.2 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,
Climate change
each point has a chance of $1/n$ to be a record high. Since over $n$ 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 64.2.** 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:

$$\text{PrevRec}(m, y) = \begin{cases} y & \text{if } \text{MonDev}(m, y) = \max_{m'}(\text{MonDev}(m', y)) \\ y' & \text{if } \text{MonDev}(m, y) < \text{MonDev}(m, y') \\
\text{and } \text{MonDev}(m, y') = \max_{m'' < m'}(\text{MonDev}(m'', y)) \end{cases}$$

Again, use array notation. This is also a great place to use the `Where` clause.

**Exercise 64.3.** 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 64.4.** Copy the code from [http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf](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 64.5.** 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.
PART VI

ADVANCED TOPICS
Chapter 65

External libraries

If you have a C++ compiler, you can write as much software as you 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?

65.1 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 oftet 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.

65.1.1 Using an external library

Using a software library typically means that

- your program has a line
  \[
  \text{#include "fancylib.h"}
  \]
- and you compile and link as:
  \[
  \text{icpc -c yourprogram.cxx -I/usr/include/fancylib}
  \]
  \[
  \text{icpc -o yourprogram yourprogram.o -L/usr/lib/fancylib -lfancy}
  \]

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.
65.1.2 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`

65.1.2.1 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
65.2 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 command-line options are to be interpreted.
- You could use the cxxopts library. This is what we will be exploring now.

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:

  ```
  options.add_options()
    ("h,help", "usage information")
    ("n,nsize", "size of the thing",
     cxxopts::value<int>()->default_value("4096"))
    // et cetera
  ;
  ```

- Parse the options:

  ```
  auto result = options.parse(argc, argv);
  ```

- Get result values:

  ```
  if (result.count("help")>0) {
    std::cout << options.help() << std::endl;
    return 0;
  }
  int array_size = result["nsize"].as<int>();
  ```

Options can be specified the usual ways:

- 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
  ```

Victor Eijkhout 471
myprogram -n 10
myprogram --nsize 100
myprogram --nsize=1000

Exercise 65.1. Incorporate this package into primality testing: exercise 52.21.

65.3 Catch2 unit testing

Test a simple function

```c
int five() { return 5; }
```

Successful test:
### 65.3. Catch2 unit testing

#### Code

```cpp
TEST_CASE("needs to be 5", "[1]") {
    REQUIRE(five() == 5);
}
```

#### Output

```
[catch] require:

Undefined symbols for architecture x86_64:
"Catch::Generators::GeneratorUntypedBase::
  GeneratorUntypedBase()", referenced from:
  CATCH2_INTERNAL_TEST_6
() in require.o
  Catch::Generators::Generators<int>::"
Generators() in require.o
  Catch::Generators::Generators<int>::"
Generators() in require.o
  Catch::Generators::Generators<int>::
Generators<
  Catch::Generators::GeneratorWrapper<int, int, int>::
(Catch::Generators::GeneratorWrapper<int>&&,
  int&&, int&&, int&&) in require.o
  Catch::Generators::SingleValueGenerator<int
>::"SingleValueGenerator()
  in require.o
  Catch::Generators::SingleValueGenerator<int
>::"SingleValueGenerator()
  in require.o
"Catch::Generators::
  acquireGeneratorTracker(
    Catch::StringRef, Catch::
    SourceLineInfo const&")", referenced from:
  CATCH2_INTERNAL_TEST_6
() in require.o
"Catch::
  BinaryExpr<int, int>::
  streamReconstructedExpression
  (std::__1::basic_ostream<
    char, std::__1::
    char_traits<char>&>
    const in require.o
"Catch::
  makeTestInvoker(
    void (*)(()))", referenced from:
  __GLOBAL__sub_I_require
  .cxx in require.o
"Catch::
  handleExpr(Catch::
    ITransientExpression const
    &)", referenced from:
  CATCH2_INTERNAL_TEST_0
() in require.o
  CATCH2_INTERNAL_TEST_2
() in require.o
"Catch::
  handleThrowingCallSkipped
```

---

Victor Eijkhout
65. External libraries

Unsuccessful test:
CODE

```cpp
TEST_CASE( "not six","[2]" ) {
  REQUIRE( five()==6 );
}
```

OUTPUT

```cpp
[catch] requireerr:
```

```
Undefined symbols for architecture x86_64:
  "Catch::Generators::GeneratorUntypedBase::GeneratorUntypedBase()", referenced from:
  CATCH2_INTERNAL_TEST_6() in require.o
  Catch::Generators::Generators<int>:::
  Catch::Generators::Generators<int>:::
  Catch::Generators::Generators<int>:::
  Catch::Generators::Generators<int>:::
  Catch::Generators::GeneratorWrapper<int, int, int>(Catch::Generators::GeneratorWrapper<int>&&, int&&, int&&, int&&) in require.o
  Catch::Generators::SingleValueGenerator<int>::"SingleValueGenerator()"
  Catch::Generators::SingleValueGenerator<int>::"SingleValueGenerator()"
  Catch::Generators::SingleValueGenerator<int>::"SingleValueGenerator()"
  Catch::Generators::SingleValueGenerator<int>::"SingleValueGenerator()"
  Catch::Generators::acquireGeneratorTracker(
     Catch::StringRef, Catch::SourceLineInfo const&)", referenced from:
  CATCH2_INTERNAL_TEST_6() in require.o
  Catch::BinaryExpr<int, int>::streamReconstructedExpression
  (std::__1::basic_ostream<char, std::__1::char_traits<char>>&, const void(*)(())), referenced from:
  __GLOBAL__sub_I_requirecxx in require.o
  Catch::AssertionHandler::handleExpr(Catch::ITransientExpression const &), referenced from:
  CATCH2_INTERNAL_TEST_0() in require.o
  CATCH2_INTERNAL_TEST_2() in require.o
  Catch::AssertionHandler::handleThrowingCallSkipped
```

Victor Eijkhout
Function that throws:

```cpp
void even(int e) {
    if (e % 2 == 1) throw(1);
    cout << "Even number: "
         << e << "\n";
}
```

Test that it throws or not:
65.3. Catch2 unit testing

```cpp
TEST_CASE("even fun", "[3]") { 
  REQUIRE_NO_THROW(even(2));
  REQUIRE_THROWS(even(3));
}
```

Output

Undefined symbols for architecture x86_64:
  "Catch::Generators::GeneratorUntypedBase::"GeneratorUntypedBase()", referenced from:
  CATCH2_INTERNAL_TEST_6 ( ) in require.o
  Catch::Generators::Generators<int>::":
  Generators() in require.o
  Catch::Generators::Generators<int>::":
  Generators() in require.o
  Catch::Generators::Generators<int>::":
  Generators=catch::GeneratorWrapper<int>, int , int>(catch::
  Generators::GeneratorWrapper<int>&& , int&&, int&&, int&&) in require.o
  Catch::Generators::SingleValueGenerator<int >":"SingleValueGenerator()
    in require.o
  Catch::Generators::SingleValueGenerator<int >":"SingleValueGenerator()
    in require.o
  "Catch::Generators::acquireGeneratorTracker(
    Catch::StringRef, Catch::SourceLineInfo const&)"
  referenced from:
  CATCH2_INTERNAL_TEST_6 ( ) in require.o
  "Catch::StringMaker<int, void>::convert(int)"
  referenced from:
  Catch::BinaryExpr<int, int>::streamReconstructedExpression
  [std::__1::basic_ostream<
    char, std::__1::char_traits<char> >&] const in require.o
  "Catch::makeTestInvoker(
    void (*)(()))", referenced from:
    __GLOBAL__sub_I_require_cxx in require.o
  "Catch::AssertionHandler::
    handleExpr(Catch::
      ITransientExpression const &)", referenced from:
    CATCH2_INTERNAL_TEST_0 ( ) in require.o
    CATCH2_INTERNAL_TEST_2 ( ) in require.o
  "Catch::AssertionHandler::
    handleThrowingCallSkipped
    catch (AssertionHandler::...)
    handleExpression..."
Run the same test for a set of numbers:
### 65.3. Catch2 unit testing

```cpp
TEST_CASE( "even set","[4]" ) {
    int e = GENERATE( 2,4,6,8 );
    REQUIRE_NO_THROW( even(e) );
}
```

**Output**

```
[catch] requiregen:
Undefined symbols for architecture x86_64:
  "Catch::Generators::GeneratorUntypedBase::GeneratorUntypedBase()", referenced from:
  CATCH2_INTERNAL_TEST_6 in require.o
  Catch::Generators::Generators<int>::":"
  Generators() in require.o
  Catch::Generators::Generators<int>::":"
  Generators() in require.o
  Catch::Generators::Generators<int>::":"
  Generators<
  Catch::Generators::GeneratorWrapper<int>, int , int, int>(
  Catch::Generators::GeneratorWrapper<int>&&, int&&, int&&, int&&) in require.o
  Catch::Generators::SingleValueGenerator<int >:"SingleValueGenerator() in require.o
  Catch::Generators::SingleValueGenerator<int >:"SingleValueGenerator() in require.o
  "Catch::Generators::acquireGeneratorTracker(
  Catch::StringRef, Catch::SourceLineInfo const&)", referenced from:
  CATCH2_INTERNAL_TEST_6 in require.o
  "Catch::StringMaker<int, void>::convert(int)", referenced from:
  Catch::BinaryExpr<int, int>::streamReconstructedExpression
  (std::__1::basic_ostream< char, std::__1::char_traits<char> >&)
  const in require.o
  "Catch::makeTestInvoker( void (*)(())), referenced from:
  __GLOBAL__sub_I_require_cxx in require.o
  "Catch::AssertionHandler::handleExpr(Catch::
  ITransientExpression const &)", referenced from:
  CATCH2_INTERNAL_TEST_0 in require.o
  CATCH2_INTERNAL_TEST_2 in require.o
  "Catch::AssertionHandler::handleThrowingCallSkipped
  ...",
```
How is this different from using a loop? Using \textit{GENERATE} runs each value as a separate program.

Variants:

\begin{lstlisting}[language=C]
\begin{array}{l}
\textbf{int} \ i = \textit{GENERATE}( \textit{range}(1,100) ); \\
\textbf{int} \ i = \textit{GENERATE}_\text{COPY}( \textit{range}(1,n) ); \\
\end{array}
\end{lstlisting}
Chapter 66

Programming strategies

66.1 A philosophy of programming

<table>
<thead>
<tr>
<th>Code for the reader, not the writer</th>
<th>Yes, your code will be executed by the computer, but:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• You need to be able to understand your code a month or year from now.</td>
<td></td>
</tr>
<tr>
<td>• Someone else may need to understand your code.</td>
<td></td>
</tr>
<tr>
<td>• ⇒ make your code readable, not just efficient</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High level and low level</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Don’t waste time on making your code efficient, until you know that that time will actually pay off.</td>
</tr>
<tr>
<td>• Knuth: ‘premature optimization is the root of all evil’.</td>
</tr>
<tr>
<td>• ⇒ first make your code correct, then worry about efficiency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Variables, functions, objects, form a new ‘language’: code in the language of the application.</td>
</tr>
<tr>
<td>• ⇒ your code should look like it talks about the application, not about memory.</td>
</tr>
<tr>
<td>• Levels of abstraction: implementation of a language should not be visible on the use level of that language.</td>
</tr>
</tbody>
</table>

66.2 Programming: top-down versus bottom up

The exercises in chapter 52 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 52.9.

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...

66.2.1 Worked out example

Take a look at exercise 6.12. 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:

```c
// 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:

```c
// compute the sequence from 'start'
int current = starting;
while (current!=1) {
    // update current value
    length++;
}
```

6. Refine the update of the current value:

```c
// update current value
if (current%2==0)
    current /= 2;
else
    current = 3*current+1;
```

### 66.3 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.

### 66.4 Documentation

Take a look at Doxygen.
66. Programming strategies

66.5 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:

```cpp
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:

```cpp
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:

```cpp
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 67

Tiniest of introductions to algorithms and data structures

67.1 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.

67.1.1 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 67.1. Write a class that implements a stack of integers. It should have methods

```java
void push(int value);
int pop();
```

67.1.2 Linked lists

Before doing this section, make sure you study 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.
67. Tiniest of introductions to algorithms and data structures

We illustrate this in figure 67.1.

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 67.2.

67.1.2.1 Data definitions

In C++ you have a choice of pointer types. Conceptually we can say that the list object owns the first node, and each node owns the next. Therefore we use the `unique_ptr`; however, you can also use `shared_ptr` throughout, at slight overhead cost.

We declare the basic classes.

**Definition of List class**  A linked list has as its only member a pointer to a node:

```cpp
class List {
private:
  unique_ptr<Node> head(nullptr);
public:
  List() {};
}
```

Initially null for empty list.

**Definition of Node class**  A node has information fields, and a link to another node:

```cpp
class Node {
friend class List;
```
67.1. Data structures

```cpp
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 << endl; }
```

A Null pointer indicates the tail of the list.

67.1.2.2 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.

**Recursive computation of the list length**

```cpp
int recursive_length() {
    if (head==nullptr)
        return 0;
    else
        return head->listlength();
}
```

```cpp
int listlength_recursive() {
    if (!has_next()) return 1;
    else return 1+next->listlength();
}
```

The structure of an iterative version is intuitively clear: we have a pointer that goes down the list, incrementing a counter at every step. There is one complication: with C++ smart pointers, the variable that contains the current element can not be a unique pointer.

**Iterative computation of the list length**  Use a **bare pointer**, which is appropriate here because it doesn’t own the node.

```cpp
int listlength_iterative() {
    int count = 0;
    Node *current_node = head.get();
    while (current_node!=nullptr) {
        current_node = current_node->next.get(); count += 1;
    }
    return count;
}
```

(You will get a compiler error if you try to make current_node a smart pointer: you can not copy a unique pointer.)
67. Tiniest of introductions to algorithms and data structures

**Exercise 67.2.** Write a function

```cpp
bool List::contains_value(int v);
```

to test whether a value is present in the list.

Try both recursive and iterative.

### 67.1.2.3 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.

Our choice of using unique pointers dictates a certain design.

**Insert routine design** We will write functions

```cpp
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 the the value is on the current node, or gets inserted after it.

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.

![Insertion Diagram](image.png)

**Step 1: dealing with an empty list**

**Exercise 67.3.** Write a `List::length` method, so that this code gives the right output:

```cpp
List mylist;
cout << "Empty list has length: " << mylist.length_iterative() << endl;
cout << endl;
```

**Step 2: insert the first element**
Exercise 67.4. 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.

```cpp
mylist.insert(3);
cout << "After one insertion the length is: "
   << mylist.listlength_iterative() << endl;
if (mylist.contains_value(3))
   cout << "Indeed: contains 3" << endl;
else
   cout << "Hm. Should contain 3" << endl;
if (mylist.contains_value(4))
   cout << "Hm. Should not contain 4" << endl;
else
   cout << "Indeed: does not contain 4" << endl;
cout << endl;
```

Step 3: inserting an element that already exists

Exercise 67.5. 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:

```cpp
mylist.insert(3);
cout << "Inserting the same item gives length: "
   << mylist.listlength_iterative() << endl;
if (mylist.contains_value(3)) {
   cout << "Indeed: contains 3" << endl;
   auto headnode = mylist.headnode();
   cout << "head node has value " << headnode->value()
        << " and count " << headnode->count() << endl;
} else
   cout << "Hm. Should contain 3" << endl;
cout << endl;
```

Step 4: inserting an element before another

Exercise 67.6. One of the remaining cases is inserting an element that goes at the head. Update your insert method to get this to work:

```cpp
mylist.insert(2);
cout << "Inserting 2 goes at the head; now the length is: "
```
67. Tiniest of introductions to algorithms and data structures

67.7 Step 5: inserting an element at the end

Exercise 67.7. Finally, if an item goes at the end of the list:

```cpp
mylist.insert(4);
if (mylist.contains_value(4))
    cout << "Indeed: contains 4" << endl;
else
    cout << "Hm. Should contain 4" << endl;
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << endl;
else
    cout << "Hm. Should contain 3" << endl;
```

67.1.3 Trees

Before doing this section, make sure you study 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

```cpp
class Node {
private:
```
because that would recursively need infinite memory. So instead we use pointers.

```cpp
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 = init; }
        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.

```cpp
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:

```cpp
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.
# 67. Tiniest of introductions to algorithms and data structures

```c
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
}
```

## 67.2 Algorithms

This really really goes beyond this book.
- Simple ones: numerical
- Connected to a data structure: search

### 67.2.1 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 behaviour. We very briefly discuss two algorithms.

#### 67.2.1.1 Bubble sort

An array \(a\) of length \(n\) is sorted if

\[
\forall i < n-1: a_i \leq a_{i+1}.
\]

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.

```c
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 71.1.1) 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.
67.2.1.2 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 67.8. 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?

67.3 Programming techniques

67.3.1 Memoization

In section 7.5 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.

```cpp
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] << endl;
    return values[top];
}
```
67. Tiniest of introductions to algorithms and data structures
Chapter 68

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.

68.1 Loops as quantors

Quite often, algorithms can be expressed mathematically. In that case you should make your program look like the mathematics.

68.1.1 Forall-quantor

Consider a simple example: testing if a number is prime. The predicate ‘isprime’ can be expressed as:

\[ \text{isprime}(n) \equiv \forall_{2 \leq f < n} \neg \text{divides}(f, n) \]

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.

\[ \neg \text{divides}(2, n) \land \ldots \land \neg \text{divides}(n - 1, n) \]

And this sequence of ‘and’ conjunctions can be programmed:

```cpp
for (int f=2; f<n; f++)
    isprime = isprime && !isprime(f, p)
```
Now our only worry is how to initialize \texttt{isprime}. The initial value corresponds to an ‘and’ conjunction over an empty set, which is true, so:

\begin{verbatim}
bool isprime(true);
for (int f=2; f<n; f++)
isprime = isprime \&\& not divides(f,p)
\end{verbatim}

### 68.1.2 Thereis-quantor

What if we had expressed primeness as:

\[
isprime(n) \equiv -\exists_{2 \leq f < n} \text{ divides}(f,n)
\]

To get a pure quantor, and not a negated one, we write:

\[
isnotprime(n) \equiv \exists_{2 \leq f < n} \text{ divides}(f,n)
\]

Spelling out the exists-quantor as

\[
isnotprime(n) \equiv \text{ divides}(2, n) \cup \ldots \cup \text{ divides}(n - 1, n)
\]

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.

\begin{verbatim}
for (int f=2; f<n; f++)
isnotprime = isnotprime \lor divides(f,p)
bool isprime = not isnotprime;
\end{verbatim}

Again we take care to initialize the reduction variable correctly: applying \(\exists_{s \in S} P(s)\) over an empty set \(S\) is \texttt{false}: bool isnotprime=false;

\begin{verbatim}
for (int f=2; f<n; f++)
isnotprime = isnotprime \lor divides(f,p)
bool isprime = not isnotprime;
\end{verbatim}

### 68.2 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:
\[
P(\ell_k, m_k, k) = \begin{cases} 
\ell_k < k \\
\land c(\ell_k) = m_k \text{(only if } k > 0) \\
\land \forall \ell < k : c(\ell) \leq m_k
\end{cases}
\]
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:
\[
\begin{align*}
&k = 0; \\
&\{ P(\ell_k, m_k, k) \} \\
&\text{while} \ ( \ k < N \ ) \ \{ \\
&\quad \{ P(\ell_k, m_k, k) \} \\
&\quad \text{update;} \\
&\quad \{ P(\ell_{k+1}, m_{k+1}, k + 1) \} \\
&\quad k = k + 1; \\
&\}
\end{align*}
\]
The update has to extend the predicate from \( k \) to \( k + 1 \). Let us consider the parts of it.

We need to establish
\[
\forall \ell < k + 1 : c(\ell) \leq m_{k+1}
\]
We split the range \( \ell < k + 1 \) into \( \ell < k \) and \( \ell = k \):
- the first part
  \[
  \forall \ell < k : c(\ell) \leq m_{k+1}
  \]
  is true if \( m_{k+1} \geq m_k \);
- the part
  \[
  \ell = k : c(\ell) \leq m_{k+1}
  \]
  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:
68. Provably correct programs

- 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 cannot satisfy \(m_{k+1} = c(\ell_k)\) for any \(k\).)
- If \(c(k) \leq m_k\), we need to set \(m_{k+1} = m_k\). Again, \(m_{k+1} > m_k\) cannot be satisfied by any \(\ell_{k+1}\), so we conclude \(m_{k+1} = m_k\).

68.3 Flame

Dijkstra quote, part 1

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)

Dijkstra quote, part 2

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.

Matrix-vector product

\[ y = Ax \]

Partitioned:

\[
\begin{pmatrix}
    y_T \\
    y_B
\end{pmatrix}
= \begin{pmatrix}
    A_T \\
    A_B
\end{pmatrix} \begin{pmatrix}
    x
\end{pmatrix}
\]

Two equations:

\[
\begin{aligned}
    y_T &= A_T x \\
    y_B &= A_B x
\end{aligned}
\]

Inductive construction

\[
\begin{pmatrix}
    y_T \\
    y_B
\end{pmatrix}
= \begin{pmatrix}
    A_T \\
    A_B
\end{pmatrix} \begin{pmatrix}
    x
\end{pmatrix}
\]

Assume only equation

\[ y_T = A_T x \]

is satisfied, and grow the \(T\) block.
Algorithm outline

\[
\begin{pmatrix}
  y_T \\
  y_B
\end{pmatrix}
= 
\begin{pmatrix}
  A_T \\
  A_B
\end{pmatrix}
(x)
\]

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.

Inductive step  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

\[
\begin{pmatrix}
  y_T \\
  y_B
\end{pmatrix}
= 
\begin{pmatrix}
  A_T \\
  A_B
\end{pmatrix}
(x)
\]

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

\[
\begin{pmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_3
\end{pmatrix}
= \begin{pmatrix}
  A_1 \\
  A_2 \\
  \vdots \\
  A_3
\end{pmatrix}
(x)
\]

so

\[
\begin{cases}
  y_1 = A_1 x & \text{we had this} \\
  y_2 = A_2 x & \text{we need this}
\end{cases}
\]

**Resulting 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

**Matrix-vector product, the other way around**

\( y = Ax \)

Partitioned:

\[
(y) = (A_L \quad A_R) \begin{pmatrix}
  x_T \\
  x_B
\end{pmatrix}
\]

Equation:

\[
\begin{cases}
y = A_L x_T + A_R x_B
\end{cases}
\]

**Inductive construction**

\[
(y) = (A_L \quad A_R) \begin{pmatrix}
  x_T \\
  x_B
\end{pmatrix}
\]

Assume

\( y = A_L x_T \)

is constructed, and grow the \( T \) block.
**Inductive step** Before

\[
(y) = (A_L \ A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}
\]

split:

\[
(y) = \begin{pmatrix} A_1 & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_3 \end{pmatrix}
\]

Then the update step, and

After

\[
(y) = \begin{pmatrix} A_1 & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_3 \end{pmatrix}
\]

and unsplit

\[
(y) = (A_L \ A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}
\]

**Derivation of the update** Before the update:

\[
(y) = \begin{pmatrix} A_1 & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_3 \end{pmatrix}
\]

so

\[
y = A_1 x_1
\]

is true

Then the update step, and

After

\[
(y) = \begin{pmatrix} A_1 & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_3 \end{pmatrix}
\]

so

\[
y = A_1 x_1 + A_2 x_2
\]

in other words, we need

\[
y \leftarrow y + A_2 x_2
\]
Resulting algorithm

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

Two algorithms

for $r = 1, m$
  $y_r = A_{r,s} x_s$

$y \leftarrow 0$

for $c = 1, n$
  $y \leftarrow y + A_{s,c} x_c$
Chapter 69

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.

69.1 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.
69. Unit testing and Test-Driven Development

69.2 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](https://github.com/catchorg).

69.2.1 File structure

Let’s assume you have a file structure with

- a very short main program, and
- a library file that has all functions use by the main.

In order to test the functions, you supply another main, which contains only unit tests; This is illustrated in figure 69.1.

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.

**Unittest driver file**  The framework supplies its own main:

```cpp
#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
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

```cpp
#include "catch.hpp"
```

Note: as of September 2021 this requires the development version of the repository, not any 2.x release.

### 69.2.2 Compilation

The setup suggested above requires you to add compile and link flags to your setup. This is system-dependent.

**Compiling the tester at TACC** One-line solution:

```
icpc -o tester test_main.cxx \
   -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
   -lCatch2Main -lCatch2
```

**Compile and link options at TACC** Variables for a Makefile:

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2Main -lCatch2
```

### 69.2.3 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

```cpp
#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.

```cpp
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.
Correctness through ‘require’ clause Tests go in tester.cxx:

```c++
TEST_CASE( "test that f always returns positive" ) {
    for (int n=0; n<1000; n++)
        REQUIRE( f(n)>0 );
}
```

- `TEST_CASE` acts like independent program.
- `REQUIRE` is like `assert` but more sophisticated
- Can contain (multiple) tests for correctness.

Exercise 69.1.

1. Write a function
   ```c++
   double f(int n) { /* .... */ }
   ```
   with values in the range (0,1).
2. Write a unit test for this.
   You can base this off the file `tdd.F90`.

Tests labels: catch-approx Boolean:

```c++
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Integer:

```c++
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!==0 );
```

Beware floating point:

```c++
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!==Catch::Approx(1.0) );
```

In general exact tests don’t work.

Output for failing tests Run the tester:

```
--------------------------------
test the increment function
cxx:25

--------------------------------
test.cxx:29: FAILED:
   REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
   1 == 2
```

```c++
REQUIRE( test cases: 1 | 1 failed
```
For failing tests you get a diagnostic message, but in the above case it would print 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.

**Diagnostic information for failing tests**  INFO: print out information at a failing test

```c
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.

**Test for exceptions**  Suppose function $g(n)$

- succeeds for input $n > 0$
- fails for input $n \leq 0$
  throws exception

```c
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 teardown, to use terms that you sometimes come across in unit testing. Catch2 supports this: you can make ‘sections’ for part in between setup and teardown.

**Tests with code in common**  Use `SECTION` if tests have intro/outtro in common:

```c
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)
69. Unit testing and Test-Driven Development

69.3 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

```cpp
using quadratic = tuple<double, double, double>;
```

When needed, you can unpack this again with

```cpp
auto [a, b, c] = coefficients;
```

(\text{Can you think of an} \text{ assert} \text{ statement here that might be useful?})

\textbf{Exercise 69.2.} \textbf{Write a function}

```cpp
double discriminant( quadratic coefficients );
```

that computes \( b^2 - 4ac \), and test:

```cpp
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*1.5 ) ) == Catch::Approx(-.01) );
}
```

It may be illustrative to see what happens if you leave out the approximate equality test:

```cpp
REQUIRE( discriminant( make_tuple(.1, .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 \( D = 0 \).

\textbf{Exercise 69.3.} \textbf{Write a function}

```cpp
bool discriminant_zero( quadratic coefficients );
```

that passes the test

```cpp
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:

```cpp
a = 2; b = 4; c = 2;
a = 2; b = sqrt(40); c = 5; // !!!!
a = 3; b = 0; c = 0.;
```

Introduction to Scientific Programming
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 69.4.** 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 69.5.** Write the function simple_root that returns the single root. For confirmation, test

```cpp
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 69.6.** Write a function that returns the two roots as a std::pair:

```cpp
pair<double,double> double_root( quadratic coefficients );
```

Test:

```cpp
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 69.7.** Write a function

```cpp
variant< bool, double, pair<double,double> > compute_roots( quadratic coefficients );
```

Test:

```cpp
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 );
}
```
69.4 Eight queens example

See 55.3.
Chapter 70

Debugging with gdb

We are going to use the following simple program:

```c
void say(int n) {
    cout << "hello world " << n << endl;
}

int main() {
    int ii;
    for (int i=0; i<10; i++) {
        ii = i*i;
        ii++;
        say(ii);
    }
    return 0;
}
```

70.1 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
```
70. Debugging with gdb

hello world 17
hello world 26
hello world 37
hello world 50
hello world 65
hello world 82

[Inferior 1 (process 30981) exited normally]

70.1.1 Run with commandline arguments

This program is self-contained, but if you had a program that takes commandline arguments:

```bash
./myprogram 25
```
you can supply those in gdb:

```bash
(gdb) run 25
```

70.1.2 Source listing and proper compilation

Inside gdb, you can get a source listing with the `list` command.

Let's try our program again:

```bash
[] 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:

```bash
[] 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.)

70.2 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:

```c
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)

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;
70.3 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 \( \text{i} = i \times i; \)
25 (gdb) print i
26 \$1 = 4

You can even let expressions be evaluated with local variables:
27 (gdb) print 2*i
28 \$2 = 8

You can combine this looking at values with breakpoints. Say you want to know when the variable \( \text{ii} \) gets more than 40:

29 (gdb) break 26 if ii>40
30 Breakpoint 1 at 0x4009cd: file hello.cxx, line 26.
31 (gdb) run
32 Starting program: /home/eijkhout/intro-programming-private/code/gdb/hello
33 hello world 1
34 hello world 2
35 hello world 5
36 hello world 10
37 hello world 17
38 hello world 26
39 hello world 37
40 Breakpoint 1, main () at hello.cxx:26
41 26 say(ii);
42 Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x86_64
43 (gdb) print i
44 \$1 = 7

70.4 A NaN example

The following program:

17 float root(float n)
18 {
19 float r;
20 float nl = n-1.1;
21 r = sqrt(nl);
22 return r;
23 }
24 int main() {
25 float x=9,y;
70. Debugging with gdb

prints some numbers that are ‘not-a-number’:

```c++
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
root: -nan
root: -nan
root: -nan
```

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:

```
    float n1 = n-1.1;
    r = sqrt(n1);
```

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.

### 70.5 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:

```c
#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
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 71

Complexity

71.1 Order of complexity

71.1.1 Time complexity

Exercise 71.1. 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 71.2. How many operations, as a function of \( n \), are performed in these two solutions?

71.1.2 Space complexity

Exercise 71.3. 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 71.4. How much space do the two solutions require?
71. Complexity
PART VII

INDEX AND SUCH
Chapter 72

General index of terms

```plaintext
#ifndef, 234
#define, 232, 453
#error, 235
 ifdef, 234, 235
 ifndef, 233, 234
 include, 231
 once, 235
 pack, 235
 typedef, 233
 abstraction, 75
 accessor, 108
 Amazon
delivery truck, 443
prime, 443, 449
Amazon Prime, 32
Apple, 29
argument, 80
 actual, 316
default, 89
dummy, 316
keyword, 317
optional, 318
positional, 317
array, 127
 associative, 248
assumed-shape, 346
automatic, 339, 346
initialization, 340
operations, semantics of, 341
rank, 342
static, 339
ASCII, 375
assertion, 517
assignment, 46

asterisk
 in Fortran formatted I/O, 322
autotools, see GNU, autotools

bisection, 76, 413
Boost, 155
bottom-up, 481
Boyer-Moore, 462
breakpoint, 513
bubble sort, 147, 492
bug, 22
bus error, 130

C
 parameter passing, 205–209
pointer, 201–210, 280
preprocessor, 289
string, 156, 280
C preprocessor, see preprocessor
C++, 273
 C++03, 274
 C++11, 155, 167, 218, 274
 C++14, 218, 274–275
 C++17, 43, 56, 62, 85, 156, 218, 250, 252,
 259, 275
 C++20, 54, 147, 157, 182, 218, 225, 240, 246,
 275, 330
Core Guidelines, 484
 standard, 146
Caesar cypher, 154
calendars, 275
call-back, 271
callable, 261
calling environment, 84, 185
capture, 167, 169
```
INDEX

case sensitive, 45
cast, 53, 268
    lexical, 155
Catch2, 504
charconv, 156
class, 99, 99
    abstract, 114
    base, 112
    derived, 111
    iterable, 265
closure, 96, 167
Cmake, 470, 471, 505
code
    duplication, 78
    maintainance, 483
code reuse, 78
Collatz conjecture, 74
column-major, 343, 364
compilation
    separate, 221, 330, 430
    compile-time constant, 339
compiler, 30, 41
    and preprocessor, 231
    one pass, 80
compiling, 30
complex numbers, 47, 247
concept, 240
concepts, 275
conditional, 57, 377
configure, 470
connected components, see graph, connected
const
    reference, 211
    construct name, 305
constructor, 101, 200, 281
    copy, 116, 212
    default, 101, 106
    defaulted, 107
    delegating, 115, 422
    range, 279
container, 248
contains
    for class functions, 333
    in modules, 330
continuation character, 291
copy
depth, 338
    shallow, 338
copy constructor
    see constructor, copy, 116
coroutines, 275
covid-19, 429
data model, 54
data race, 261
datatype, 45, 374
debugger, 246
DEC PDP-11, 54
define, see #pragma define, see #pragma
define
definition vs use, 93
dependence, 341
dereference, 202
dereferencing, 143
dereferencing a
    nullptr, 199
destructor, 97, 117
    at end of scope, 96
    in Fortran, see final procedure
Dijkstra
    Edsger, 503
do
    concurrent, 74
do concurrent, 351
do loop
    implicit, 364
    and array initialization, 340
    implied, 306
dynamic
    programming, 441
ebola, 429
ECMA, 250
efficiency gap, 442
Eigen, 141
eight queens, 252, 419
emacs, 29, 43, 290
encoding
    extendible, 156
error
    compile-time, 44
    run-time, 44
    syntax, 44
error, see `#pragma error`

exception, 130, 243
- catch, 244
- catching, 243
- throwing, 243
executable, 30, 42, 331
exponent part, 48
expression, 46
extent
- of array dimension, 345

Fasta, 461
Fastq, 462
file
- binary, 41, 44
- executable, 221
- handle, 117
- include, 99
- object, 221, 331
- source, 41
final, 335
floating point error, 303
floating point number, 48
fmtlib, 279
for
- indexed, 132
- range-based, 131
Fortran
- 90, 289
- case ignores, 290
- comments, 291
- Fortran2008, 331
forward declaration, 220
- of classes, 95
- of functions, 95
function, 75, 313, 314
- argument, 79
- arguments, 79
- body, 79
- call, 76, 77
- defines scope, 80
- definition, 76
- header, 80
- parameter, 79
- parameters, 79
- pointwise application to array, 388
- prototype, 80
- result type, 79
function try block, 245
functional programming, 81, 185
functor, 122
gdb
- `break`, 513
- `cont`, 513
- `delete`, 513
- `list`, 512
- `next`, 514
- `run`, 511
- `run with commandline arguments`, 512
- `step`, 514
gerrymandering, 437
glyph, 156
GNU, 41
- autotools, 470
- Goldbach conjecture, 402
Google, 433
- developer documentation style guide, 397
graph
- connected, 434
- diameter, 435
greedy search, see search, greedy
has-a relation, 109
hdf5, 363
header, 430
- header file, 219, 221, 231
- and global variables, 224
- treatment by preprocessor, 224
  vs modules, 275
header-only, 224
heap, 146, 146
- fragmentation, 146
hexadecimal, 201
Holmes
- Sherlock, 154
homebrew, 29
Horner’s rule, 415
host association, 310
I/O
- formatted, 363
- list-directed, 363
unformatted, 363
if
   ternary, 62, 378
ifdef, see #pragma ifdef
ifndef, see #pragma ifndef
include
   path, 232
include, see #pragma include
incubation period, 429
index
   section, 340
inheritance, 111
initialization
   variable, 45
initializer
   in conditional, 62
      list, 128, 131
      member, 103, 245, 410
initializer list, 104
inline, 318
is-a relation, 111
ISO
   bindings, 294
iteration
   of a loop, see loop, iteration
iterator, 143, 175, 274
julia
   compared to python, 374
keywords, 44
kind selector, 293
label, 323
lambda, see closure, 271
   expression, 167
      generic, 172
lazy evaluation, 182
lazy execution, 182
lexicographic ordering, 69
library
   software, 469
linear regression, 466
linker, 221, 331
Linux, 29
list
   linked, 485–490
in Fortran, 359–361
   single-linked, 183
locality, 341
logging, 163
loop, 65
   body, 65
      counter, 65
      for, 65
      header, 65
      index, 66
      inner, 69, 383
      iteration, 65
      nest, 69, 383
      outer, 69, 383
      variable, 66
      while, 65
lvalue, 271
macports, 29
Make, 221, 232, 469
makefile, 222, 232, 430
Manhattan distance, 443
mantissa, 48
math
   functions (in C++), 88
max, 88
member
   data, 99
      function, 99
      initializer, see initializer, member, 139
      initializer list, 104
memoization, 441, 493
memory
   bottleneck, 454
      leak, 117, 146, 207, 207, 346
memory leak, 195
memory leaking, 209
method, 104
   abstract, 114
   overriding, 113
methods, see member, function
Microsoft
   Windows, 29
   Word, 22
module, 225
   sub, 331
modules, 275
move semantics, 272
multiple dispatch, 385, 392
multiplication
   Egyptian, 88
namespace, 227
Newton’s method, 83
NP-hard, 447
NULL, 199
null terminator, 280
object, 99
   state of, 105
object file, see file, object
once, see #pragma once
OpenFrameworks, 273
operator
   arithmetic, 181
   bitwise, 59
   comparison, 59
   logic, 58
   overloading, 121
      and copies, 273
      of parentheses, 122
   precedence, 59
   short-circuit, 378
   shortcut, 52, 376
   spaceship, 275
   unary star, 176
opt2, 447
output
   binary, 365
   raw, 365
   unformatted, 365
pack, see #pragma pack
package manager, 29, 470
Pagerank, 433
parameter, 80, see also function, parameter
   actual, 80
   formal, 80, 317
   input, 84
   output, 84, 250
   pass by value, 82
   passing, 81
      by reference, 185, 200, 281
by value, 185
passing by reference, 81, 84
   in C, 84
   passing by value, 81
throughput, 84
Pascal’s triangle, 147
pass by reference, see parameter, passing by reference
pass by value, see parameter, passing by value
path
   Hamiltonian, 462
PETSc, 246
pipe, 182
pointer, 123
   and heap, 146
   arithmetic, 143, 205
   bare, 197, 281, 487
   dereference, 176
   dereferencing, 355
   null, 199, 485
   opaque
      in C++, 199
   smart, 146, 191
   unique, 197
   void, 270
   weak, 195, 198
Poisson
distribution, 257
polymorphism, 121, 385
   of constructors, 107
pop, 485
preprocessor
   and header files, 224
   conditionals, 234
   macro
      parametrized, 233
      macros, 232–234
procedure, 311
   final, 335
   internal, 310
procedures
   internal, 318
   module, 318
program
   statements, 42
programming
dynamic, 440
parallel, 74
prototype, 220
    of class method, 119
punch cards, 289
push, 485
putty, 29
python, 26
quicksort, 493
radix point, 48
RAII, 280
random
    seed, 368
random number
    generator, 256, 258
        Fortran, 368
    seed, 259
range-based for loop, see for, range-based
    ranges, 275
recurrence, 352
recursion, see function, recursive
    depth, 90
    mutual, 88
reduction, 180
reference, 83, 185
    argument, 200, 281
    const, 137, 185, 189
        to class member, 186
    to class member, 186
reference count, 197
regression test
    setesting, regression, 503
reserved words, 45
return, 79
    makes copy, 189
root finding, 413
roundoff error analysis, 55
runtime error, 241
rvalue, 271
    reference, 273
scope, 78, 93
    and stack, 146
    dynamic, 96
in conditional branches, 61
lexical, 93, 96
of function body, 80
search
    greedy, 446, 447
segmentation fault, 130
shape
    of array, 345
shell
    inspect return code, 55
short circuit evaluation, 61
side-effects, 213
significand, 48
Single Source Shortest Path, 435
SIR model, 428
smartphone, 22
smatch, 249
software library, see library, software
source
    format
        fixed, 289
        free, 289
source code, 30
stack, 90, 97, 146, 146, 346, 485
    overflow, 90, 146, 346
Standard Template Library, 247
state, 106
statement functions, 318
stream, 162
string, 152
    concatenation, 152
    null-terminated, 156
raw literal, 155, 250
    size, 152
    stream, 155
structured binding, 251
subprogram, see function
superuser, 470
symbol
    debugging, 512
    table, 512
syntax
    error, 241
system test
    setesting, system, 503
templates
    and separate compilation, 223
Terminal, 29
testing
    regression, 503
    system, 503
    unit, 503
text
    formatting, 275
time zones, 275
timer
    resolution, 368
top-down, 481	
tuple, 250
    denotation, 250
type
    deduction, 131
    derived, 325
    nullable, 252
    return
    trailing, 264
    signature
    , 385
typedef, see #pragma typedef

Unicode, 156, 374, 375
unit, 364
unit test
    see testing, unit, 503
Unix, 29
UTF8, 156

values
    boolean, 49
variable, 44, 374
    assignment, 45
    declaration, 45, 45
    global, 224
        in header file, 224
    initialization, 49
    lifetime, 94
    numerical, 47
    shadowing, 94
    static, 96, 309
vector, 227, 403
    bounds checking, 130
    index, 129

initialization, 130
methods, 134
subscript, 129
subvector, 176
vi, 29
Virtualbox, 29
Visual Studio, 29
VMware, 29
X windows, 29
Xcode, 29
XSEDE, 397
Chapter 73

Index of C++ keywords

__FILE__, 246
__LINE__, 246
__cplusplus, 274
abort, 242
abs, 88
accumulate, 180
algorithm, 88, 123, 172, 179
all_of, 179
any, 254
any_cast, 254
any_of, 172, 179, 180, 267
array, 146
assert, 242, 508
at, 129, 130, 133, 134, 147
auto, 131, 274
auto_ptr, 274
back, 134
bad_alloc, 245
bad_exception, 245
basic_ios, 165
begin, 175, 175, 265
break, 70
cerr, 163, 279
cchar, 151
cin, 47, 50, 164, 279
clang++, 42
cmath, 51, 88, 169
complex, 247
const, 211, 213, 218
const_cast, 217, 270
constexpr, 218, 218, 274, 275
continue, 71
copy, 177
cout, 50, 279
data, 147
decltype, 265
delete, 275
denorm_min, 255
dereference, 175
distance, 178
divides, 181
dynamic_cast, 268
default, 175, 175, 265
denorm_min, 255
dereference, 175
distance, 178
divides, 181
dynamic_cast, 268
default, 175, 175, 265
denorm_min, 255
dereference, 175
distance, 178
divides, 181
dynamic_cast, 268
default, 175, 175, 265
denorm_min, 255
dereference, 175
distance, 178
divides, 181
dynamic_cast, 268
friend, 115
from_chars, 156
front, 134
function, 168
functional, 181
g++, 41, 42
gdb, 246
get, 196
get_if, 252
ggetline, 164, 165
has_value, 252
icpc, 41, 42
if, 62
Inf, 55, 256
insert, 135
insert, 177
int, 54
int, 48
intptr_t, 270
is_eof, 165
is_open, 165
isnan, 245, 256
iterator, 175
itoa, 155
join, 261
limits, 255
limits.h, 255
long, 54
long, 54
long long, 54
longjmp, 246
lowest, 255
main, 54
malloc, 146, 200, 204, 208, 209, 279, 281
map, 248
max, 255
max_element, 178, 181
MAX_INT, 255
min, 255
minus, 181
modulus, 181
monostate, 253
multiplies, 181
mutable, 217
mutable, 172
NaN, 55, 256
NDEBUG, 242
negate, 181
new, 139, 140, 195, 197, 206, 209, 275
noexcept, 245
none_of, 179
NULL, 156
nullopt, 252
nullptr, 84, 199, 200, 268, 485
nullptr_t, 199
numbers, 275
numeric, 180
numeric_limits, 255
open, 164
optional, 252
out_of_range, 243
override, 113
pair, 509
pair, 250
plus, 181
pop_back, 135
printf, 157, 279
private, 102, 105, 108
protected, 112
public, 102, 108
push_back, 134, 135
rand, 258
RAND_MAX, 258
range, 182
ranges, 182
rbegin, 176
regex, 249
regex_match, 249
regex_search, 249
reinterpret_cast, 268, 270
rend, 176
return, 55
scanf, 279

Introduction to Scientific Programming
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>setjmp</td>
<td>246</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>281, 486</td>
</tr>
<tr>
<td>short</td>
<td>54, 54</td>
</tr>
<tr>
<td>size</td>
<td>133, 134, 147</td>
</tr>
<tr>
<td>size_t</td>
<td>268</td>
</tr>
<tr>
<td>sizeof</td>
<td>205, 208</td>
</tr>
<tr>
<td>sort</td>
<td>181</td>
</tr>
<tr>
<td>source_location</td>
<td>246</td>
</tr>
<tr>
<td>span</td>
<td>147, 147, 275, 279, 452</td>
</tr>
<tr>
<td>sprintf</td>
<td>155</td>
</tr>
<tr>
<td>srand</td>
<td>259</td>
</tr>
<tr>
<td>static</td>
<td>319</td>
</tr>
<tr>
<td>static_assert</td>
<td>242</td>
</tr>
<tr>
<td>static_cast</td>
<td>268, 270</td>
</tr>
<tr>
<td>std::any</td>
<td>199</td>
</tr>
<tr>
<td>string</td>
<td>280</td>
</tr>
<tr>
<td>stringstream</td>
<td>123, 155</td>
</tr>
<tr>
<td>struct</td>
<td>250, 401</td>
</tr>
<tr>
<td>swap</td>
<td>273</td>
</tr>
<tr>
<td>switch</td>
<td>60, 62</td>
</tr>
<tr>
<td>this</td>
<td>123, 197</td>
</tr>
<tr>
<td>to_chars</td>
<td>156</td>
</tr>
<tr>
<td>to_string</td>
<td>155</td>
</tr>
<tr>
<td>to_vector</td>
<td>182</td>
</tr>
<tr>
<td>transform</td>
<td>182</td>
</tr>
<tr>
<td>true</td>
<td>49, 51</td>
</tr>
<tr>
<td>tuple</td>
<td>250</td>
</tr>
<tr>
<td>union</td>
<td>252</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>197, 281, 486</td>
</tr>
<tr>
<td>using</td>
<td>44, 233</td>
</tr>
<tr>
<td>utility</td>
<td>54</td>
</tr>
<tr>
<td>valgrind</td>
<td>246</td>
</tr>
<tr>
<td>value</td>
<td>252</td>
</tr>
<tr>
<td>value_or</td>
<td>252</td>
</tr>
<tr>
<td>variant</td>
<td>85, 252, 508, 509</td>
</tr>
<tr>
<td>vector</td>
<td>127, 131, 133, 146, 176, 182, 279</td>
</tr>
<tr>
<td>view</td>
<td>182</td>
</tr>
<tr>
<td>visit</td>
<td>254</td>
</tr>
<tr>
<td>void</td>
<td>79, 80</td>
</tr>
<tr>
<td>weak_ptr</td>
<td>198</td>
</tr>
<tr>
<td>while</td>
<td>72, 133</td>
</tr>
</tbody>
</table>
Chapter 74

Index of Fortran keywords

`.AND., 300
.and., 300
.eq., 300
.false., 300
.ge., 300
.gt., 300
.le., 300
.lt., 300
.ne., 300
.or., 300
.true., 300

Abs, 348
advance, 361
aimag, 297
AIMIG, 296
All, 348
all, 349
allocatable, 292
allocate, 346, 361
Any, 348
any, 349
Associated, 358

bit_size, 294

c_sizeof, 294
call, 312, 314
case, 300
CHAR, 321
Character, 292
class, 334
Close, 363, 365
CMPLX, 296, 297
Common, 310

Complex, 292, 297
concurrent, 351
CONJG, 296
Contains, 310, 311
contains, 314, 318, 330, 333, 335, 367
continue, 306
Count, 348
Cshift, 348
data, 364
DBLE, 296
deallocate, 346
DIM, 348
dimension, 292, 339
dimension (:), 345
do, 303, 305, 309
DOT_PRODUCT, 348
Dot_Product, 348
end, 290
end do, 305, 309
End Program, 290
entry, 318
exit, 305
external, 367
F90, 289
FLOAT, 296
for all, 351
forall, 350
Format, 363
format, 364
Function, 313, 346
function, 314
gfortran, 290
go to, 306
huge, 293, 294
ICHAR, 321
if, 299
ifort, 290
implicit none, 329, 330
in, 316
INDEX, 321
inout, 316
INT, 296
Integer, 292
integer, 291
intent, 292
Interface, 311
interface, 314, 367, 367
intrinsic, 318
iso_c_binding, 294
kind, 293
Lbound, 344
LEN, 321
len, 321
Logical, 292, 300
logical, 291
MASK, 348
MATMUL, 349
MatMul, 348
MaxLoc, 348
MaxVal, 348
MinLoc, 348
MinVal, 348
Module, 310, 311, 335
module, 326, 329
NINT, 296
Nullify, 358
Open, 363
open, 365
Optional, 318
optional, 318
out, 316
parameter, 292, 292
pointer, 355
Present, 318
Print, 363
print, 363
private, 331, 336
procedure, 335
Product, 348
Program, 290, 311
protected, 331
public, 331
random_number, 368
random_seed, 368
Read, 363
REAL, 296
Real, 292
real, 297
real(4), 291
real(8), 291, 294
Recursive, 313
recursive, 313
RESHAPE, 345
reshape, 342
Result, 315, 346
result, 313
return, 312, 313
Save, 310
save, 318, 319
Select, 301
select, 300
selected_int_kind, 293
selected_real_kind, 293
size, 345
SNGL, 296
SPREAD, 345
stat=ierror, 346
stop, 290
storage_size, 294
Subroutine, 346
subroutine, 313
Sum, 348
system_clock, 368
target, 356
Transpose, 348
TRIM, 321
trim, 321
Type, 292, 335
type, 325, 333, 334, 336

Ubound, 344
use, 314, 329, 330, 336

where, 349
while, 305
Write, 296, 361, 363
write, 364
Chapter 75

Bibliography


