Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

2018-2020, last formatted October 11, 2020

Contents

I Introduction 17
1 **Introduction** 19
1.1 *Programming and computational thinking* 19
1.1.1 History 19
1.1.2 Is programming science, art, or craft? 20
1.1.3 Computational thinking 21
1.1.4 Hardware 24
1.1.5 Algorithms 24
1.2 *About the choice of language* 24
2 **Warming up** 27
2.1 *Programming environment* 27
2.1.1 Language support in your editor 27
2.2 *Compiling* 28
2.3 *Your environment* 28
3 **Teachers guide** 29
3.1 *Justification* 29
3.2 *Timeline for a C++/F03 course* 30
3.2.1 Project-based teaching 31
3.2.2 Choice: Fortran or advanced topics 31
3.3 *Grading guide* 31
3.3.1 Code style 31

II C++ 33
4 **Basic elements of C++** 35
4.1 *From the ground up: Compiling C++* 35
4.1.1 A quick word about unix commands 36
4.1.2 C++ is a moving target 37
4.2 *Statements* 37
4.3 *Variables* 39
4.3.1 Variable declarations 39
4.3.2 Assignments 40
4.3.3 Terminal input 41
4.3.4 Datatypes 41
4.3.5 Initialization 44

4.4 *Input/Output, or I/O as we say* 44
4.5 *Expressions* 44
4.5.1 Numerical expressions 45
4.5.2 Truth values 45
4.5.3 Type conversions 46
4.6 *Advanced topics* 47
4.6.1 Library functions 47
4.6.2 Number values and undefined values 47
4.6.3 Constants 48
4.6.4 Numerical precision 48
4.7 *Review questions* 48
5 **Conditionals** 51
5.1 *Conditionals* 51
5.2 *Operators* 52
5.3 *Switch statement* 54
5.4 *Scopes* 54
5.5 *Advanced topics* 55
5.5.1 Short-circuit operators 55
5.5.2 Ternary if 55
5.5.3 Initializer 56
5.6 *Review questions* 56
6 **Looping** 59
6.1 *The 'for' loop* 59
6.1.1 Loop syntax 62
6.2 *Looping until* 62
6.3 *Advanced topics* 66
6.3.1 Parallelism 66
6.4 *Exercises* 66
7 **Functions** 69
7.1 *Function definition and call* 69
7.1.1 Why use functions? 71
7.2 *Anatomy of a function definition and call* 72
7.2.1 Another option for defining functions 73
7.3 *Void functions* 73
7.4 *Parameter passing* 74
7.4.1 Pass by value 74
7.4.2 Pass by reference 76
7.5 *Recursive functions* 78
7.6 *Advanced function topics* 80
7.6.1 Default arguments 80
7.6.2 Polymorphic functions 80
7.6.3 Stack overflow 80
7.7 *Library functions* 81
7.7.1 Random function 81
7.8 *Review questions* 81
8 **Scope** 83

8.1 *Scope rules* 83
8.1.1 Lexical scope 83
8.1.2 Shadowing 83
8.1.3 Lifetime versus reachability 84
8.1.4 Scope subtleties 85
8.2 *Static variables* 86
8.3 *Scope and memory* 86
8.4 *Review questions* 87
9 **Structures** 89
9.1 *Why structures?* 89
9.2 *The basics of structures* 89
9.3 *Structures and functions* 91
10 **Classes and objects** 95
10.1 *What is an object?* 95
10.1.1 Constructor 96
10.1.2 Methods, introduction 97
10.1.3 Initialization 97
10.1.4 Member access 98
10.1.5 Methods 99
10.1.6 Default constructor 101
10.1.7 Accessors 101
10.1.8 Examples 102
10.2 *Inclusion relations between classes* 103
10.2.1 Literal and figurative has-a 104
10.3 *Inheritance* 105
10.3.1 Methods of base and derived classes 106
10.3.2 Virtual methods 107
10.3.3 Advanced topics in inheritance 108
10.4 *Advanced topics* 108
10.4.1 Returning by reference 108
10.4.2 Accessor functions 109
10.4.3 Accessability 110
10.4.4 Polymorphism 110
10.4.5 Operator overloading 110
10.4.6 Functors 111
10.4.7 Copy constructor 112
10.4.8 Destructor 113
10.4.9 ‘this’ pointer 114
10.4.10 Static members 114
10.5 *Review question* 115
11 **Arrays** 117
11.1 *Introduction* 117
11.1.1 Vector creation 117
11.1.2 Element access 118
11.1.3 Initialization 119
11.2 *Going over all array elements* 120

11.2.1 Ranging over a vector *120*
11.2.2 Ranging over the indices *120*
11.2.3 Ranging by reference *121*
11.3 *Vector are a class* *122*
11.3.1 Vector methods *122*
11.3.2 Vectors are dynamic *123*
11.4 *Vectors and functions* *124*
11.4.1 Pass vector to function *124*
11.4.2 Vector as function return *124*
11.5 *Vectors in classes* *126*
11.5.1 Timing *126*
11.6 *Wrapping a vector in an object* *127*
11.7 *Multi-dimensional cases* *128*
11.7.1 Matrix as vector of vectors *128*
11.7.2 A better matrix class *129*
11.8 *Advanced topics* *129*
11.8.1 Iterators *129*
11.8.2 Old-style arrays *130*
11.8.3 Stack and heap allocation *132*
11.8.4 The Array class *133*
11.8.5 Span *133*
11.9 *Exercises* *134*
12 **Strings** *137*
12.1 *Characters* *137*
12.2 *Basic string stuff* *137*
12.3 *String streams* *140*
12.4 *Advanced topics* *140*
12.4.1 Conversation to/from string *140*
12.4.2 Unicode *141*
12.5 *C strings* *141*
13 **Input/output** *143*
13.1 *Formatted output* *143*
13.1.1 Floating point output *145*
13.1.2 Saving and restoring settings *146*
13.2 *File output* *147*
13.3 *Output your own classes* *147*
13.4 *Output buffering* *148*
13.5 *Input* *148*
13.5.1 File input *149*
13.5.2 Input streams *149*
14 **References** *151*
14.1 *Reference* *151*
14.2 *Pass by reference* *151*
14.3 *Reference to class members* *152*
14.4 *Reference to array members* *154*
14.5 *rvalue references* *155*

15 **Pointers** *157*
15.1 *The 'arrow' notation* *157*
15.2 *Making a shared pointer* *158*
15.2.1 Pointers and arrays *158*
15.2.2 Smart pointers versus address pointers *160*
15.3 *Garbage collection* *160*
15.4 *More about pointers* *161*
15.4.1 Get the pointed data *161*
15.5 *Advanced topics* *162*
15.5.1 Unique pointers *162*
15.5.2 Base and derived pointers *162*
15.5.3 Shared pointer to 'this' *162*
15.5.4 Weak pointers *163*
15.5.5 Null pointer *163*
15.5.6 Opaque pointer *164*
15.5.7 Pointers to non-objects *164*
15.6 *Smart pointers vs C pointers* *164*
16 **C-style pointers and arrays** *167*
16.1 *What is a pointer* *167*
16.2 *Pointers and addresses, C style* *167*
16.3 *Arrays and pointers* *169*
16.4 *Pointer arithmetic* *170*
16.5 *Multi-dimensional arrays* *170*
16.6 *Parameter passing* *171*
16.6.1 Allocation *171*
16.6.2 Use of `new` *173*
16.7 *Memory leaks* *174*
17 **Const** *175*
17.1 *Const arguments* *175*
17.2 *Const references* *175*
17.2.1 Const references in range-based loops *176*
17.3 *Const methods* *177*
17.4 *Overloading on const* *177*
17.5 *Const and pointers* *178*
17.5.1 Old-style const pointers *179*
17.6 *Mutable* *180*
17.7 *Compile-time constants* *181*
18 **Prototypes and header files** *183*
18.1 *Prototypes for functions* *183*
18.1.1 Separate compilation *184*
18.1.2 Header files *184*
18.1.3 C and C++ headers *185*
18.2 *Prototypes for class methods* *186*
18.3 *Header files and templates* *186*
18.4 *Namespaces and header files* *187*
18.5 *Global variables and header files* *187*

18.6 *Modules* 188
19 **Namespaces** 189
19.1 *Solving name conflicts* 189
19.1.1 Namespace header files 190
19.2 *Best practices* 191
20 **Preprocessor** 193
20.1 *Textual substitution* 193
20.1.1 A new use for 'using' 194
20.1.2 Parametrized macros 194
20.2 *Conditionals* 194
20.2.1 Check on a value 195
20.2.2 Check for macros 195
20.2.3 Including a file only once 195
20.3 *Other pragmas* 195
21 **Templates** 197
21.1 *Templated functions* 197
21.2 *Templated classes* 198
21.2.1 Specific implementation 198
21.3 *Concepts* 198
22 **Error handling** 201
22.1 *General discussion* 201
22.2 *Mechanisms to support error handling and debugging* 202
22.2.1 Assertions 202
22.2.2 Exception handling 202
22.2.3 'Where does this error come from' 204
22.2.4 Legacy mechanisms 205
22.2.5 Legacy C mechanisms 205
22.3 *Tools* 205
23 **Standard Template Library** 207
23.1 *Complex numbers* 207
23.2 *Containers* 207
23.2.1 Maps: associative arrays 208
23.2.2 Iterators 208
23.3 *Regular expression* 211
23.3.1 Regular expression syntax 212
23.4 *Tuples and structured bindings* 212
23.5 *Union-like stuff: tuples, optionals, variants* 213
23.5.1 Tuples 214
23.5.2 Optional 214
23.5.3 Variant 214
23.5.4 Any 215
23.6 *Algorithms* 216
23.7 *Limits* 216
23.7.1 Storage 218
23.8 *Random numbers* 218
23.9 *Time* 219

23.10 *Ranges* 219
23.10.1 Infinite sequences 221
24 **Obscure stuff** 223
24.1 *Auto* 223
24.1.1 Declarations 223
24.1.2 Iterating 224
24.1.3 `decltype`: declared type 225
24.2 *Iterating over classes* 225
24.3 *Lambdas* 227
24.3.1 Lambda members of classes 228
24.3.2 Generic lambdas 229
24.3.3 Lambda in algorithm 229
24.3.4 Lambda variables by reference 230
24.4 *Casts* 230
24.4.1 Static cast 231
24.4.2 Dynamic cast 231
24.4.3 Const cast 233
24.4.4 Reinterpret cast 233
24.4.5 A word about void pointers 233
24.5 *lvalue vs rvalue* 233
24.5.1 Conversion 234
24.5.2 References 235
24.5.3 Rvalue references 235
24.6 *Move semantics* 235
24.7 *Graphics* 236
24.8 *Standards timeline* 236
24.8.1 C++98/C++03 236
24.8.2 C++11 236
24.8.3 C++14 237
24.8.4 C++17 237
24.8.5 C++20 237
25 **Graphics** 239
26 **C++ for C programmers** 241
26.1 *I/O* 241
26.2 *Arrays* 241
26.2.1 Vectors from C arrays 241
26.3 *Dynamic storage* 242
26.4 *Strings* 242
26.5 *Pointers* 242
26.5.1 Parameter passing 243
26.6 *Objects* 243
26.7 *Namespaces* 243
26.8 *Templates* 243
26.9 *Obscure stuff* 243
26.9.1 Lambda 243
26.9.2 Const 243

26.9.3 Lvalue and rvalue *243*
27 **C++ review questions** *245*
27.1 *Arithmetic* *245*
27.2 *Looping* *245*
27.3 *Functions* *245*
27.4 *Vectors* *246*
27.5 *Vectors* *246*
27.6 *Objects* *246*

III Fortran *249*
28 **Basics of Fortran** *251*
28.1 *Source format* *251*
28.2 *Compiling Fortran* *252*
28.3 *Main program* *252*
28.3.1 Program structure *252*
28.3.2 Statements *253*
28.3.3 Comments *253*
28.4 *Variables* *253*
28.4.1 Declarations *254*
28.4.2 Precision *254*
28.4.3 Initialization *255*
28.5 *Input/Output, or I/O as we say* *256*
28.6 *Expressions* *256*
28.7 *Review questions* *257*
29 **Conditionals** *259*
29.1 *Forms of the conditional statement* *259*
29.2 *Operators* *259*
29.3 *Select statement* *260*
29.4 *Boolean variables* *260*
29.5 *Review questions* *260*
30 **Loop constructs** *261*
30.1 *Loop types* *261*
30.2 *Interruptions of the control flow* *262*
30.3 *Implied do-loops* *262*
30.4 *Review questions* *263*
31 **Scope** *265*
31.1 *Scope* *265*
31.1.1 Variables local to a program unit *265*
31.1.2 Variables in an internal procedure *266*
32 **Procedures** *267*
32.1 *Subroutines and functions* *267*
32.2 *Return results* *269*
32.2.1 The ‘result’ keyword *270*
32.2.2 The ‘contains’ clause *271*
32.3 *Arguments* *271*

32.4 *Types of procedures* 272
32.5 *More about arguments* 273
33 **String handling** 275
33.1 *String denotations* 275
33.2 *Characters* 275
33.3 *Strings* 275
33.4 *Strings versus character arrays* 276
33.5 *Formatted I/O* 276
33.5.1 Format letters 276
33.5.2 Repeating and grouping 276
34 **Structures, eh, types** 279
35 **Modules** 281
35.1 *Modules for program modularization* 282
35.2 *Modules* 282
35.2.1 Polymorphism 283
35.2.2 Operator overloading 283
36 **Classes and objects** 285
36.1 *Classes* 285
36.1.1 Final proceduares: destructors 286
36.1.2 Inheritance 287
37 **Arrays** 289
37.1 *Static arrays* 289
37.1.1 Initialization 290
37.1.2 Array sections 290
37.1.3 Integer arrays as indices 291
37.2 *Multi-dimensional* 292
37.2.1 Querying an array 293
37.2.2 Reshaping 294
37.3 *Arrays to subroutines* 294
37.4 *Allocatable arrays* 294
37.4.1 Returning an allocated array 295
37.5 *Array output* 295
37.6 *Operating on an array* 296
37.6.1 Arithmetic operations 296
37.6.2 Intrinsic functions 296
37.6.3 Restricting with `where` 297
37.6.4 Global condition tests 297
37.7 *Array operations* 298
37.7.1 Loops without looping 298
37.7.2 Loops without dependencies 299
37.7.3 Loops with dependencies 300
37.8 *Review questions* 301
38 **Pointers** 303
38.1 *Basic pointer operations* 303
38.2 *Pointers and arrays* 305
38.3 *Example: linked lists* 305

39 **Input/output** *309*
39.1 *Types of I/O* *309*
39.2 *Print to terminal* *309*
39.2.1 Print on one line *309*
39.2.2 Printing arrays *310*
39.3 *File and stream I/O* *310*
39.3.1 Units *310*
39.3.2 Other write options *310*
39.4 *Unformatted output* *311*
40 **Leftover topics** *313*
40.1 *Interfaces* *313*
40.1.1 Polymorphism *313*
40.2 *Random numbers* *314*
40.3 *Timing* *314*
41 **Fortran review questions** *315*
41.1 *Fortran versus C++* *315*
41.2 *Basics* *315*
41.3 *Arrays* *315*
41.4 *Subprograms* *316*

IV Julia (largely yet to be written) *317*
42 **Basics of Julia** *319*
42.1 *Statements* *319*
42.2 *Variables* *320*
42.2.1 Datatypes *320*
42.2.2 Truth values *321*
42.3 *Expressions* *322*
42.3.1 Terminal input *322*
42.4 *Tuples* *322*
43 **Conditionals** *323*
43.1 *Basic syntax* *323*
43.2 *Advanced topics* *324*
43.2.1 Short-circuit operators *324*
43.2.2 Ternary if *324*
44 **Looping** *327*
44.1 *Basics* *327*
44.1.1 Stride *328*
44.1.2 Nested loops *328*
45 **Functions** *331*
46 **Arrays** *333*
46.1 *Introduction* *333*
46.1.1 Element access *333*
46.1.2 Vectors and matrices *334*
46.1.3 Type hierarchy *334*
46.2 *Functions* *334*

46.2.1 Built-in functions *334*
46.2.2 Scalar functions *334*
46.2.3 Vectors are dynamic *334*
46.3 *Comprehensions* *335*
46.4 *Multi-dimensional cases* *335*
47 **Structs** *337*
47.1 *Why structures?* *337*
47.2 *The basics of structures* *337*
47.3 *Structures and functions* *338*
47.3.1 Structures versus objects *338*
48 **More Julia** *339*
48.1 *Assertions* *339*

V Exercises and projects *341*
49 **Style guide for project submissions** *343*
50 **Prime numbers** *345*
50.1 *Arithmetic* *345*
50.2 *Conditionals* *345*
50.3 *Looping* *346*
50.4 *Functions* *347*
50.5 *While loops* *347*
50.6 *Structures* *347*
50.7 *Classes and objects* *348*
50.7.1 Exceptions *349*
50.7.2 Prime number decomposition *349*
50.8 *Eratosthenes sieve* *350*
50.8.1 Arrays implementation *351*
50.8.2 Streams implementation *351*
50.9 *Range implementation* *352*
51 **Geometry** *353*
51.1 *Basic functions* *353*
51.2 *Point class* *353*
51.3 *Using one class in another* *355*
51.4 *Is-a relationship* *356*
51.5 *More stuff* *357*
52 **Infectuous disease simulation** *359*
52.1 *Model design* *359*
52.1.1 Other ways of modeling *359*
52.2 *Coding up the basics* *360*
52.3 *Population* *361*
52.4 *Contagion* *362*
52.5 *Spreading* *362*
52.6 *Diseases without vaccine: Ebola and Covid-19* *363*
52.7 *Project writeup and submission* *363*
52.7.1 Program files *363*

52.7.2 Writeup 363
52.8 *Bonus: mathematical analysis* 364
53 **Google PageRank** 365
53.1 *Basic ideas* 365
53.2 *Clicking around* 366
53.3 *Graph algorithms* 367
53.4 *Page ranking* 367
53.5 *Graphs and linear algebra* 368
54 **Redistricting** 369
54.1 *Basic concepts* 369
54.2 *Basic functions* 370
54.2.1 Voters 370
54.2.2 Populations 370
54.2.3 Districting 371
54.3 *Strategy* 372
54.4 *Efficiency: dynamic programming* 374
54.5 *Extensions* 374
55 **Amazon delivery truck scheduling** 377
55.1 *Problem statement* 377
55.2 *Coding up the basics* 377
55.2.1 Address list 377
55.2.2 Add a depot 380
55.2.3 Greedy construction of a route 380
55.3 *Optimizing the route* 381
55.4 *Multiple trucks* 382
55.5 *Amazon prime* 383
55.6 *Dynamicism* 383
56 **DNA Sequencing** 385
56.1 *Basic functions* 385
56.2 *De novo shotgun assembly* 385
56.2.1 Overlap layout consensus 386
56.2.2 De Bruijn graph assembly 386
56.3 *'Read' matching* 386
56.3.1 Naive matching 386
56.3.2 Boyer-Moore matching 387
57 **High performance linear algebra** 389
57.1 *Mathematical preliminaries* 389
57.2 *Matrix storage* 390
57.2.1 Submatrices 392
57.3 *Multiplication* 392
57.3.1 One level of blocking 392
57.3.2 Recursive blocking 392
57.4 *Performance issues* 393
57.4.1 Parallelism (optional) 393
57.4.2 Comparison (optional) 393
58 **Memory allocation** 395

59 **Cryptography** *397*
59.1 *Basic ideas* *397*
60 **Climate change** *399*
60.1 *Reading the data* *399*
60.2 *Statistical hypothesis* *400*

VI Advanced topics *403*
61 **Programming strategies** *405*
61.1 *A philosophy of programming* *405*
61.2 *Programming: top-down versus bottom up* *405*
61.2.1 Worked out example *406*
61.3 *Coding style* *407*
61.4 *Documentation* *407*
61.5 *Testing* *407*
61.6 *Best practices: C++ Core Guidelines* *408*
62 **Tiniest of introductions to algorithms and data structures** *409*
62.1 *Data structures* *409*
62.1.1 Stack *409*
62.1.2 Linked lists *409*
62.1.3 Trees *414*
62.2 *Algorithms* *416*
62.2.1 Sorting *416*
62.3 *Programming techniques* *417*
62.3.1 Memoization *417*
63 **Provably correct programs** *419*
63.1 *Loops as quantors* *419*
63.1.1 Forall-quantor *419*
63.1.2 Thereis-quantor *420*
63.2 *Predicate proving* *420*
64 **Complexity** *423*
64.1 *Order of complexity* *423*
64.1.1 Time complexity *423*
64.1.2 Space complexity *423*

VII Index and such *425*
65 **General index** *427*
66 **Index of C++ keywords** *435*
67 **Index of Fortran keywords** *439*
68 **Index of Julia keywords** *443*
69 **remaining index** *447*
70 **Bibliography** *451*

PART I

INTRODUCTION

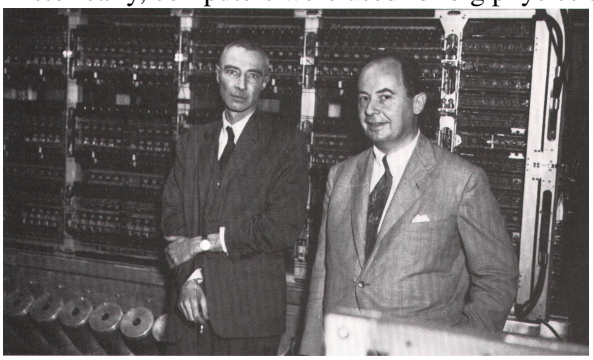
Chapter 1

Introduction

1.1 Programming and computational thinking

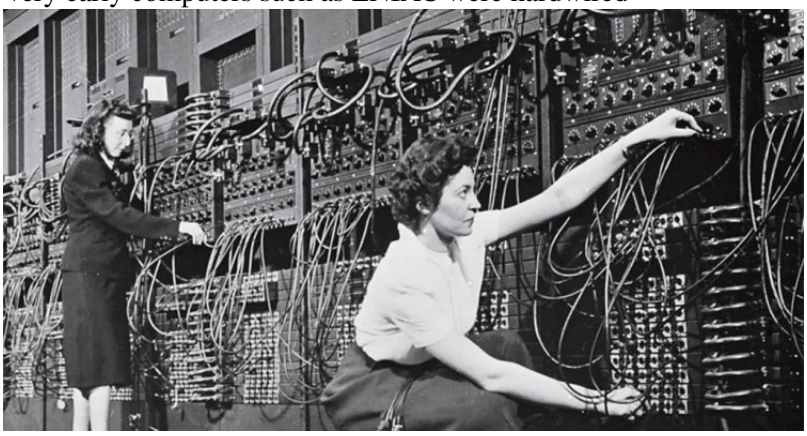
1.1.1 History

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



(Robert Oppenheimer and John von Neumann)

Very early computers such as ENIAC were hardwired

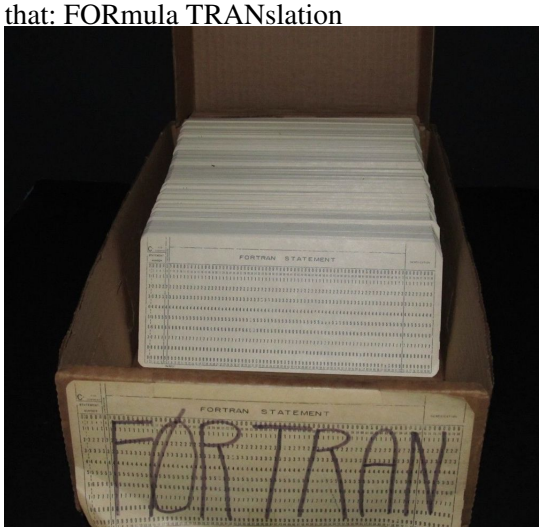


later became ‘stored program’ computer.
see http://eniacprogrammers.org/

Later programs were written on punchcards



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



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[1].

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

Programs can get pretty big:

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

Mathematical thinking:

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

Sufficient condition $\neq$ existence proof.
Computational thinking: actual design of a solution

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

Coming up with a strategy takes creativity!

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

Calculate how many schools for hair dressers the US can sustain. Identify the relevant factors, estimate

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

their sizes, and perform the calculation.

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

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

Looking up a name in the phone book

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

What is the average search time in the two cases?
Having a correct solution is not enough!

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

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

A program has layers of abstractions.

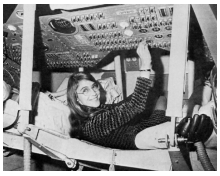
Abstraction means your program talks about your application concepts, rather than about numbers and characters and such.
Your program should read like a story about your application; not about bits and bytes.
Good programming style makes code intelligible and maintainable.
(Bad programming style may lead to lower grade.)

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

What is the structure of the data in your program?



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



Stack: you can only get at the top item



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



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

1.1.4 Hardware

Yes, it's there, but we don't think too much about it in this course.
Advanced programmers know that hardware influences the speed of execution (see TACC's ISTC course).

1.1.5 Algorithms

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

The instructions are written in some language:

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

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

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

1.2 About the choice of language

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

- They can express the sorts of problems you want to tackle in scientific computing, and

- they execute your program efficiently.

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

Python vs C++ on bubblesort:

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

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

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

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

Python with quicksort algorithm:

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

[] python arraysort.py 5000
Elapsed time: 0.00210881233215
```

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

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.

2.2 Compiling

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

Here is the workflow for program development

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

2.3 Your environment

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.

3.2 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. Remaining time will go to exams and elective topics.

lesson#	Topic	Exercises				
		in-class	homework	prime	geom	infect
1	Statements and expressions	4.4	4.11 (T)	50.1		
2	Conditionals	5.2	5.3 (T)	50.2		
4	Looping	6.3	6.4 (T)	50.3, 50.4		
5	continue					
6	Functions	7.1	7.5	50.6, 50.7 (T)		
7	continue	7.8			51.1	
8	I/O		13.1			
9	Structs		9.1	50.8 (T)		
10	Objects			50.9 (T), 50.11	51.3	52.1
11	continue					
12	has-a relation				51.9 (T), 51.10, 51.1, 51.12	52.2
13	inheritance				51.13, 51.14	
14	Arrays		11.19	50.15		52.2 and further
15	continue					
16	Strings					
Advanced						
Pointers and C-style addresses		Section 62.1.2				
Prototypes (and separate compilation) and namespaces						
Error handling and exceptions		22.1				
Lambdas		24.3				

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

3.2.1 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.

prime Prime number testing, culminating in prime number sequence objects, and testing a corollary of the Goldbach conjecture. Chapter 50.

geom Geometry related concepts; this is mostly an exercise in object-oriented programming. Chapter 51.

infect The spreading of infectuous diseases; these are exercises in object-oriented design. Students can explore various real-life scenarios. Chapter 52.

pagerank The Google Pagerank algorithm. Students program a simulated internet, and explore pageranking, including 'search engine optimization'. This exercise uses lots of pointers. Chapter 53.

scheduling Exploration of concepts related to the Traveling Salesman Problem (TSP), with some modifications that model *Amazon Prime*.

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].

3.2.2 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.

If the course focuses solely on C++, the third month can be devoted to

- templates,
- exceptions,
- namespaces,
- multiple inheritance,
- the cpp preprocessor,
- closures.

3.3 Grading guide

3.3.1 Code style

Ugly code is deceptive to the reader and to the programmer. The following can be downgraded.

3.3.1.1 Uninitialized variables

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

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

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

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

This whole issue can be 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 | ./missin

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

Aborted (core dumped)
```

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?

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:

```
#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
```

- *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.

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 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
- The option/value pair `-o myprogram`. Most Unix commands have various options that are, as the name indicates, optional. For instance you can tell the compiler to try very hard to make a fast program:

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

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

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

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

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

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

4.1.2 C++ is a moving target

The C++ language has gone through a number of standards. (This is described in some detail in section 24.8.) 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:

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

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

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

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

4.2 Statements

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

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

  ```
  cout << "Hello world" << endl; // say hi!
  ```

  and arbitrary:

  ```
  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?

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.

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

```
#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?

- The language standard says that `main` has to be of type `int`; the *return statement* returns an int.
- Compilers are fairly tolerant of deviations from this.
- Usual interpretation: returning zero means success; anything else failure;
- This *return code* can be detected by the *shell*

Code:

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

Output
[basic] return:

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

4.3 Variables

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

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

Think of a variable as a labeled placed in memory.

- The variable is defined in a *variable declaration*,
- which can include an *variable initialization*.
- After a variable is defined, and given a value, it can be used,
- or given a (new) value in a *variable assignment*.

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

4.3.1 Variable declarations

A variable is defined 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.

- 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.

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;
```

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 $i = 5$.

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

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

These are not math equations: variable on the lhs gets the value of the rhs.
You see that you can assign both a simple value or an *expression*.

Update:

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

Integer add/subtract one:

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

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

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

Exercise 4.6.

```
#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.

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.

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

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

4.3.4 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 23.1. For strings see chapter 12.

At some point you may start to wonder precisely what the range of integers or real numbers is that is storedin an `int` or `float` variable. This is addressed in section 23.7.

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`:

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

There are also `short` and `long` integers.

- A short int is at least 16 bits;
- An integer is at least 16 bits, but often 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 model*s, see https://en.cppreference.com/w/cpp/language/types.

4.3.4.2 Floating point variables

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

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

with:

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

Floating point numbers are 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, $1/2$ is representable but $1/3$ is not.

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

Floating point arithmetic is full of pitfalls.

- Don't count on `3*(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

```
float one = 1.;
```

and

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

Complex numbers exist, see section 23.1.

4.3.4.3 Boolean values

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

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

4.3.4.4 Strings

Strings, that is, strings of characters, are not a C++ built-in datatype. Thus, they take some extra setup to use. See chapter 12 for a full discussion. For now, if you want to use strings:

- Add the following at the top of your file:

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

- Declare string variables as

  ```
  string name;
  ```

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

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

What happens if you enter first and last name?

4.3.5 Initialization

It is a possible to give a variable a value right when it's created. This is known as *initialization* and it's different from creating the variable and later assigning to it (section 4.3.2).

There are two ways of initializing a variable

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

Note that writing

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

is not an initialization: it's a declaration followed by an assignment.
If you declare a variable but not initialize, you can not count on its value being anything, in particular not zero. Such implicit initialization is often omitted for performance reasons.

4.4 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.

You have already seen `cout`:

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

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

For fine-grained control over the output, and file I/O, see chapter 13.

4.5 Expressions

The most basic step in computing is to form expressions such as sums, products, logical conjuctions, string appending. Let's start with constants.

4.5.1 Numerical expressions

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

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

Math function in *cmath*:

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

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

Exercise 4.11. Write a program that :

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

4.5.2 Truth values

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

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

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

4.5.2.1 Short-cut operators

Logical expressions in C++ are evaluated using *shortcut operators*. Strictly speaking, the expression

```
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.

4.5.2.2 *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.

Bitwise: `&` `|` `^`

4.5.3 Type conversions

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

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

or

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

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

Exercise 4.12.

- 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

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

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

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

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

or any other mechanism that forces a conversion, without changing the result. Another mechanism is the *cast*; this will be discussed in section 24.4.

Exercise 4.13. Write a program that asks for two integer numbers `n1,n2`.

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

$$n_1 \mod n_2$$

(without using the `%` modulus operator!)
Print out the value you get.
- Also print out the result from using the modulus operator:`%`.
- Investigate the 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.

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

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

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

Review 4.4. True of false?

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

4.6 Advanced topics

4.6.1 Library functions

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

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

Others, such as *max*, are in the less common `algorithm`:

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

4.6.2 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 representated 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:

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

Your program will not be interrupted if a `NaN` or `Inf` is generated: the computation will merrily (and at full speed) progress with these numbers. See section 23.7 for detection of such quantities.

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

4.6.3 Constants

- Default: `double`
- Float: `3.14f` or `3.14F`
- Long double: `1.22l` or `1.22L`.

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

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

Binary numbers are new to C++17.

4.6.4 Numerical precision

Truncation and precision are tricky things.

Code:

```
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:

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

Review 4.6. Given

```
int n;
```

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

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

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

Chapter 5

Conditionals

A program consisting of just a list of assignment and expressions would not be terribly versatile. At least you want to be able to say 'if $x > 0$, 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.

Single statement

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

Single statement and alternative:

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

Multiple statements:

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

Chaining conditionals (where the dots stand for omitted code):

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

Nested conditionals:

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

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

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

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

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

5.2 Operators

You have already seen arithmetic expressions; now we need to look at logical expressions: just what can be tested in a conditional. 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.

```
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
```

Here are the most common *logic operators* and *comparison operators*.

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

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

Remark 1 *There are also* bitwise operators.

Exercise 5.2. Read in an integer. If it is even, print 'even', otherwise print 'odd':

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

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

Exercise 5.3. Read in a positive integer. If it's a multiple of three print 'Fizz!'; if it's a multiple of five print 'Buzz!'. It it is a multiple of both three and five print 'Fizzbuzz!'. Otherwise print nothing.

Note:

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

Review 5.1. True or false?

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

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

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

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

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

Review 5.2. Any comments on the following?

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

5.3 Switch statement

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

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

Code:

```
switch (n) {
case 1 :
case 2 :
  cout << "very small" << endl;
  break;
case 3 :
  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
large
```

Exercise 5.4. Suppose the variable `n` is a nonnegative integer. Write a `switch` statement that has the same effect as:

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

5.4 Scopes

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

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

See chapter 8 for more detail.

5.5 Advanced topics

5.5.1 Short-circuit operators

C++ 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.

5.5.2 Ternary if

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

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

But what about the case where the true and false branch 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 = foo ? 5 : 6;
```

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

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

5.5.3 Initializer

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

Code:

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

Output
[basic] ifinit:

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

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

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

with the variable globally defined.

5.6 Review questions

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

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

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

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

compare:

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

```
}
```

Chapter 6

Looping

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

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

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

- Inspect or change every element of a database table.

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

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

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

We will now first consider the `for` loop; the `while` loop comes in section 6.2.

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:

```
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*.

Remark 2 *Declaring the loop variable in the loop header is also a modern addition to the C language. Use compiler flag* `-std=c99`.

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++.

The loop header has three components, all of which are optional.

- An initialization. This is usually a declaration and initialization of an integer *loop variable*. Using floating point values is less advisable.
- A stopping test, usually involving the loop variable. If you leave this out, you need a different mechanism for stopping the loop; see section 6.2.
- An increment, often `i++` or spelled out `i=i+1`. You can let the loop count down by using `i--`.

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!

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.
- `i++` for a loop that counts forward;
- `i--` for a loop that counts backward;
- `i+=2` to cover only odd or even numbers, depending on where you started;
- `i*=10` to cover powers of ten.

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

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

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

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

Review 6.2. What is the last iteration executed?

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

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

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

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

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

Some variations on the simple case.

- It is preferable to declare the loop variable in the loop header:

  ```
  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:

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

  You will see an example where this makes sense below.
- The stopping test can be omitted

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

  if the loops ends in some other way. You’ll see this later.
- The stopping test doesn’t need to be an upper bound. Here is an example of a loop that counts down to a lower bound.

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

- The test is performed before each iteration:

Code:

```
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
```

(Historical note: at some point Fortran was post-test, so one iteration was always performed.)

- The loop body can be a single statement:

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

  or a block:

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

6.1.1 Loop syntax

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

```
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.

```
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 6.3. 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 $i = \text{constant}$ and $j = \text{constant}$ respectively, a diagonal is defined by $i + j = \text{constant}$.

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 appopriate for looping over the elements of an array, but not if you are coding some

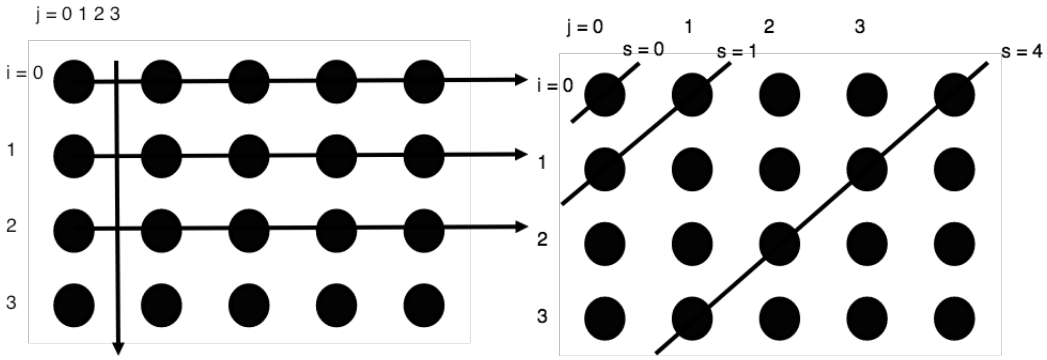


Figure 6.1: Lexicographic and diagonal ordering of an index set

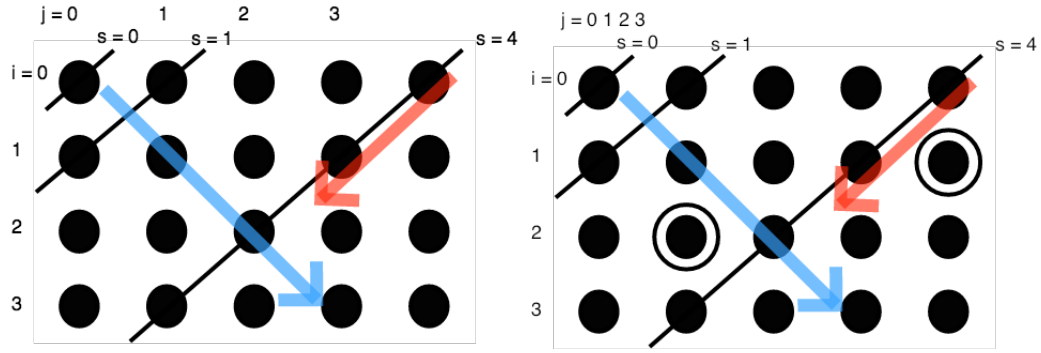


Figure 6.2: Illustration of the second part of exercise 6.4

.

process that needs to run until some dynamically determined condition is satisfied. In this section you will see some ways of coding such cases.

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

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

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

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

For the following exercise, see figure 6.2 for inspiration.

Exercise 6.4. 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 6.5. All three parts of a loop header are optional. What would be the meaning of

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

?

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

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

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

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

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

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

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

Alternative:

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

The only difference is in layout.

The other possibility for 'looping until' is a `while` loop, which repeats until a condition is met. Syntax:

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

or

```
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.

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

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

Code:

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

Problem: code duplication.

Code:

```
int invar;
do {
  cout << "Enter a positive number: " ;
  cin >> invar; cout << endl;
  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 6.6. At this point you are ready to do the exercises in the prime numbers project, section 50.3.

Exercise 6.7. 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 6.8. 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

6.3.1 Parallelism

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

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

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

- Inspect or change every element of a database table.

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

6.4 Exercises

Exercise 6.9. 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 6.10. The integer sequence

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

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

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

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

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

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

Exercise 6.11. 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`.

Exercise 6.12. **Root finding.** 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 exercise you will implement a simple scheme.

Suppose x_-, x_+ are such that

$$x_- < x_+, \qquad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values are of opposite sign. Then there is a zero in the interval (x_-, x_+). Try to find this zero by looking at the halfway point, and based on the function value there, considering the left or right interval.

- How do you find x_-, x_+? This is tricky in general; if you can not find them in the interval $[-10^6, +10^6]$, halt the program.
- Finding the zero exactly may also be tricky or maybe impossible. Stop your program if $|x_- - x_+| < 10^{-10}$.

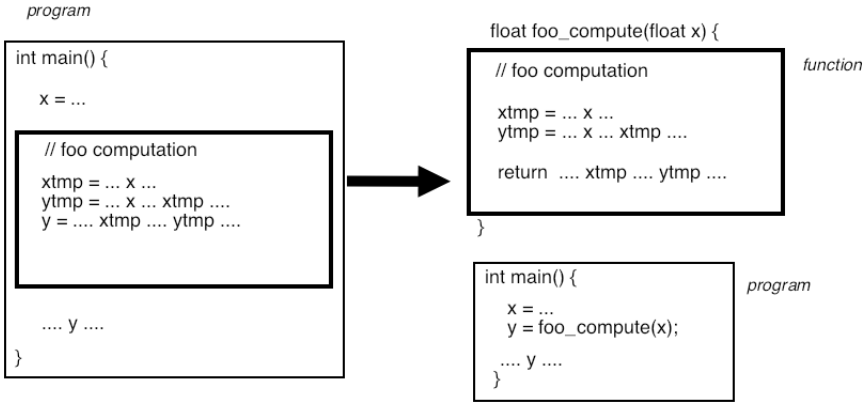
The website http://www.codeforwin.in/2015/06/for-do-while-loop-programming-exercises.html

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.



By introducting a function name you have introduced *abstraction*: your program now uses terms related to your problem, and not only `for` and `int` and such.

7.1 Function definition and call

There are two sides to a function:

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

Example: zero-finding through bisection.

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

Step 1: everything in the main program.

Code:

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

Output
[func] bisect1:

```
Zero happens at: 1.4375
```

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

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

Used in main:

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

Add function for zero finding:

(note the local variable)

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

The main no longer contains any implementation details (local variables, method used):

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

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 `(int n)` 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.

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.

Function 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 occurences of the function code do not have to be identical:

Suppose you do the same computation twice:

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

With a function this can be replaced by:

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

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

Example: multiple norm calculations:

Repeated code:

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

becomes:

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

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

A final argument for using functions is code maintainability:

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

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

Review 7.1. True or false?

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

The compiler needs to know about a function before you can use it

Solution 1: define before use

```
double f(double x) {
  return x+1; }

int main() {
  double x=1;
  double y = f(x);
}
```

Solution 2: declare before use, define later

```
double f(double);

int main() {
  double x=1;
  double y = f(x);
}

double f(double x) {
  return x+1; }
```

7.2 Anatomy of a function definition and call

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

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**.)

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. This is why above we said that the function definition has to come before the main program.

This is not entirely true: the translation of a program that uses a function can proceed once the compiler know its name, and the types of the inputs and result. Just like you can declare a variable and set its value later, you can declare the existence of a function and give its definition later. The resulting code looks like:

```
int my_computation(int);
int main() {
  int result;
  result = my_computation(5);
  return 0;
};
int my_computation(int i) {
   return i+3;
}
```

The line above the main program is called a *function header* or *function prototype*. See chapter 18 for more details.

7.3 Void functions

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

```
void print_header() {
  cout << "****************" << endl;
  cout << "* Output       *" << endl;
  cout << "****************" << endl;
  }
int main() {
  print_header();
  cout << "The results for day 25:" << endl;
```

```
  // code that prints results ....
  return 0;
}
```

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

Review 7.2. True or false?

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

7.4 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. (Later you will learn how to pack together multiple outputs.)

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

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(x1,x2,x3);
  ```

- 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:

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.

Code:

```
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;
```

Output
[func] passvalue:

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

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

Later we will worry about the cost of this copying.

Exercise 7.1. Write two functions

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

and a program that uses them to swap two values:

```
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:

Code:

```
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;
```

Output
[func] localparm:

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

Exercise 7.2. If you are doing the prime project (chapter 50) you can now do exercise 50.6.

Early computers had no hardware for computing a square root. Instead, they used *Newton's method*. Suppose you have a value y and you want want to compute $x = \sqrt{y}$. This is equivalent to finding the zero of

$$f(x) = x^2 - y$$

where y is fixed. To indicate this dependence on y, we will write $f_y(x)$. Newton's method then finds the zero by evaluating

$$x_{\text{next}} = x - f_y(x)/f'_y(x)$$

until the guess is accurate enough, that is, until $f_y(x) \approx 0$.

Exercise 7.3.

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

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*:

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

Code:

```
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << 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.)

You can make a function parameter into a reference of a variable in the main program:

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

```
void f(int &n) {
  n = /* some expression */ ;
};
int main() {
```

```
  int i;
  f(i);
  // i now has the value that was set in the function
}
```

Using the ampersand, the parameter is *passed by reference*: instead of copying the value, the function receives a *reference*: information where the variable is stored in memory.

Remark 3 *The* pass by reference mechanism in C *was different and should not be used in C++. In fact it was not a true pass by reference, but passing an address by value.*

We also the following terminology for function parameters:

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

Code:

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

  int var = 0;
  f(var);
  cout << var << 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.

```
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' ....
```

Exercise 7.4. Write a `void` function `swapij` of two parameters that exchanges the input values:

```
int i=2,j=3;
swapij(i,j);
// now i==3 and j==2
```

Exercise 7.5. 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.

```
int number,divisor,remainder;
// read in the number and divisor
if ( is_divisible(number,divisor,remainder) )
  cout << number << " is divisible by " << divisor << endl;
else
  cout << number << "/" << divisor <<
      " has remainder " << remainder << endl;
```

Exercise 7.6. If you are doing the geometry project, you should now do the exercises in section 51.1.

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

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

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

$$F(n) = n \times F(n-1) \qquad \text{if } n > 0, \text{ otherwise } 1$$

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

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

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

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

So what happens if you write

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

Well,

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

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

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

Exercise 7.7. The sum of squares:

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

can be defined recursively as

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

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

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

Exercise 7.8. Write a recursive function for computing Fibonacci numbers:

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

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

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

Remark 4 *A function does not need to call itself directly to be recursive; if it does so indirectly we can call this* mutual recursion.

Remark 5 *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 62.3.1.

7.6 Advanced function topics

7.6.1 Default arguments

Functions can have *default argument*(s):

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

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

7.6.2 Polymorphic functions

You can have multiple functions with the same name:

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

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

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

7.6.3 Stack overflow

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

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

and its implementation:

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

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

```
int oneless = 2;
```

and then the computation of g_2. But that computation involved

```
int oneless = 1;
```

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

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

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

it may lead to *stack overflow*.

7.7 Library functions

7.7.1 Random function

There is an easy (but not terribly great) *random number generator* that works the same as in C (for a better generator, see section 23.8):

```
#include <random>
using std::rand;
float random_fraction =
    (float)rand()/(float)RAND_MAX;
```

The function `rand` yields an integer – 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.

If you run your program twice, you will twice get the same sequence of random numbers. That is great for debugging your program but not if you were hoping to do some statistical analysis. Therefore you can set the *random number seed* from which the random sequence starts by the `srand` function. Example:

```
srand(time(NULL));
```

seeds the random number generator from the current time.

7.8 Review questions

Review 7.3. What is the output of the following programs (assuming the usual headers):

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

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

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

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

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

Review 7.4. Suppose a function

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

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

Review 7.5. Suppose a function

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

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

Review 7.6. Suppose a function

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

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

Review 7.7. Suppose a function

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

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

Chapter 8

Scope

8.1 Scope rules

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

8.1.1 Lexical scope

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

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

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

8.1.2 Shadowing

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

Code:

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

Output
[basic] shadowtrue:

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

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

This is independent of dynamic / runtime behaviour!

Exercise 8.1. What is the output of this code?

```
bool something{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;
```

Exercise 8.2. What is the output of this code?

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

8.1.3 Lifetime versus reachability

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

```
void f() {
  ...
```

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

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

A special case of this is recursion:

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

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

8.1.4 Scope subtleties

8.1.4.1 Mutual recursion

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

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

you need a *forward declaration*

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

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

There is also forward declaration of *classes*.

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

8.1.4.2 Closures

The use of lambdas or *closures* (section 24.3) comes with another exception to general scope rules. Read about ‘capture’.

8.2 Static variables

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

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

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

For example

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

gives a stream of integers.

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

8.3 Scope and memory

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

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

Recursive functions offer a complication:

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

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

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

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

Code:

```
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

Review 8.1. Is this a valid program?

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

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

Review 8.2. What is the output of:

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

Review 8.3. What is the output of:

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

```
  return 0;
}
```

Review 8.4. What is the output of:

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

Chapter 9

Structures

9.1 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[1] A **`struct`** acts like a datatype for which you choose the name. A **`struct`** contains other datatypes; these can be elementary, or other structs.

```
struct Point {
  double x; double y; int label;
};
```

The elements of a structure are also called *members*.

The reason for using structures is to 'raise the level of abstraction': instead of talking about x, y-values, you can now talk about points and such. This makes your code look closer to the application you are modeling.

9.2 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 inside your main program or before it. The latter is preferable.

```
// definition of the struct
struct StructName { int num; double val; }
int main() {
  // declaration of struct variables
  StructName mystruct1,mystruct2;
  .... code that uses your structures ....
}
```

There are various ways of setting the members of the structure:

1. Note for instructors: structures in C++ are actually largely the same as classes. This chapter really teaches structures as they appear in C. However, that's a good first step towards classes.

- You can set defaults that hold for any structure of that type;
- you can all members at once;
- or you can set any member individually.

You assign a whole struct, or set defaults in the definition.

```
struct Point_a { double x; double y; } ;

// initialization when you create the variable:
struct Point_a x_a = {1.5,2.6};
```

Once you have defined a structure, you can make variables of that type. Setting and initializing them takes a new syntax:

Code:

```
int main() {

  struct Point p1,p2;

  p1.x = 1.; p1.y = 2.; p1.label = 5;
  p2 = {3.,4.,5};

  p2 = p1;
  cout << "p2: "
       << p2.x << "," << p2.y
       << endl;
```

Output
[struct] point:

```
./point
p2: 1,2
```

Period syntax: compare to possessive 'apostrophe-s' in English.

Review 9.1. True or false?

- All members of a struct have to be of the same type.
- Writing

  ```
  struct numbered { int n; double x; };
  ```

  creates a structure with an integer and a double as members.
- All members of a struct have to be of different types.
- With the above definition and `struct numbered xn;`,

  ```
  cout << xn << endl;
  ```

  Is this correct C++?
- With the same definitions, is this correct C++?

  ```
  xn.x = xn.n+1;
  ```

Remark 6 *Before C++14, if you used initializations in the* `struct` *definition, you could not use the brace-assignment. This has been fixed.*

```
// initialization done in the structure definition:
struct Point_b { double x=0; double y=0; } ;

struct Point_b x_b;
// legal since 14: ILLEGAL:
x_b = {3.7, 4.8};
// always legal
x_b.x = 3.7; x_b.y = 4.8;
```

9.3 Structures and functions

You can use structures with functions, just as you could with elementary datatypes.

You can pass a structure to a function:

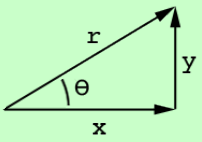
Code:

```
double distance
  ( struct point p1,
    struct point p2 )
{
  double
    d1 = p1.x-p2.x, d2 = p1.y-p2.y;
  return sqrt( d1*d1 + d2*d2 );
}
  /* ... */
  cout << "dx=" << dx
       << ", dy=" << dy << endl;
  struct point p2 = { p1.x+dx,p1.y+dy };
  cout << "Distance: "
       << distance(p1,p2) << endl;
```

Output
[struct] pointfun:

```
dx=5, dy=12
Distance: 13
```

Exercise 9.1. Write a point structure, and a function that, given such a point, returns the angle with the x-axis. (Hint: the `atan` function is in `cmath`)



Code:

```
struct point a = {1.,1.};
double
  alpha = angle(a),
  pifrac = (4.*atan(1.0)) / alpha;
cout << "Angle of ("
     << a.x << "," << a.y
     << ") is:\n " << angle(a)
     << ", or pi/" << pifrac
     << endl;
```

Output
[struct] pointangle:

```
Angle of (1,1) is:
 0.785398, or pi/4
Angle of (0.866025,0.5) is:
 0.523599, or pi/6
```

The mechanisms of *parameter passing by value* and *parameter passing by reference* apply to structures just as to simple datatypes.

Exercise 9.2. Write a `void` function that has a `struct Point` parameter, and exchanges its coordinates:

$$\begin{pmatrix}2.5\\3.5\end{pmatrix} \rightarrow \begin{pmatrix}3.5\\2.5\end{pmatrix}$$

Code:

```
struct point a = {3.,2.};
cout << "Flip of ("
     << a.x << "," << a.y << ")";
flip(a);
cout << " is ("
     << a.x << "," << a.y
     << ")" << endl;
```

Output
[struct] pointflip:

```
Flip of (3,2) is (2,3)
```

Exercise 9.3. Write a function $y = f(x, a)$ that takes a `struct Point` and `double` parameter as input, and returns a point that is the input multiplied by the scalar.

$$\begin{pmatrix} 2.5 \\ 3.5 \end{pmatrix}, 3 \to \begin{pmatrix} 7.5 \\ 10.5 \end{pmatrix}$$

Exercise 9.4. If you are doing the prime project (chapter 50) you can now do exercise 50.8.

Exercise 9.5. Write a function `inner_product` that takes two `Point` structures and computes the inner product. Test your code on some points where you know the answer.

For a more sophisticated version of this exercise:

Exercise 9.6. Write two functions:

1. `rotate` takes a point v and a scalar α and returns a point that is v rotated over α.
2. `inner_product` that takes two points and returns their inner product.

Now apply the inner product function on a point v and v rotated by $\pi/2$. Is their inner product zero?

Here you can learn about floating point arithmetic and floating point error.

Now write a boolean function `is_orthogonal` that works in the presence of floating point error.

You can use initializer lists as struct *denotations*
(with the `distance` function previous defined):

Code:

```
struct point p1{1.,2.}, p2{6.,14.};
cout << "Distance: "
     << distance( p1,p2 )
     << endl;
cout << "Distance: "
     << distance( {1.,2.}, {6.,14.} )
     << endl;
```

Output
[struct] pointdenote:

```
Distance: 13
Distance: 13
```

Exercise 9.7. Take exercise 9.5 and rewrite it to use denotations.

You can return a structure from a function:

Code:

```
struct point point_add
       ( struct point p1,
         struct point p2 ) {
   struct point p_add =
     {p1.x+p2.x,p1.y+p2.y};
   return p_add;
};
  /* ... */
  v3 = point_add(p1,p2);
  cout << "Added: " <<
    v3.x << "," << v3.y << endl;
```

Output
[struct] pointadd:

```
Added: 5,6
```

(In case you're wondering about scopes and lifetimes here: the explanation is that the returned value is copied.)

Exercise 9.8. Write a 2×2 matrix class (that is, a structure storing 4 real numbers), and write a function `multiply` that multiplies a matrix times a vector.

Can you make a matrix structure that is based on the vector structure (essentially the same as a `Point` struct), for instance using vectors to store the matrix rows, and then using the inner product method to multiply matrices?

Chapter 10

Classes and objects

10.1 What is an object?

You learned about **struct**s (chapter 9) as a way of abstracting from the elementary data types. The elements of a structure were called its members.

You also saw that it is possible to write functions that work on structures. We now combine these two elements to give a new level of abstraction:

An object is an entity that you can request to do certain things. These actions are the *methods* and to make these possible the object probably stores data, the *members*.
When designing a class, first ask yourself: 'what functionality should the objects support'.
Objects comes in *class*es. A class is like a datatype: you can make objects of a class like variables of a datatype.

Objects belong to a class, so, as with structures:

- 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.

Objects behave largely like any other datatype.

Let's look at a simple example: as with structures we want a `Point` object, corresponding to a mathematical vector in $\mathbb{R}^2$. Given a point, you could want to know its distance to the origin (let's call this its 'length), or its angle with the x-axis.

Small illustration: vector objects.

Code:

```
Point p(1.,2.); // make point (1,2)
cout << "distance to origin "
     << p.length() << endl;
p.scaleby(2.);
cout << "distance to origin "
     << p.length() << 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; in a **struct** we use it for the data members; in an object we (also) use it for methods.

Exercise 10.1. Thought exercise:
What data does the object need to store to do this?
Is there more than one possibility?

Similar to `struct`:

- You have to define what an object looks like by giving a

```
class myobject { /* ... */ };
```

  definition, typically before the `main`.
- You create specific objects with a declaration

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

- You let the objects do things:

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

10.1.1 Constructor

Next we'll look at a syntax for creating class objects that is new. If you create an object, you actually call a function that has the same name as the class: the *constructor*.

Usually you write your own constructor, for instance to initialize data members. This fragment of the `class Point` shows the constructor and the data members:

```
class Point {
private: // members
  double x,y;
public: // methods, first the constructor:
  Point
    ( double in_x,double in_y )
    : x(in_x),y(in_y) {};
  /* ... */
};
```

Slightly less confusing:

```
class Point {
private: // members
  double x,y;
public: // methods
  Point( double in_x,double in_y )
    : x(in_x),y(in_y) {};
};
```

- Keyword `private` indicates that data is internal
- keyword `public` indicates that the constructor function can be used in the program.
- The initialization `x(x)` is a *member initializer list*; it should be read as `membername(argumentname)`. Yes, having `x` twice is a little confusing.

Remark 7 *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.*

10.1.2 Methods, introduction

Methods are things you can ask your class objects to do. For instance, in the `Point` class, you could ask a vector what its length is, or you could ask it to scale its length by some number.

Let's define method `length`.

Definition in the class:

```
double length() {
  return sqrt(x*x + y*y); };
```

Use in the program:

```
Point pt(5,12);
double
  s = pt.length();
```

- Methods look like ordinary functions,
- except that they can use the data members of the class, for instance `x`;
- Methods can only be used on an object with the 'dot' notation. They are not independently defined.

10.1.3 Initialization

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

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

Each object will have its members initialized to these values.

Other syntax for initialization:

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

The members stored can be different from the constructor arguments.
Example: create a vector from `x,y` cartesian coordinates, but store `r,theta` polar coordinates:

```
class Point {
private: // members
  double r,theta;
public: // methods
  Point( double x,double y ) {
    r = sqrt(x*x+y*y);
    theta = atan(y/x);
  }
```

Allows for reuse of names:

Code:

```
class Vector {
private:
  double x,y;
public:
  Vector( double x,double y ) : x(x),y(y) {
  }
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 = "
       << p1.getx() << "," << p1.gety()
       << endl;
```

Output
[geom] pointinitxy:

```
p1 = 1,2
```

Also saves on constructer invocation if the member is an object.

Initializer lists can be used as denotations.

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

10.1.4 Member access

Objects are quite a bit like structures: they have data members, and they use the 'dot' syntax. In fact, if we make data members public (Not! A! Good! Idea!) they behave entirely the same:

```
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;
}
```

Data members should not be accessed directly from outside an object, but using them inside a method is proper:

```
class Point {
private:
  double x,y;
public:
  void flip() {
    Point flipped;
    flipped.x = y;
    flipped.y = x;
  };
};
```

10.1.5 Methods

You have just seen examples of class *method*s: a function that is only defined for objects of that class, and that has access to the private data of that object. Let's now look at more meaningful methods. For instance, for the `Point` class you can define functions such as `length` and `angle`.

Code:

```
class Point {
private:
  double x,y;
public:
  Point( double x,double y )
    : x(x),y(y) {};
  double length() {
    return sqrt(x*x + y*y); };
  double angle() {
    return 0.; /* something trig */; };
};

int main() {
  Point p1(1.,2.);
  cout << "p1 has length "
       << p1.length() << endl;
```

Output

[geom] pointfunc:

```
p1 has length 2.23607
```

Exercise 10.2. Add a method `angle` to the `Point` class.

How many parameters does it need?

Exercise 10.3. Discuss the pros and cons of this design:

```
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 10.1. T/F?

- A class is primately determined by the data it stores
- A class is primarily determing by its methods

- If you change class data, you need to change the constructor call

10.1.5.1 Methods that return objects

So far you have seen methods that use the data members of an object to return some quantity. It is also possible to alter the members. For instance, you may want to scale a vector by some amount:

Code:

```
class Point {
  /* ... */
  void scaleby( double a ) {
    vx *= a; vy *= a; };
  /* ... */
};
  /* ... */
  Point p1(1.,2.);
  cout << "p1 to origin "
       << p1.length() << endl;
  p1.scaleby(2.);
  cout << "p1 to origin "
       << p1.length() << endl;
```

Output
[geom] pointscaleby:

```
p1 to origin 2.23607
p1 to origin 4.47214
```

The methods you have seen so far only returned elementary datatypes. It is also possible to return an object, even from the same class. For instance, instead of scaling the members of a vector object, you could create a new object based on the scaled members:

Code:

```
class Point {
  /* ... */
  Point scale( double a ) {
    return Point( vx*a, vy*a ); };
  /* ... */
  cout << "p1 to origin "
       << p1.length() << endl;
  Point p2 = p1.scale(2.);
  cout << "p2 to origin "
       << p2.length() << endl;
```

Output
[geom] pointscale:

```
p1 to origin 2.23607
p2 to origin 4.47214
```

(also known as 'move semantics') Instead of

```
  Point scale( double a ) {
    return Point( vx*a, vy*a ); };
```

we could have written:

```
Point scale( double a ) {
  Point new_point( vx*a, vy*a );
  return new_point;
};
```

However, that may involve an extra copy.
(Depends on standard version.)

10.1.5.2 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.

Example: scaling a vector:

```
void scaleby( double a ) {
  x *= a; y *= a; };
```

10.1.6 Default constructor

So far you've seen class and their constructors. A class with no constructor defined by you still has a *default constructor*. For example:

```
class IamZero {
  private:
  int i=0;
  public:
  void print() { cout << i; };
};
```

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

```
IamZero zero;
zero.print();
```

Such a class acts like it has a constructor

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

Bear this in mind as you study the following code:

```
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

```
Point p2;
```

calls the default constructor. Now that you have defined your own constructor, the default constructor no longer exists. So you need to define it explicitly:

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

10.1.7 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?

Objects are really supposed to be accessed through their functionality.

Another reason for making data members inaccessible is that we can change the internals of a class, without affecting the code that uses the class. Consider the scenario where you have a `Point` class, and you store the x, y coordinates. Then you change you mind and decide to store r, θ coordinates instead. If

your program used the fact that it could directly access the x, y coordinates, you would be in for a lot of rewriting. On the other hand if you have a function `getx()`, your program does not need any rewriting, and you would only change how that function was written.

```
class PositiveNumber { /* ... */ }
class Point {
private:
  // data members
public:
  Point( float x,float y ) { /* ... */ };
  Point( PositiveNumber r,float theta ) { /* ... */ };
  float get_x() { /* ... */ };
  float get_y() { /* ... */ };
  float get_r() { /* ... */ };
  float get_theta() { /* ... */ };
};
```

The 'get' functionality is independent of what data members the `Point` class has.

- 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.

We make a class for points on the unit circle
You don't want to be able to change just one of `x,y`!

```
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.

10.1.8 Examples

Objects can model fairly abstract things:

Code:

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

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

Output
[object] stream:

```
Next: 0
Next: 1
Next: 2
```

Exercise 10.4.

- 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.

Exercise 10.5. If you are doing the prime project (chapter 50), now is a good time to do exercise 50.9.

10.2 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.

10.2.1 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

```
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:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
}
```

In both cases the code using the object is written as if the segment object contains two points. This illustrates how object-oriented programming can decouple the API of classes from their actual implementation.

Related to this decoupling, a class can also have two very different constructors.

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

Depending on how you actually implement the class, the one constructor will simply store the defining data, and the other will do some conversion from the given data to the actually stored data.

This is another strength of object-oriented programming: you can change your mind about the implementation of a class without having to change the program that uses the class.

When you have a has-a relation between classes, the 'default constructor' problem (section 10.1.6) can pop up again:

```
class Inner { /* ... */ };
class Outer {
private:
  Inner inside_thing;
```

Two possibilities for constructor:

```
Outer( Inner thing )
: inside_thing(thing) {};
```

The *Inner* object is copied during construction of *Outer* object.

```
Outer( Inner thing ) {
  inside_thing = thing;
};
```

The *Outer* object is created, including construction of *Inner* object, then the argument is copied into place: ⇒ needs default constructor on *Inner*.

Exercise 10.6. If you are doing the geometry project, this is a good time to do the exercises in section 51.3.

10.3 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).

- 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.

```
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:

  ```
  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:

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

Methods and data can be

- private, because they are only used internally;
- public, because they should be usable from outside a class object, for instance in the main program;
- protected, because they should be usable in derived classes.

Exercise 10.7. If you are doing the geometry project, you can now do the exercises in section 51.4.

10.3.1 Methods of base and derived classes

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class can *override* a base class method:

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

Code:

```
class Base {
protected:
  int i;
public:
  Base(int i) : i(i) {};
  virtual int value() { return i; };
};

class Deriv : public Base {
public:
  Deriv(int i) : Base(i) {};
  virtual int value() override {
    int ivalue = Base::value();
    return ivalue*ivalue;
  };
};
```

Output
[object] virtual:

```
25
```

10.3.2 Virtual methods

- Method defined in base class: can be used in any derived class.
- Method define in derived class: can only be used in that particular derived class.
- Method defined both in base and derived class, marked `override`: derived class method replaces (or extends) base class method.
- Virtual method: base class only declares that a routine has to exist, but does not give base implementation.
  A class is called *abstract class* if it has virtual methods; pure virtual if all methods are virtual.
  You can not make abstract objects.

Special syntax for *abstract method*:

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

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

Suppose `DenseVector` derives from `VirtualVector`:

```
DenseVector v(5);
v.setlinear(7.2);
cout << v[3] << endl;
```

```
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);
  };
};
```

missing snippet virtfuncfunc

Exercise 10.8. Advanced! Write a small 'integrator' library for Ordinary Diffential Equations (ODEs)s. Assuming 'autonomous ODEs', that is $u' = f(t)$ with no u-dependence, there are two simple integration schemes:

- explicit: $u_{n+1} = u_n + \Delta t\, f_n$; and

- implicit: $u_{n+1} = u_n + \Delta t\, f_{n+1}$.

Write an abstract `Integrator` class where the `nextstep` method is pure virtual; then write `ExplicitIntegrator` and `ImplicitIntegrator` classes deriving from this.

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

10.3.3 Advanced topics in inheritance

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

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

- 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'.

10.4 Advanced topics

The remainder of this section is advanced material. Make sure you have studied section 14.3.

10.4.1 Returning by reference

Return a reference to a private member:

```
class Point {
private:
  double x,y;
public:
  double &x_component() { return x; };
};
int main() {
  Point v;
  v.x_component() = 3.1;
}
```

Only define this is really needed.

Returning a reference saves you on copying.
Prevent unwanted changes by using a 'const reference'.

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

10.4.2 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:

```
class thing {
private:
 float x;
public:
 float get_x() { return x; };
 void set_x(float v) { x = v; }
};
```

This has advantages:

- You can print out any time you get/set the value; great for debugging:

  ```
  void set_x(float v) {
    cout << "setting: " << v << endl;
    x = v; }
  ```

- You can catch specific values: if x is always supposed to be positive, print an error (throw an exception) if nonpositive.

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

Use a single accessor for getting and setting:

Code:

```
class SomeObject {
private:
  float x=0.;
public:
  SomeObject( float v ) : x(v) {};
  float &xvalue() { return x; };
};

int main() {
  SomeObject myobject(1.);
  cout << "Object member initially :"
       << myobject.xvalue() << 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!

Exercise 10.9. This is a good time to do the exercises in section 51.2.

10.4.3 Accessability

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.

10.4.4 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*.

10.4.5 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

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

For instance:

Code:

```
Point operator*(double factor) {
  return Point(factor*vx,factor*vy);
};
/* ... */
cout << "p1 to origin "
     << p1.length() << endl;
Point scale2r = p1*2.;
cout << "scaled right: "
     << scale2r.length() << endl;
// ILLEGAL Point scale2l = 2.*p1;
```

Output
[geom] pointmult:

```
p1 to origin 2.23607
scaled right: 4.47214
```

Can even redefine equals and parentheses.

Exercise 10.10. Write a `Fraction` class, and define the arithmetic operators on it.

10.4.6 Functors

A special case of operator overloading is *overloading the parentheses*. This makes an object look like a function; we call this a *functor*.

Simple example:

Code:

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

Output
[object] functor:

```
5
```

Exercise 10.11. Extend that class as follows: instead of printing the argument directly, it should print it multiplied by a scalar. That scalar should be set in the constructor. Make the following code work:

Code:

```
IntPrintTimes printx2(2);
printx2(1);
for ( auto i : {5,6,7,8} )
  printx2(i);
```

Output
[object] functor2:

```
2
10
12
14
16
```

(The `for_each` is part of `algorithm`)

10.4.7 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*. This constructor is invoked whenever you do an obvious copy:

```
my_object x,y; // regular or default constructor
x = y;         // copy 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.)

Example of the copy constructor in action:

```
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; };
};
```

Code:

```
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
```

Class with a vector:

```
class has_vector {
private:
  vector<int> myvector;
public:
  has_vector(int v) { myvector.push_back(v); };
  void set(int v) { myvector.at(0) = v; };
  void printme() { cout
      << "I have: " << myvector.at(0) << endl; };
};
```

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

Code:

```
has_vector a_vector(5);
has_vector other_vector(a_vector);
a_vector.set(3);
a_vector.printme();
other_vector.printme();
```

Output
[object] copyvector:

```
I have: 3
I have: 5
```

10.4.8 Destructor

Just as there is a constructor routine to create an object, there is a *destructor* to destroy the object. As with the case of the default constructor, there is a default destructor, which you can replace with your own.

A destructor can be useful if your object contains dynamically created data: you want to use the destructor to dispose of that dynamic data to prevent a *memory leak*. Another example is closing files for which the *file handle* is stored in the object.

The destructor is typically called without you noticing it. For instance, any statically created object is destroyed when the control flow leaves its scope.

Example:

Code:

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << 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 10.12. Write a class

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

used as

Code:

```
{
  HasInt v(5);
  v.set(6);
  v.set(-2);
}
```

Output
[object] destructexercise:

```
**** object created with 5 ****
**** object set to 6 ****
**** object set to -2 ****
**** object destroyed after 2 updates ****
```

The destructor is called when you throw an exception:

Code:

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << endl; };
  ~SomeObject() {
    cout <<  "calling the destructor"
         << endl; };
};
  /* ... */
  try {
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
    throw(1);
  } catch (...) {
    cout << "Exception caught" << endl;
  }
```

Output
[object] exceptdestruct:

```
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

10.4.9 'this' pointer

Inside an object, a *pointer* to the object is available as `this`:

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint;
  };
};
```

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

```
class someclass;
void somefunction(const someclass &c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(*this);
};
```

(Rare use of dereference star)

10.4.10 Static members

A static member acts as if it's shared between all objects.
(Note: C++17 syntax)

Code:

```
class myclass {
private:
  static inline int count=0;
public:
  myclass() { count++; };
  int create_count() { return count; };
};
  /* ... */
  myclass obj1,obj2;
  cout << "I have defined "
       << obj1.create_count()
       << " objects" << endl;
```

Output
[link] static17:

```
I have defined 2 objects
```

```
class myclass {
private:
  static int count;
public:
  myclass() { count++; };
  int create_count() { return count; };
};
  /* ... */
// in main program
int myclass::count=0;
```

10.5 Review question

Review 10.2. 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 10.3. Describe various ways to initialize the members of an object.

Chapter 11

Arrays

11.1 Introduction

An *array* is an indexed data structure that for each index stores an integer, floating point number, character, object, et cetera. In scientific applications, arrays often correspond to vectors and matrices, potentially of quite large size. (If you know about the Finite Element Method (FEM), you know that vectors can have sizes in the millions or beyond.)

In this chapter you will see the C++ `vector` construct, which implements the notion of an array of things, whether they be numbers, strings, objects.

> *C difference:* While C++ can use the C mechanisms for arrays, for almost all purposes it is better to use `vector`. In particular, this is a safer way to do dynamic allocation. The old mechanisms are briefly discussed in section 11.8.2.

11.1.1 Vector creation

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

Definition and/or initialization:

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

vector<type> name;
vector<type> name(size);
vector<type> name(size,init_value);
```

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` of the vector is up to you, and
- `size` is the (initial size of the vector). This is an integer, or more precisely, a `size_t` parameter.
- Initialize all elements to `init_value`.

Review 11.1. T/F?

- It is possible to write a valid C++ program where you define a variable `vector`.

11.1.2 Element access

There are two ways of accessing vector elements.

1. With the 'dot' notation that you know from structures and objects, you can write:

Code:

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

Output
[array] assignatfun:

```
4,8
```

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

Code:

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

Output
[array] assignbracket:

```
4,8
```

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.
- The same holds for indexing with square brackets.
- The *array index* (or *array subscript*) `i` starts numbering at zero.
- Therefore, if a vector has n elements, its last element has index `n-1`.

Have you wondered what happens if you access an array element outside the bounds of the vector?

```
vector<float> x(6); // size 6, index ranges 0..5
x.at(6) = 5.; // oops!
i = -2;
x[i] = 3; // very much oops!
```

Accessing an element outside the array bounds is hard to detect at compile time. Most likely, it will only be detected at runtime. There is now a difference in how the two accessing methods do *array bounds checking*.

1. Using the `at` method will always do a bounds test, and abort your program immediately if you access an element outside the array bounds. (See section 22.2.2 for how this works and how you can handle such errors.)
2. The bracket notation `a[i]` performs no bounds tests: it calculates a memory address based on the array 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, the lack of 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 array access out of bounds.
   - Your program may continue running, but giving wrong results, since reading from outside the array probably gives you meaningless values.

For now, it is best to use the `at` method throughout.

Indexing out of bounds can go undetected for a while:

Code:

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

Output
[array] segmentation:

```
element -2801 is 0
element -2802 is 0
element -2803 is 0
element -2804 is 1.41531e-43
element -2805 is 1.8519e+28
element -2806 is 7.31695e+28
element -2807 is 2.56297e+08
element -2808 is 2.65628e+20
/bin/sh: line 1: 83140 Segmentation fault: 11
```

Review 11.2. The following codes are not correct in some sense. How will this manifest itself?

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

```
vector<int> a(5);
a.at(6) = 1.;
```

11.1.3 Initialization

In some applications you will create an array, 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 array is called *array initialization*, and there is more than one way to do so.

First of all, you can easily set a vector to a constant:

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:

Code:

```
{
  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
```

11.2 Going over all array alements

If you need to consider all the elements in a vector, you typically use a `for` loop. There are various ways of doing this.

Conceptually, a `vector` can correspond to a set of things, and the fact that they are indexed is purely incidental, or it can correspond to an ordered set, and the index is essential. If your algorithm requires you to access all elements, it is important to think about which of these cases apply, since there are two different mechanism.

11.2.1 Ranging over a vector

First of all consider the cases where you consider the vector as a collection of elements, and the loop functions like a mathematical 'for all'.

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

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

Code:

```
vector<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] dynamicmax:

```
Max: 6 (should be 6)
```

(You can spell out the type of the vector element, but such type specifications can be complex. In that case, using `auto` is quite convenient.)

So-called *initializer lists* can also be used as a list denotation:

Code:

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

Output
[array] rangedenote:

```
2,3,5,7,9,
```

11.2.2 Ranging over the indices

If you actually need the index of the element, you can use a traditional `for` loop with loop variable.

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

```
for (int i= /* from first to last index */ )
  // statement about index i
```

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

Code:

```
int tmp_idx = 0;
int tmp_max = numbers.at(tmp_idx);
for (int i=0; i<numbers.size(); i++) {
  int v = numbers.at(i);
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << endl;
```

Output
[array] vecidxmax:

```
Max: 6.6 at index: 3
```

Exercise 11.1. 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 11.2. Find the element with maximum absolute value in a vector. Use:

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

Hint:

```
#include <cmath>
..
absx = abs(x);
```

Exercise 11.3. Find the location of the first negative element in a vector.

Which mechanism do you use?

Exercise 11.4. Check whether a vector is sorted.

11.2.3 Ranging by reference

Range-based loop indexing makes a copy of the vector element. If you want to alter the vector, use a reference:

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

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2) << endl;
```

Output
[array] vectorrangeref:

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

Exercise 11.5. If you do the prime numbers project, you can now do exercise 50.15.

Indexing in `while` loop and such:

```
x = a.at(i++); /* is */ x = a.at(i); i++;
y = b.at(++i); /* is */ i++; y = b.at(i);
```

Code:

```
vector<int> numbers{3,5,7,8,9,11};
int index{0};
while ( numbers[index++]%2==1 ) ;
cout << "The first even number"
     << " appears at index "
     << index << endl;
```

Output
[loop] plusplus:

```
The first even number appears at index 4
```

Exercise 11.6. Exercise: modify the preceeding code so that after the while loop `index` is the number of leading odd elements.

11.3 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 21 for the details). We could say that `vector<int>` is a new data type, pronounced 'vector-of-int', and you can make variables of that type.

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
     << vcopy.at(2) << endl;
```

Output
[array] vectorcopy:

```
3.5,7
```

11.3.1 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 11.7. 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.

What type of loop are you using?

11.3.2 Vectors are dynamic

A vector can be grown or shrunk after its creation. For instance, you can use the *push_back* method to add elements at the end:

Extend with *push_back*:

Code:

```
vector<int> mydata(5,2);
mydata.push_back(35);
cout << mydata.size() << endl;
cout << mydata[mydata.size()-1] << endl;
```

Output
[array] vectorend:

```
6
35
```

also *pop_back*, *insert*, *erase*.
Flexibility comes with a price.

It is tempting to use *push_back* to create a vector dynamically.

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

However, this dynamic resizing involves memory management, and maybe operating system functions. This will probably be inefficient. Therefore you should use such dynamic mechanisms only when strictly necessary. If you know the size, create a vector with that size. If the size is not precisely known but you have a reasonable upper bound, you can reserve the vector at that size:

```
vector<int> iarray;
iarray.reserve(100);
while ( ... )
  iarray.push_back( ... );
```

11.4 Vectors and functions

11.4.1 Pass vector to function

You can pass a vector to a function:

```
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.

Code:

```
void set0
  ( vector<float> v,float x )
{
  v.at(0) = x;
}
  /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << endl;
```

Output
[array] vectorpassnot:

```
3.5
```

Exercise 11.8. Revisit exercise 11.7 and introduce a function for computing the L_2 norm.

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

Code:

```
void set0
  ( vector<float> &v,float x )
{
  v.at(0) = x;
}
  /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << endl;
```

Output
[array] vectorpassref:

```
4.6
```

Passing a vector by reference means that the subprogram becomes able to alter it. To prevent that, pass it as a *const reference*; section 17.2.

11.4.2 Vector as function return

You can have a vector as return type of a function.
Example: this function creates a vector, with the first element set to the size:

Code:

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

Output
[array] vectorreturn:

```
x1 size: 10
zero element check: 10
```

Exercise 11.9. Write a function of one `int` argument n, which returns vector of length n, and which contains the first n squares.

Exercise 11.10. Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_vector(length);
vector<float> sorted = sort(values);
```

This creates a vector of random values of a specified length, and then makes a sorted copy of it.

Instead of making a sorted copy, sort in-place
(overwrite original data with sorted data):

```
int length = 10;
vector<float> values = random_vector(length);
sort(values); // the vector is now sorted
```

Find arguments for/against that approach.

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

(See section 7.7.1 for the random fuction.)

Exercise 11.11. 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?

11.5 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

- The constructor first creates `values` as a zero size vector,
- then it creates an anonymous vector of size `n`,
- and by assigning it, destroys the earlier created zero size vector.

This is somewhat inefficient, and the optimal solution is to create the vector as part of the *member initializer list*:

11.5.1 Timing

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

You can push elements into a vector:

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

If you allocate the vector statically, you can assign with `at`:

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

With subscript:

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

You can also use `new` to allocate*:

```
  int *stat = new int[LENGTH];
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

*Considered back practice. Do not use.

For `new`, see section 16.6.2. However, note that this mode of allocation is basically never needed.

Timings are partly predictable, partly surprising:

```
  Flexible time: 2.445
  Static at time: 1.177
  Static assign time: 0.334
  Static assign time to new: 0.467
```

The increased time for `new` is a mystery.

So do you use `at` for safety or `[]` for speed? Well, you could use `at` during development of the code, and insert

```
#define at(x) operator[](x)
```

for production.

11.6 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.

```
class namedvector {
private:
  string name;
  vector<int> values;
public:
  printable(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:

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

11.7 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.

11.7.1 Matrix as vector of vectors

Multi-dimensional is harder with vectors:

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

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.

```
class matrix {
private:
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
};
```

Exercise 11.12. Write `rows()` and `cols()` methods for this class that return the number of rows and columns respectively.

Exercise 11.13. Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

11.7.2 A better matrix class

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

```
class matrix {
private:
  std::vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n)
    : m(m),n(n),the_matrix(m*n) {};
  void set(int i,int j,double v) {
    the_matrix.at( i*n +j ) = v;
  };
  double get(int i,int j) {
    return the_matrix.at( i*n +j );
  };
  /* ... */
};
```

Exercise 11.14. 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 11.15. Write a method `element` of type `double&`, so that you can write

```
A.element(2,3) = 7.24;
```

11.8 Advanced topics

11.8.1 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 23.2.2 for details) a pointer to a vector element. Mirroring the index-loop convention of

```
for (int i=0; i<hi; i++)
  element = vec.at(i);
```

you can iterate:

```
for (auto elt_ptr=vec.begin(); elt_ptr!=vec.end(); ++elt_ptr)
  element = *elt_ptr;
```

Some remarks:

- This is one of the very few places where you need the asterisk in C++ for *dereferencing*; section 16.2. (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:

Code:

```
vector<int> mydata(5,2);
mydata.push_back(35);
cout << mydata.size() << endl;
cout << mydata[mydata.size()-1] << endl;
```

Output
[array] vectorend:

```
6
35
```

Code:

```
vector<int> mydata(5,2);
mydata.push_back(35);
cout << mydata.size() << endl;
cout << *( --mydata.end() ) << endl;
```

Output
[array] vectorenditerator:

```
6
35
```

11.8.2 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.

Code:

```
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << endl;
}
{
  int numbers[5]{5,4,3,2,1};
  numbers[3] = 21;
  cout << numbers[3] << endl;
}
```

Output
[array] staticinit:

```
2
21
```

This has the (minimal) advantage of not having the overhead of a class mechanism. On the other hand, it has a number of disadvantages:

- You can not query the size of an array by its name: you have to store that information separately in a variable.
- Passing such an array to a function is really passing the address of its first element, so it is always (sort of) by reference.

Range-based indexing works the same as with vectors:

Code:

```
int numbers[] = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should be 6)" << endl;
```

Output
[array] rangemax:

```
Max: 6 (should be 6)
```

11.8.2.1 C-style arrays and subprograms

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

```
void array_set( double ar[],int idx,double val) {
  ar[idx] = val;
}
  /* ... */
  array_set(array,1,3.5);
```

Exercise 11.16. 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.

11.8.2.2 Multi-dimensional arrays

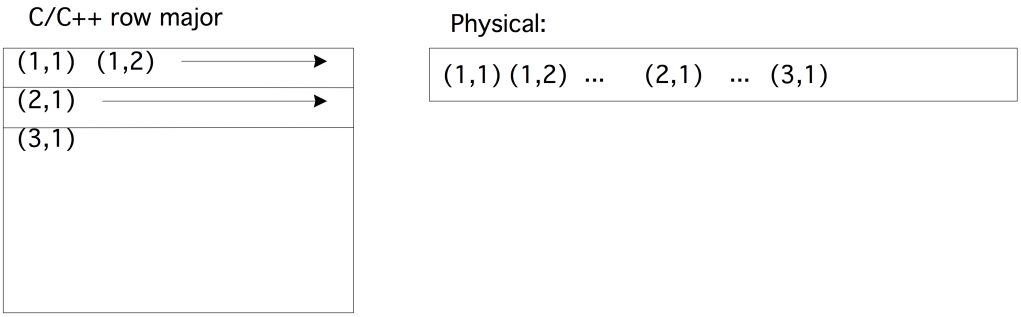
Multi-dimensional arrays can be declared and used with a simple extension of the prior syntax:

```
float matrix[15][25];

for (int i=0; i<15; i++)
  for (int j=0; j<25; j++)
    // something with matrix[i][j]
```

Passing a multi-dimensional array to a function, only the first dimension can be left unspecified:

```
void print12( int ar[][6] ) {
  cout << "Array[1][2]: " << ar[1][2] << endl;
  return;
}
  /* ... */
    int array[5][6];
    array[1][2] = 3;
    print12(array);
```



11.8.2.3 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:

```
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.

11.8.3 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 leak*s 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 15.

11.8.4 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.

11.8.5 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;
subx[1] = 5.; // same as: x[2] = 5.;
```

In C++ you can write

```
vector<double> x(N);
vector<double> subx( x.begin()+1,x.end() );
```

but that allocates new storage.

If you really want two vector-like objects to share data there is the `span` class, which is in the STL of C++20. Until this standard is available you can use the *Guideline Support Library (GSL)*, for instance implemented in https://github.com/Microsoft/GSL.

A span is little more than a pointer and a size, so it allows for the above use case. Also, it does not have the overhead of creating a whole new vector.

```
vector<double> v;
auto v_span = gsl::span<double>( v.data(),v.size() );
```

The `span` object has the same `at`, `data`, and `size` methods, and you can iterate over it, but it has no dynamic methods.

11.9 Exercises

Exercise 11.17. 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 11.18. Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

Pascal's triangle contains binomial coefficients:

```
Row    1:                      1
Row    2:                    1   1
Row    3:                  1   2   1
Row    4:                1   3   3   1
Row    5:              1   4   6   4   1
Row    6:            1   5  10  10   5   1
Row    7:          1   6  15  20  15   6   1
Row    8:        1   7  21  35  35  21   7   1
Row    9:      1   8  28  56  70  56  28   8   1
Row   10:    1   9  36  84 126 126  84  36   9   1
```

where

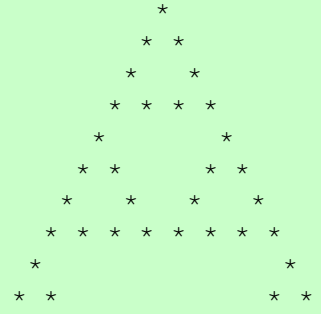
$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

The coefficients can easily be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} \end{cases}$$

Exercise 11.19.

- 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.
- 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 $n \times n$.

Exercise 11.20. Extend the Pascal exercise:
Optimize your code to use precisely enough space for the coefficients.

Exercise 11.21. 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 11.22. Put eight queens on a chessboard so that none threatens any other.

Exercise 11.23. From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

Chapter 12

Strings

12.1 Characters

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

Equivalent to (short) integer:

Code:

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

Output
[string] intchar:

```
x is at position 120
one further lies y
```

Also: `'x'-'a'` is distance `a--x`

Exercise 12.1. Write a program that accepts an integer $1 \cdots 26$ and prints the so-manieth letter of the alphabet.

Extend your program so that if the input is negative, it prints the minus-so-manieth uppercase letter of the alphabet.

12.2 Basic string stuff

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

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

(Do not use the C legacy mechanisms.)

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

Normally, quotes indicate the start and end of a string. So what if you want a string with quotes in it?

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

Code:

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

Output
[string] quote:

```
a b c
a "b" c
"a ""b """c
```

That last mechanism is referred to as a *raw string literal*.

Strings can be *concatenated*:

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
```

You can query the *size*:

Code:

```
string five_text{"fiver"};
cout << five_text.size() << endl;
```

Output
[string] stringsize:

```
5
```

or use subscripts:

```
cout << "The second character is <<"
     << txt[1] << ">>" << endl;
```

Same as ranging over vectors.
Range-based for:

Code:

```
cout << "By character: ";
for ( char c : abc )
  cout << c << " ";
cout << endl;
```

Output
[string] stringrange:

```
By character: a b c
```

Ranging by index:

Code:

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

Output
[string] stringindex:

```
By character: a b c
```

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

Code:

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

Output
[string] stringrangeset:

```
Shifted: bcd
```

```
for ( auto c : some_string)
  // do something with the character 'c'
```

Review 12.1. True or false?

- `'0'` is a valid value for a `char` variable
- `"0"` is a valid value for a `char` variable
- `"0"` is a valid value for a `string` variable
- `'a'+'b'` is a valid value for a `char` variable

Exercise 12.2. The oldest method of writing secret messages is the *Caesar cypher*. You would take an integer s and rotate every character of the text over that many positions:

$s \equiv 3$: "acdz" $\Rightarrow$ "dfgc".

Write a program that accepts an integer and a string, and display the original string rotated over that many positions.

Exercise 12.3. (this continues exercise 12.2)
If you find a message encrypted with the Caesar cipher, can you decrypt it? Take your inspiration from the *Sherlock Holmes* story 'The Adventure of the Dancing Men', where he uses the fact that 'e' is the most common letter.

Can you implement a more general letter permutation cipher, and break it with the 'dancing men' approach?

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

Code:

```
string five_chars;
cout << five_chars.size() << endl;
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 12.4. 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 12.5. Write a function to convert an integer to a string: the input `215` should give `two hundred fifteen`, et cetera.

Exercise 12.6. 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.

12.3 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*:

Like cout (including conversion from quantity to string), but to object, not to screen.

- Use the << operator to build it up; then
- use the *str* method to extract the string.

```
#include <sstream>
stringstream s;
s << "text" << 1.5;
cout << s.str() << endl;
```

12.4 Advanced topics

12.4.1 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 12.3.

- The *Boost* library has a *lexical cast*.

Additionally, in C++17 there is the *charconv* header with `to_chars` and `from_chars`. These are low level routines that do not throw exceptions, do not allocate, and are each other's inverses. The low level nature is for instance apparent in the fact that they work on character buffers (not null-terminated). Thus, they can be used to build more sophisticated tools on top of.

12.4.2 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.

Example:

12.5 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 13

Input/output

13.1 Formatted output

- `cout` uses default formatting
- Possible: pad a number, use limited precision, format as hex/base2, etc
- Many of these output modifiers need

```
#include <iomanip>
```

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

Code:

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

Output
[io] cunformat:

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

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

Code:

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

  // 'setw' applies only once:
  cout << "Width is 6:" << endl;
  cout << ">"
       << setw(6) << 1 << 2 << 3 << endl;
  cout << 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

Width is 6:
>     123
```

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

Code:

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

Output
[io] formatpad:

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

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

Instead of right alignment you can do left:

Code:

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

Output
[io] formatleft:

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Finally, you can print in different number bases than 10:

Code:

```
#include <iomanip>
using std::setbase;
using std::setfill;
  /* ... */
  cout << setbase(16)
       << setfill(' ');
  for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
      cout << i*16+j << " " ;
    cout << 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
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Exercise 13.1. Make the first line in the above output align better with the other lines:

```
 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

```
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

Hex output is useful for pointers (chapter 15):

Code:

```
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: 140732746264196
address of i, hex    : 0x7ffee5596e84
```

Back to decimal:

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

13.1.1 Floating point output

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

Code:

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
  /* ... */
  x = 1.234567;
  for (int i=0; i<10; i++) {
    cout << setprecision(4) << x << 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
```

(Notice the rounding)

Fixed precision applies to fractional part:

Code:

```
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
```

Combine width and precision:

Code:

```
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 13.2. Use integer output to print real numbers aligned on the decimal:

Code:

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

Combining width and precision:

Code:

```
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
```

13.1.2 Saving and restoring settings

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

cout.flags(old_settings);

int old_precision = cout.precision();

cout.precision(old_precision);
```

13.2 File output

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:

Code:

```
#include <fstream>
using std::ofstream;
  /* ... */
  ofstream file_out;
  file_out.open("fio_example.out");
  /* ... */
  file_out << number << endl;
  file_out.close();
```

Output
[io] fio:

```
echo 24 | ./fio ; \
  cat fio_example.out
A number please:
Written.
24
```

Compare: `cout` is a stream that has already been opened to your terminal 'file'.

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

Code:

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

Output
[io] fiobin:

```
echo 25 | ./fiobin ; \
  od fio_binary.out
A number please: Written.
0000000    000031  000000
0000004
```

13.3 Output your own classes

You have used statements like:

```
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.

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

```
class container {
  /* ... */
  int value() const {
  /* ... */
  };
  /* ... */
ostream &operator<<(ostream &os,const container &i) {
  os << "Container: " << i.value();
  return os;
};
  /* ... */
```

```
container eye(5);
cout << eye << endl;
```

13.4 Output buffering

In C, the way to get a newline in your output was to include the character `\n` in the output. This still works in C++, and at first it seems there is no difference with using `endl`. However, `endl` does more than breaking the output line: it performs a `std::flush`.

Output is usually not immediately written to screen or disc or printer: it is saved up in buffers. This can be for efficiency, because output a single character may have a large overhead, or it may be because the device is busy doing something else, and you don't want your program to hang waiting for the device to free up.

However, a problem with buffering is the output on the screen may lag behind the actual state of the program. In particular, if your program crashes before it prints a certain message, does it mean that it crashed before it got to that line, or does it mean that the message is hanging in a buffer.

This sort of output, that absolutely needs to be handled when the statement is called, is often called *logging* output. The fact that `endl` does a flush would mean that it would be good for logging output. However, it also flushes when not strictly necessary. In fact there is a better solution: `std::cerr` works just like `cout`, except it doesn't buffer the output.

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

13.5 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:

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

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

You can not use `cin` and `getline` in the same program.
More info: http://www.cplusplus.com/forum/articles/6046/.

13.5.1 File input

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

```
#include <fstream>
using std::ifstream;
  /* ... */
  ifstream input_file;
  input_file.open("fox.txt");
  string oneline;
  while (getline(input_file,oneline)) {
    cout << "Got line: <<" << oneline << ">>" << endl;
  }
```

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 13.3. 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*.

13.5.2 Input streams

Test, mostly for file streams: *is_eof is_open*

Chapter 14

References

14.1 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.

```
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:

```
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*.)

14.2 Pass by reference

If you use a mathematical style of subprograms, where some values go in, and a new entity comes out, in effect all the inputs can be copied. This style is called *functional programming*, and there is much to be said for it. For instance, it makes it possible for the compiler to reason about your program. The only thing you have to worry about is the cost of copying, if the inputs are of non-trivial size, such as arrays.

However, sometimes you want to alter the inputs, so instead of a copy you need a way of accessing the actual input object. That's what *references* are invented for: to allow a subprogram access to the actual input entity.

A bonus of using references is that you do not incur the cost of copying. So what if you want this efficiency, but your program is really functional in design? Then you can use a *const reference*: the

argument is passed by reference, but you indicate explicitly that the subprogram does not alter it, again allowing compiler optimizations.

A reference makes the function parameter a synonym of the argument.

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

```
class BigDude {
public:
   vector<double> array(5000000);
}

void f(BigDude d) {
  cout << d.array[0];
};

int main() {
   BigDude big;
   f(big); // whole thing is copied
```

Instead write:

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

Prevent changes:

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

14.3 Reference to class members

Here is the naive way of returning a class member:

```
class Object {
private:
  SomeType thing;
public:
  SomeType get_thing() {
    return thing; };
};
```

The problem here is that the return statement makes a copy of `thing`, which can be expensive. Instead, it is better to return the member by *reference*:

```
  SomeType &get_thing() {
    return thing; };
```

The problem with this solution is that the calling program can now alter the private member. To prevent that, use a *const reference*:

Code:

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc(); an_int.inc();
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
  /* Compiler error: an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
```

Output
[const] constref:

```
Contained int is now: 4
Contained int is now: 17
```

In the above example, the function giving a reference was used in the left-hand side of an assignment. If you would use it on the right-hand side, you would not get a reference. The result of an expression can not be a reference.

Let's again make a class where we can get a reference to the internals:

```
class myclass {
private:
  int stored{0};
public:
  myclass(int i) : stored(i) {};
  int &data() { return stored; };
};
```

Now we explore various ways of using that reference on the right-hand side:

Code:

```
myclass obj(5);
cout << "object data: "
     << obj.data() << endl;
int dcopy = obj.data();
dcopy++;
cout << "object data: "
     << obj.data() << endl;
int &dref = obj.data();
dref++;
cout << "object data: "
     << obj.data() << endl;
auto dauto = obj.data();
dauto++;
cout << "object data: "
     << obj.data() << endl;
auto &aref = obj.data();
aref++;
cout << "object data: "
     << obj.data() << endl;
```

Output
[func] rhsref:

```
object data: 5
object data: 5
object data: 6
object data: 6
object data: 7
```

(On the other hand, after `const auto &ref` the reference is not modifiable. This variant is useful if you want read-only access, without the cost of copying.)

You see that, despite the fact that the method `data` was defined as returning a reference, you still need to indicate whether the left-hand side is a reference.

See section 17.1 for the interaction between `const` and references.

14.4 Reference to array members

You can define various operator, such as `+-*/` arithmetic operators, to act on classes, with your own provided implementation; see section 10.4.5. 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 acess to arrays and/or vectors that are private to the object.

Suppose you have an object that contains an `int` array. You can return an element by defining the subscript (square bracket) operator for the class:

```
class vector10 {
private:
  int array[10];
public:
  /* ... */
  int operator()(int i) {
    return array[i];
  };
  int operator[](int i) {
    return array[i];
  };
};
  /* ... */
  vector10 v;
  cout << v(3) << endl;
  cout << v[2] << endl;
  /* compilation error: v(3) = -2; */
```

Note that `return array[i]` will return a copy of the array element, so it is not possible to write

```
myobject[5] = 6;
```

For this we need to return a reference to the array element:

```
  int& operator[](int i) {
    return array[i];
  };
};
  /* ... */
  cout << v[2] << endl;
  v[2] = -2;
  cout << v[2] << endl;
```

Your reason for wanting to return a reference could be to prevent the *copy of the return result* that is induced by the `return` statement. In this case, you may not want to be able to alter the object contents, so you can return a *const reference*:

```
  const int& operator[](int i) {
    return array[i];
  };
};
  /* ... */
  cout << v[2] << endl;
  /* compilation error: v[2] = -2; */
```

14.5 rvalue references

See the chapter about obscure stuff; section 24.5.3.

Chapter 15

Pointers

Pointers are an indirect way of associating an entity with a variable name. Consider for instance self-referential data structures:

> A list is a node that functions as the head, and which has a tail that is a list.

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

This does not work: would take infinite memory.

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

Pointer 'points' to the location of the tail.

This chapter will explain C++ *smart pointer*s, and give some uses for them.

15.1 The 'arrow' notation

- If `x` is object with member `y`:
  `x.y`
- If `xx` is pointer to object with member `y`:
  `xx->y`
- In class methods `this` is a pointer to the object, so:

```
class x {
  int y;
  x(int v) {
    this->y = v; }
}
```

- Arrow notation works with old-style pointers and new shared/unique pointers.

15.2 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?

Allocation and pointer in one:

```
shared_ptr<Obj> X =
    make_shared<Obj>( /* constructor args */ );
  // or:
auto X = make_shared<Obj>( /* args */ );

X->method_or_member;
```

Using shared pointers requires at the top of your file:

```
    #include <memory>
    using std::shared_ptr;
    using std::make_shared;
```

Code:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto &val() { return x; };
};

int main() {
  auto X = make_shared<HasX>(5);
  cout << X->val() << endl;
  X->val() = 6;
  cout << X->val() << endl;
```

Output
[pointer] pointx:

```
5
6
```

The constructor syntax is a little involved for vectors:

```
auto x = make_shared<vector<double>>(vector<double>{1.1,2.2});
```

15.2.1 Pointers and arrays

The prototypical example use of pointers is in linked lists. Let a class `Node` be given:

```
class Node {
private:
  int datavalue{0};
  shared_ptr<Node>
    tail_ptr{nullptr};
public:
  Node() {}
  Node(int value)
    : datavalue(value) {};
  bool has_next() {
    return tail_ptr!=nullptr; };
  void set_tail
      ( shared_ptr<Node> tail ) {
    tail_ptr = tail; };
```

```
std::string string() {
  stringstream strung;
  strung << datavalue;
  if (has_next()) {
    strung << "," << tail_ptr->string();
  }
  return strung.str();
};
```

Example use:

Code:

```
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
[tree] simple:

```
List <<23,45>> has length 2
```

Many operations on linked lists can be done recursively:

```
int Node::list_length() {
  if (!has_next()) return 1;
  else return 1+tail_ptr->list_length();
};
```

Exercise 15.1. Write a recursive *append* method that appends a node to the end of a list:

Code:

```
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 15.2. Write a recursive *insert* method that inserts a node in a list, such that the list stays sorted:

Code:

```
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>>
```

Exercise 15.3. For a more sophisticated approach to linked lists, do the exercises in section 62.1.2.

Exercise 15.4. If you are doing the prime numbers project (chapter 50) you can now do exercise 50.8.2.

15.2.2 Smart pointers versus address pointers

The oldstyle `&y` address pointer can not be made smart:

```
auto
  p = shared_ptr<HasY>( &y );
p->y = 3;
cout << "Pointer's y: "
     << p->y << 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

```
Obj *X;
*X = Obj( /* args */ );
```

There is a final way of creating a shared pointer where you cast an old-style `new` object to shared pointer

```
auto p = shared_ptr<Obj>( new Obj );
```

This is not the preferred mode of creation, but it can be useful in the case of *weak pointers*; section 15.5.4.

15.3 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.

```
// the variable 'array' doesn't exist
{
  // attach memory to 'array':
  double *array = new double[25];
  // do something with array
}
// the variable 'array' does not exist anymore
// but the memory is still reserved.
```

Shared and unique pointers do not have this problem: if they go out of scope, or are overwritten, the destructor on the object is called, thereby releasing any allocated memory.

An example.

We need a class with constructor and destructor tracing:

```
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

Let's create a pointer and overwrite it:

Code:

```
cout << "set pointer1"
     << endl;
auto thing_ptr1 =
  make_shared<thing>();
cout << "overwrite pointer"
     << endl;
thing_ptr1 = nullptr;
```

Output
[pointer] ptr1:

```
set pointer1
.. calling constructor
overwrite pointer
.. calling destructor
```

However, if a pointer is copied, there are two pointers to the same block of memory, and only when both disappear, or point elsewhere, is the object deallocated.

Code:

```
cout << "set pointer2" << 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
```

15.4 More about pointers

15.4.1 Get the pointed data

Most of the time, accessing the target of the pointer through the arrow notation is enough. However, if you actually want the object, you can get it with `get`. Note that this does not give you the pointed object, but a traditional pointer.

```
X->y;
// is the same as
X.get()->y;
// is the same as
( *X.get() ).y;
```

Code:

```
auto Y = make_shared<HasY>(5);
cout << Y->y << endl;
Y.get()->y = 6;
cout << ( *Y.get() ).y << endl;
```

Output
[pointer] pointy:

```
5
6
```

15.5 Advanced topics

15.5.1 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.

15.5.2 Base and derived pointers

Suppose you have base and derived classes:

```
class A {};
class B : public A {};
```

Just like you could assign a `B` object to an `A` variable:

```
B b_object;
A a_object = b_object;
```

is it possible to assign a `B` pointer to an `A` pointer?

The following construct makes this possible:

```
auto a_ptr = shared_pointer<A>( make_shared<B>() );
```

Note: this is better than

```
auto a_ptr = shared_pointer<A>( new B() );
```

Again a reason we don't need `new` anymore!

15.5.3 Shared pointer to 'this'

Inside an object method, the object is accessible as `this`. This is a pointer in the classical sense. So what if you want to refer to 'this' but you need a shared pointer?

For instance, suppose you're writing a linked list code, and your `node` class has a method `prepend_or_append` that gives a shared pointer to the new head of the list.

Your code would start something like this, handling the case where the new node is appended to the current:

```
shared_pointer<node> node::append
    ( shared_ptr<node> other ) {
  if (other->value>this->value) {
    this->tail = other;
```

But now you need to return this node, as a shared pointer. But `this` is a `node*`, not a `shared_ptr<node>`.

The solution here is that you can return

```
    return this->shared_from_this();
```

if you have defined your node class to inherit from what probably looks like magic:

```
class node : public enable_shared_from_this<node>
```

Note that you can only return a `shared_from_this` if already a valid shared pointer to that object exists.

15.5.4 Weak pointers

In addition to shared and unique pointers, which own an object, there is also `weak_ptr` which creates a *weak pointer*. This pointer type does not own, but at least it knows when it dangles.

```
weak_ptr<R> wp;
shared_ptr<R> sp( new R );
sp->call_member();
wp = sp;
// access through new shared pointer:
auto p = wp.lock();
if (p) {
  p->call_member();
}
if (!wp.expired()) { // not thread-safe!
  wp.lock()->call_member();
}
```

There is a subtlety with weak pointers and shared pointers. The call

```
auto sp = shared_ptr<Obj>( new Obj );
```

creates first the object, then the 'control block' that counts owners. On the other hand,

```
auto sp = make_shared<Obj>();
```

does a single allocation for object and control block. However, if you have a weak pointer to such a single object, it is possible that the object is destructed, but not de-allocated. On the other hand, using

```
auto sp  = shared_ptr<Obj>( new Obj );
```

creates the control block separately, so the pointed object can be destructed and de-allocated. Having a weak pointer to it means that only the control block needs to stick around.

15.5.5 Null pointer

In C there was a macro *NULL* that, only by convention, was used to indicate *null pointers*: pointers that do not point to anything. C++ has the `nullptr`, which is an object of type `std::nullptr_t`.

There are some scenarios where this is useful, for instance, with polymorphic functions:

```
void f(int);
void f(int*);
```

Calling `f(ptr)` where the point is `NULL`, the first function is called, whereas with `nullptr` the second is called.

Unfortunately, `dereferencing a` *nullptr* does not give an exception.

15.5.6 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 24.3). If you really need a pointer that does not *a priori* knows what it points to, use `std::any`, which is usually smart enough to call destructors when needed.

Code:

```
std::any a = 1;
cout << a.type().name() << ": " << std::any_cast<int>(a) << 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
```

15.5.7 Pointers to non-objects

In the introduction to this chapter we argued that many of the uses for pointers that existed in C have gone away in C++, and the main one left is the case where multiple objects share 'ownership' of some other object.

You can still make shared pointers to scalar data, for instance to an array of scalars. You then get the advantage of the memory management, but you do not get the `size` function and such that you would have if you'd used a `vector` object.

Here is an example of a pointer to a solitary double:

Code:

```
// shared pointer to allocated double
auto array = shared_ptr<double>( new double );
double *ptr = array.get();
array.get()[0] = 2.;
cout << ptr[0] << endl;
```

Output
[pointer] ptrdouble:

```
2
```

It is possible to initialize that double:

Code:

```
// shared pointer to initialized double
auto array = make_shared<double>(50);
double *ptr = array.get();
cout << ptr[0] << endl;
```

Output
[pointer] ptrdoubleinit:

```
50
```

15.6 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 12.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see 11.
- Traversing arrays and vectors can be done with ranges; section 11.2.
- 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 62.1.2.
- Objects on the heap.
- Use `nullptr` as a signal.

Chapter 16

C-style pointers and arrays

16.1 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 fargoing 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.

16.2 Pointers and addresses, C style

You have learned about variables, and maybe you have a mental concept of variables as 'named memory locations'. That is not too far of: while you are in the (dynamic) scope of a variable, it corresponds to a fixed memory location.

Exercise 16.1. When does a variable not always correspond to the same location in memory?

There is a mechanism of finding the actual address of a variable: you prefix its name by an ampersand. This address is integer-valued, but its range is actually greater than of the `int` type.

If you have an `int i`, then `&i` is the address of `i`.
An address is a (long) integer, denoting a memory address. Usually it is rendered in *hexadecimal* notation. C style:

Code:

```
int i;
printf("address of i: %ld\n",
       (long)(&i));
printf(" same in hex: %lx\n",
       (long)(&i));
```

Output
[pointer] printfpoint:

```
address of i: 140732908027540
 same in hex: 7ffeeefdbe94
```

and C++:

Code:

```
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: 140732790976132
address of i, hex    : 0x7ffee803ae84
```

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.

The type of '`&i`' is `int*`, pronounced 'int-star',
or more formally: 'pointer-to-int'.
You can create variables of this type:

```
int i;
int* addr = &i;
```

Now if you have have a pointer that refers to an int:

```
int i;
int *iaddr = &i;
```

you can use (for instance print) that pointer, which gives you the address of the variable. If you want the value of the variable that the pointer points to, you need to *dereference* it.

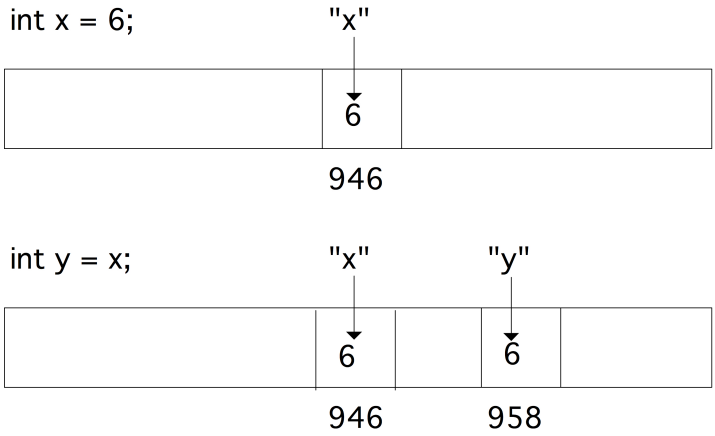
Using `*addr` 'dereferences' the pointer: gives the thing it points to;
the value of what is in the memory location.

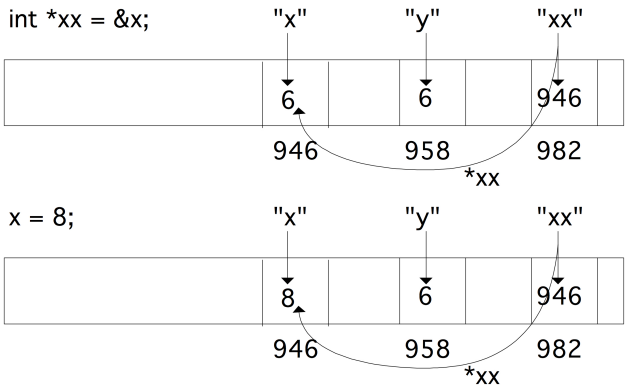
Code:

```
int i;
int* addr = &i;
i = 5;
cout << *addr << endl;
i = 6;
cout << *addr << endl;
```

Output
[pointer] cintpointer:

```
5
6
```





- *addr* is the address of *i*.
- You set *i* to 5; nothing changes about *addr*. This has the effect of writing 5 in the memory location of *i*.
- The first **cout** line dereferences *addr*, that is, looks up what is in that memory location.
- Next you change *i* to 6, that is, you write 6 in its memory location.
- The second **cout** looks in the same memory location as before, and now finds 6.

The syntax for declaring a pointer-to-sometype allows for a small variation, which indicates the two way you can interpret such a declaration.

Equivalent:

- **int*** *addr*: *addr* is an int-star, or
- **int** **addr*: **addr* is an int.

The notion **int*** *addr* is equivalent to **int** **addr*, and semantically they are also the same: you could say that *addr* is an int-star, or you could say that **addr* is an int.

16.3 Arrays and pointers

In section 11.8.2 you saw the treatment of static arrays in C++. Examples such as:

```
void array_set( double ar[],int idx,double val) {
  ar[idx] = val;
}
  /* ... */
  array_set(array,1,3.5);
```

show that, even though an parameters are normally passed by value, that is through copying, array parameters can be altered. The reason for this is that there is no actual array type, and what is passed is a pointer to the first element of the array. So arrays are still passed by value, just not the 'value of the array', but the value of its location.

So you could pass an array like this:

```
void array_set_star( double *ar,int idx,double val) {
  ar[idx] = val;
}
  /* ... */
  array_set_star(array,2,4.2);
```

Array and memory locations are largely the same:

Code:

```
double array[5] = {11,22,33,44,55};
double *addr_of_second = &(array[1]);
cout << *addr_of_second << endl;
array[1] = 7.77;
cout << *addr_of_second << endl;
```

Output
[pointer] arrayaddr:

```
22
7.77
```

You can dynamically reserve memory with **new**, which gives a something-star:

```
double *x;
x = new double[27];
```

The **new** operator is only for C++: in C you would use *malloc* to dynamically allocate memory. The above example would become:

```
double *x;
x = (double*) malloc( 27 * sizeof(double) );
```

Note that **new** takes the number of elements, and deduces the type (and therefore the number of bytes per element) from the context; *malloc* takes an argument that is the number of bytes. The **sizeof** operator then helps you in determining the number of bytes per element.

16.4 Pointer arithmetic

pointer arithmetic uses the size of the objects it points at:

```
double *addr_of_element = array;
cout << *addr_of_element;
addr_of_element = addr_of_element+1;
cout << *addr_of_element;
```

Increment add size of the array element, 4 or 8 bytes, not one!

Exercise 16.2. Write a subroutine that sets the i-th element of an array, but using pointer arithmetic: the routine should not contain any square brackets.

16.5 Multi-dimensional arrays

After

```
double x[10][20];
```

a row `x[3]` is a **double***, so is `x` a **double****?
Was it created as:

```
double **x = new double*[10];
for (int i=0; i<10; i++)
  x[i] = new double[20];
```

No: multi-d arrays are contiguous.

16.6 Parameter passing

C++ style functions that alter their arguments:

```
void inc(int &i) { i += 1; }
int main() {
  int i=1;
  inc(i);
  cout << i << endl;
  return 0;
}
```

In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```
void inc(int *i) { *i += 1; }
int main() {
  int i=1;
  inc(&i);
  cout << i << endl;
  return 0;
}
```

Now the function gets an argument that is a memory address: `i` is an int-star. It then increases `*i`, which is an int variable, by one.

Note how again there are two different uses of the ampersand character. While the compiler has no trouble distinguishing them, it is a little confusing to the programmer.

Exercise 16.3. Write another version of the *swap* function:

```
void swapij( /* something with i and j */ {
  /* your code */
}
int main() {
  int i=1,j=2;
  swapij( /* 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.

16.6.1 Allocation

In section 11.8.2 you learned how to create arrays that are local to a scope:

```
if ( something ) {
  double ar[25];
} else {
  double ar[26];
}
ar[0] = // there is no array!
```

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 16.6.2) allows you to allocate storage that transcends its scope:

```
double *array;
if (something) {
  array = new double[25];
} else {
  array = new double[26];
}
```

(Size in doubles, not in bytes as in 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*.

```
for (int i=0; i<large_num; i++) {
  double *array = new double[1000];
  // code that uses array
}
```

Every iteration reserves memory, which is never released: another *memory leak*.
Your code will run out of memory!

Memory allocated with **new** does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```
delete(array);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

No need for `malloc` or **new**

- Use **std**::`string` for character arrays, and
- **std**::`vector` for everything else.

No performance hit if you don't dynamically alter the size.

16.6.1.1 Malloc

The keywords **new** and **delete** are in the spirit of C style programming, but don't exist in C. Instead, you use `malloc`, which creates a memory area with a size expressed in bytes. Use the function **sizeof** to translate from types to bytes:

```
int n;
double *array;
array = malloc( n*sizeof(double) );
if (!array)
  // allocation failed!
```

16.6.1.2 Allocation in a function

The mechanism of creating memory, and assigning it to a 'star' variable can be used to allocate data in a function and return it from the function.

```
void make_array( double **a, int n ) {
  *a = new double[n];
}
int main() {
  double *array;
  make_array(&array,17);
}
```

Note that this requires a 'double-star' or 'star-star' argument:

- The variable `a` will contain an array, so it needs to be of type `double*`;
- but it needs to be passed by reference to the function, making the argument type `double**`;
- inside the function you then assign the new storage to the `double*` variable, which is `*a`.

Tricky, I know.

16.6.2 Use of `new`

Before doing this section, make sure you study section 16.3.

There is a dynamic allocation mechanism that is much inspired by memory management in C. Don't use this as your first choice.

Use of `new` uses the equivalence of array and reference.

```
void make_array( int **new_array, int length ) {
  *new_array = new int[length];
}
  /* ... */
  int *the_array;
  make_array(&the_array,10000);
```

Since this is not scoped, you have to free the memory yourself:

```
class with_array{
private:
  int *array;
  int array_length;
public:
  with_array(int size) {
    array_length = size;
    array = new int[size];
  };
  ~with_array() {
    delete array;
  };
};
  /* ... */
  with_array thing_with_array(12000);
```

Notice how you have to remember the array length yourself? This is all much easier by using a **std**::*vector*. See http://www.cplusplus.com/articles/37Mf92yv/.

The **new** mechanism is a cleaner variant of *malloc*, which was the dynamic allocation mechanism in C. Malloc is still available, but should not be used. There are even very few legitimate uses for **new**.

16.7 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:

```
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.

Chapter 17

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.

17.1 Const arguments

The use of const arguments is one way of protecting you against yourself. If an argument is conceptually supposed to stay constant, the compiler will catch it if you mistakenly try to change it.

Function arguments marked **`const`** can not be altered by the function code. The following segment gives a compilation error:

```
void f(const int i) {
  i++;
}
```

17.2 Const references

A more sophisticated use of const is the *const reference*:

```
void f( const int &i ) { .... }
```

This may look strange. After all, references, and the pass-by-reference mechanism, were introduced in section 7.4 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:

Code:

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc(); an_int.inc();
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
  /* Compiler error: an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
```

Output
[const] constref:

```
Contained int is now: 4
Contained int is now: 17
```

We can make visible the difference between pass by value and pass by const-reference if we define a class where the *copy constructor* explicitly reports itself:

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v << 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:

Code:

```
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);
```

Output
[const] constcopy:

```
Calling f with copy...
(calling copy constructor)
function with copy
Calling f with ref...
function with ref
... done
```

17.2.1 Const references in range-based loops

The same pass by value/reference issue comes up in range-based for loops. The syntax

```
for ( auto v : some_vector )
```

copies the vector elements to the `v` variable, whereas

```
for ( auto& v : some_vector )
```

makes a reference. To get the benefits of references (no copy cost) while avoiding the pitfalls (inadvertant changes), you can also use a const-reference here:

```
for ( const auto& v : some_vector )
```

17.3 Const methods

We can distinguish two types of methods: those that alter internal data members of the object, and those that don't. The ones that don't can be marked **const**. While this is in no way required, it contributes to a clean programming style:

Using **const** will catch mismatches between the prototype and definition of the method. For instance,

```
class Things {
  private:
  int var;
  public:
  f(int &ivar,int c) const {
    var += c; // typo: should be 'ivar'
  }
}
```

Here, the use of `var` was a typo, should have been `ivar`. Since the method is marked `const`, the compiler will generate an error.

It encourages a functional style, in the sense that it makes *side-effects* impossible:

```
class Things {
private:
  int i;
public:
  int get() const { return i; }
  int inc() { return i++; } // side-effect!
  void addto(int &thru) const { thru += i; }
}
```

17.4 Overloading on const

A const method and its non-const variant are different enough that you can use this for overloading.

Code:

```
class has_array {
private:
  vector<float> values;;
public:
  has_array(int l,float v)
    : values(vector<float>(l,v)) {};
  auto& at(int i) {
    cout << "var at" << 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 l; float v;
  cin >> l; cin >> v;
  has_array fives(l,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 17.1. 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?

17.5 Const and pointers

Let's declare a class `thing` to point to, and a class `has_thing` that contains a pointer to a `thing`.

```
class thing {
private:
  int i;
public:
  thing(int i) : i(i) {};
  void set_value(int ii) { i = ii; };
  auto value() const { return i; };
};

class has_thing {
private:
  shared_ptr<thing>
      thing_ptr{nullptr};
public:
  has_thing(int i)
    : thing_ptr
      (make_shared<thing>(i)) {};
  void print() const {
    cout << thing_ptr->value() << 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:

Code:

```
auto get_thing_ptr() const {
  return thing_ptr; };
/* ... */
  has_thing container(5);
  container.print();
  container.get_thing_ptr() =
    make_shared<thing>(6);
  container.print();
```

Output
[const] constpoint2:

```
5
5
```

If we return the pointer by reference we can change it. However, this requires the method not to be `const`. On the other hand, with the `const` method earlier we can change the object:

Code:

```
// Error: does not compile
// auto &get_thing_ptr() const {
auto &access_thing_ptr() {
  return thing_ptr; };
/* ... */
  has_thing container(5);
  container.print();
  container.access_thing_ptr() =
    make_shared<thing>(7);
  container.print();
  container.get_thing_ptr()->set_value(8);
  container.print();
```

Output
[const] constpoint3:

```
5
7
8
```

If you want to prevent the pointed object from being changed, you can declare the pointer as a `shared_ptr<const thing>`:

```
private:
  shared_ptr<const thing>
      const_thing{nullptr};
public:
  has_thing(int i,int j)
    : const_thing
      (make_shared<thing>(i+j)) {};
  auto get_const_ptr() const {
    return const_thing; };
  void crint() const {
    cout << const_thing->value() << endl; };
  /* ... */
    has_thing constainer(1,2);
    // Error: does not compile
    constainer.get_const_ptr()->set_value(9);
```

17.5.1 Old-style const pointers

For completeness, a section on const and pointers in 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;

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);

// 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;
```

17.6 Mutable

Typical class with non-const update methods, and const readout of some computed quantity:

```
class Stuff {
private:
  int i,j;
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; };
  void setj(int jnew) { j = jnew; };
  int result () const { return i+j; };
```

Attempt at caching the computation:

```
class Stuff {
private:
  int i,j;
  int cache;
  void compute_cache() { cache = i+j; };
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; compute_cache(); };
  void setj(int jnew) { j = jnew; compute_cache(); };
  int result () const { return cache; };
```

But what if setting happens way more often than getting?

```
class Stuff {
private:
  int i,j;
  int cache;
  bool cache_valid{false};
  void update_cache() {
    if (!cache_valid) {
      cache = i+j; cache_valid = true;
  };
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; cache_valid = false; };
  void setj(int jnew) { j = jnew; cache_valid = false; }
  int result () const {
    update_cache(); return cache; };
```

This does not compile, becuase `result` is const, but it calls a non-const function.

We can solve this by declaring the cache variables to be **mutable**. Then the methods that conceptually don't change the object can still stay **const**, even while altering the state of the object. (It is better not to use **const_cast**.)

```
class Stuff {
private:
  int i,j;
  mutable int cache;
  mutable bool cache_valid{false};
  void update_cache() const {
    if (!cache_valid) {
      cache = i+j; cache_valid = true;
  };
public:
  Stuff(int i,int j) : i(i),j(j) {};
  void seti(int inew) { i = inew; cache_valid = false; };
  void setj(int jnew) { j = jnew; cache_valid = false; }
  int result () const {
    update_cache(); return cache; };
```

17.7 Compile-time constants

Compilers have long been able to simplify expressions that only contain constants:

```
int i=5;
int j=i+4;
f(j)
```

Here the compiler will conclude that `j` is 9, and that's where the story stops. it also becomes possible to let `f(j)` be evaluated by the compiler, if the function `f` is simple enough. C++17 added several more variants of `constexpr` usage.

The **const** keyword can be used to indicate that a variable can not be changed:

```
const int i=5;
// DOES NOT COMPILE:
i += 2;
```

The combination **if** `constexpr` is useful with templates:

```
template <typename T>
auto get_value(T t) {
  if constexpr (std::is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

To declare a function to be constant, use `constexpr`. The standard example is:

```
constexpr double pi() {
  return 4.0 * atan(1.0); };
```

but also

```
constexpr int factor(int n) {
  return n <= 1 ? 1 : (n*fact(n-1));
}
```

(Recursion in C++11, loops and local variables in C++14.)

- Can use conditionals, if testing on constant data;
- can use loops, if number of iterations constant;
- C++20 can allocate memory, if size constant.

Chapter 18

Prototypes and header files

18.1 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

```
int tester(float);
```

A first use of prototypes is *forward declaration*.
Some people like defining functions after the main:

```
int f(int);
int main() {
  f(5);
};
int f(int i) {
  return i;
}
```

versus before:

```
int f(int i) {
  return i;
}
int main() {
  f(5);
};
```

You also need forward declaration for mutually recursive functions:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

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.

```
// file: def.cxx
int tester(float x) {
  .....
}
```

```
// file : main.cxx
int main() {
  int t = tester(...);
  return 0;
}
```

In this example a function `tester` is defined in a different file from the main program, so we need to tell `main` what the function looks like in order for the main program to be compilable:

```
// file : main.cxx
int tester(float);
int main() {
  int t = tester(...);
  return 0;
}
```

Splitting your code over multiple files and using *separate compilation* is good software engineering practice for a number of reasons.

1. If your code gets large, compiling only the necessary files cuts down on compilation time.
2. Your functions may be useful in other projects, by yourself or others, so you can reuse the code without cutting and pasting it between projects.
3. It makes it easier to search through a file without being distracted by unrelated bits of code.

(However, you would not do things like this in practice. See section 18.1.2 about header files.)

18.1.1 Separate compilation

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile
   ```
   icpc -c yourfile.cc
   ```
   which gives you a file `yourfile.o`, a so-called *object file*; and
2. Then you use the compiler as *linker* to give you the *executable file*:
   ```
   icpc -o yourprogram yourfile.o
   ```

In this particular example you may wonder what the big deal is. That will become clear if you have multiple source files: now you invoke the compile line for each source, and you link them only once.

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

At this point, you should learn about the *Make* tool for managing your programming project.

18.1.2 Header files

Even better than writing the prototype every time you need the function is to have a *header file*:

```
// file: def.h
int tester(float);
```

The header file gets included both in the definitions file and the main program:

```
// file: def.cxx
#include "def.h"
int tester(float x) {
  .....
}
```

```
// file : main.cxx
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

What happens if you leave out the `#include "def.h"` in both cases?

Having a header file is an important safety measure:

- Suppose you change your function definition, changing its return type:
- The compiler will complain when you compile the definitions file;
- So you change the 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 8 *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 9 *Header files were able to catch more errors in C than they do in C++. With polymorphism of functions, it is no longer an error to have*

```
// header.h
int somefunction(int);
```

and

```
#include "header.h"

int somefunction( float x ) { .... }
```

18.1.3 C and C++ headers

You have seen the following syntaxes for including header files:

```
#include <header.h>
#include "header.h"
```

The first is typically used for system files, with the second typically for files in your own project. There are some header files that come from the C standard library such as `math.h`; the idiomatic way of including them in C++ is

```
#include <cmath>
```

18.2 Prototypes for class methods

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file:

```
double something::dosomething( int i, char c ) {
  // do something with i,c
};
```

Data members, even private ones, need to be in the header file:

```
class something {
private:
  int localvar;
public:
  double somedo(vector);
};
```

Implementation file:

```
double something::somedo(vector v) {
   .... something with v ....
   .... something with localvar ....
};
```

Review 18.1. 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; };`

18.3 Header files and templates

The use of *templates* (see chapter 21) 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, 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:

```
template <typename T>
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:

```
template class foo<float>;
template class foo<double>;
```

18.4 Namespaces and header files

Namespaces (see chapter 19) 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.

18.5 Global variables and header files

If you have a variable that you want known everywhere, you can make it a *global variable*:

```
int processnumber;
void f() {
  ... processnumber ...
}
int main() {
  processnumber = // some system call
};
```

It is then defined in the main program and any functions defined in your program file.

Warning: it is tempting to define variables global but this is a dangerous practice.

If your program has multiple files, you should not put '`int processnumber`' in the other files, because that would create a new variable, that is only known to the functions in that file. Instead use:

```
extern int processnumber;
```

which says that the global variable `processnumber` is defined in some other file.

What happens if you put that variable in a *header file*? Since the *preprocessor* acts as if the header is textually inserted, this again leads to a separate global variable per file. The solution then is more complicated:

```
//file: header.h
#ifndef HEADER_H
#define HEADER_H
#ifndef EXTERN
#define EXTERN extern
#fi
EXTERN int processnumber
#fi

//file: aux.cc
#include "header.h"

//file: main.cc
#define EXTERN
#include "header.h"
```

(This sort of preprocessor magic is discussed in chapter 20.)

This also prevents recursive inclusion of header files.

18.6 Modules

The C++20 standard is taking a different approach to header files, through the introduction of *module*s. (Fortran90 already had this for a long time.) This largely dispenses with the need for header files included through **CPP!** (**CPP!**).

Chapter 19

Namespaces

19.1 Solving name conflicts

In section 11.3 you saw that the C++ STL comes with a *vector* class, that implements dynamic arrays. You say

```
std::vector<int> bunch_of_ints;
```

and you have an object that can store a bunch of ints. And if you use such vectors often, you can save yourself some typing by having

```
using namespace std;
```

somewhere high up in your file, and write

```
vector<int> bunch_of_ints;
```

in the rest of the file.

More safe:

```
using std::vector;
```

But what if you are writing a geometry package, which includes a vector class? Is there confusion with the STL vector class? There would be if it weren't for the phenomenon *namespace*, which acts as as disambiguating prefix for classes, functions, variables.

You have already seen namespaces in action when you wrote `std::vector`: the '`std`' is the name of the namespace.

You can make your own namespace by writing

```
namespace a_namespace {
  // definitions
  class an_object {
  };
```

so that you can write

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;
an_object myobject();
```

or

```
using a_namespace::an_object;
an_object myobject();
```

19.1.1 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:

```
#include "geolib.h"
using namespace geometry;
```

The header would contain the normal function and class headers, but now inside a named namespace:

```
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point() {};
    point( double x,double y );
    double x();
    double y();
  };
  class vector {
  private:
    point from,to;
  public:
    vector( point from,point to);
    double size();
  };
}
```

and the implementation file would have the implementations, in a namespace of the same name:

```
namespace geometry {
  point::point( double x,double y ) {
      xcoord = x; ycoord = y; };
  double point::x() { return xcoord; }; // 'accessor'
  double point::y() { return ycoord; };
  vector::vector( point from,point to) {
    this->from = from; this->to = to;
  };
  double vector::size() {
    double
      dx = to.x()-from.x(), dy = to.y()-from.y();
    return sqrt( dx*dx + dy*dy );
  };
}
```

19.2 Best practices

In this course we advocated pulling in functions explicitly:

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

It is also possible to use

```
#include <iostream>
using namespace std;
```

The problem with this is that it may pull in unwanted functions. For instance:

This compiles, but should not:

```
#include <iostream>
using namespace std;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

This gives an error:

```
#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`, you only do this in a source file, not in a header file. Anyone using the header would have no idea what functions are suddenly defined.

Chapter 20

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*.

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*. In the case of an `#include` statement, the preprocessing stage takes form of literaly inserting another file, here a *header file* into your source.

There are more sophisticated uses of the preprocessor.

20.1 Textual substitution

Suppose your program has a number of arrays and loop bounds that are all identical. To make sure the same number is used, you can create a variable, and pass that to routines as necessary.

```
void dosomething(int n) {
  for (int i=0; i<n; i++) ....
}

int main() {
  int n=100000;

  double array[n];

  dosomething(n);
```

You can also use `#define` to define a *preprocessor macro*:

```
#define N 100000
void dosomething() {
  for (int i=0; i<N; i++) ....
}

int main() {
  double array[N];

  dosomething();
```

It is traditional to use all uppercase for such macros.

20.1.1 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>>;
```

20.1.2 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:

```
#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 20.1. Write a macro that simulates 1-based indexing:

```
#define INDEX2D1BASED(i,j,n)  ????
...
double array[m,n];
for (int i=1; i<=m; i++)
  for (int j=n; j<=n; j++)
    array[ INDEX2D1BASED(i,j,n) ] = ...
```

20.2 Conditionals

There are a couple of *preprocessor conditions*.

20.2.1 Check on a value

The `#if` macro tests on nonzero. A common application is to temporarily remove code from compilation:

```
#if 0
  bunch of code that needs to
  be disabled
#endif
```

20.2.2 Check for macros

The `#ifdef` test tests for a macro being defined. Conversely, `#ifndef` tests for a macro not being defined. For instance,

```
#ifndef N
#define N 100
#endif
```

Why would a macro already be defined? Well you can do that on the compile line:

```
icpc -c file.cc -DN=500
```

Another application for this test is in preventing recursive inclusion of header files; see section 18.5.

20.2.3 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
```

20.3 Other pragmas

- Packing data structure without padding bytes by `#pack`

```
#pragma pack(push, 1)
// data structures
#pragma pack(pop)
```

Chapter 21

Templates

Sometimes you want a function or a class based on more than one different datatypes. For instance, in chapter 11 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.

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.

21.1 Templated functions

Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

Exercise 21.1. Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.

Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
float float_eps;
epsilon(float_eps);
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << endl;

double double_eps;
epsilon(double_eps);
cout << "Epsilon double: "
     << setw(10) << setprecision(4)
     << double_eps << endl;
```

Output
[template] eps:

```
Epsilon float: 1.0000e-07
Epsilon double: 1.0000e-15
```

21.2 Templated classes

The most common use of templates is probably to define templated classes. You have in fact seen this mechanism in action:

the STL contains in effect

```
template<typename T>
class vector {
private:
  // data definitions omitted
public:
  T at(int i) { /* return element i */ };
  int size() { /* return size of data */ };
  // much more
}
```

21.2.1 Specific implementation

```
template <typename T>
void f(T);
template <>
void f(char c) { /* code with c */ };
template <>
void f(double d) { /* code with d */ };
```

21.3 Concepts

Coming in the C++20 standard.

Templates can be too generic. For instance, one could write a templated `gcd` function

```
template <typename T>
T gcd( T a, T b ) {
  if (b==0) return a;
  else return gcd(b,a%b);
}
```

which will work for various integral types. To prevent it being applied to non-integral types, you can specify a *concept* to the type:

```
template<typename T>
concept bool Integral() {
  return std::is_integral<T>::value;
}
```

used as:

```
template <typename T>
requires Integral<T>{}
T gcd( T a, T b ) { /* ... */ }
```

or

```
template <Integral T>
T gcd( T a, T b ) { /* ... */ }
```

Abbreviated function templates:

```
Integral auto gcd
    ( Integral auto a, Integral auto b )
  { /* ... */ }
```

Chapter 22

Error handling

22.1 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:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
  a.at(i) = x; // runtime error
```

or undefined behaviour:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
  a[i] = x;
```

See further section 11.3.

Null pointers Using an uninitialized pointer is likely to crash your program:

```
Object *x;
if (false) x = new Object;
x->method();
```

Numerical errors such as division by zero will not crash your program, so catching them takes some care.

Guarding against errors.

- Check preconditions.
- Catch results.
- Check postconditions.

Error reporting:

- Message
- Total abort
- Exception

22.2 Mechanisms to support error handling and debugging

22.2.1 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:

```
#include <cassert>

float positive_outcome = f(x,y,z);
assert( positive_outcome>0 );
```

Since checking an assertion is a minor computation, you may want to disable it when you run your program in production by defining the `NDEBUG` macro:

```
#define NDEBUG
```

One way of doing that is passing it as a compiler flag:

```
icpc -DNDEBUG yourprog.cxx
```

Function return values

22.2.2 Exception handling

22.2.2.1 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 abort. You may see text on your screen

```
terminating with uncaught exception
```

The operative word here is *exception*: an exceptional situation that caused the normal progrm flow to have been aborted. We say that your program *throws* an *exception*.

Code:

```
vector<float> x(5);
x.at(5) = 3.14;
```

Output
[except] boundthrow:

```
make[3]: *** No rule to make target `run_boundt
```

Now you know that there is an error in your program, but you don't know where it occurs. You can find out, but trying to *catch the exception*.

Code:

```
vector<float> x(5);
for (int i=0; i<10; i++) {
  try {
    x.at(i) = 3.14;
  } catch (...) {
    cout << "Exception indexing at: "
         << i << endl;
    break;
  }
}
```

Output
[except] boundcatch:

```
make[3]: *** No rule to make target `run_boundc
```

22.2.2.2 Throw your own exceptions

Throwing an *exception* is one way of signalling an error or unexpected behaviour:

```
void do_something() {
  if ( oops )
    throw(5);
}
```

It now becomes possible to detect this unexpected behaviour by *catching* the exception:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

If you're doing the prime numbers project, you can now do exercise 50.12.

You can throw integers to indicate an error code, a string with an actual error message. You could even make an error class:

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

You can use exception inheritance!

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std::string c ) {
  // handle string exception
}
```

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

Exercise 22.1. Define the function

$$f(x) = x^3 - 19x^2 + 79x + 100$$

and evaluate $\sqrt{f(i)}$ for the integers $i = 0 \ldots 20$.

- First write the program naively, and print out the root. Where is $f(i)$ negative? What does your program print?
- You see that floating point errors such as the root of a negative number do not make your program crash or something like that. Alter your program to throw an exception if $f(i)$ is negative, catch the exception, and print an error message.
- Alter your program to test the output of the `sqrt` call, rather than its input. Use the function *isnan*

```
#include <cfenv>
using std::isnan;
```

and again throw an exception.

A *function try block* will catch exceptions, including in initializer lists of constructors.

```
f::f( int i )
  try : fbase(i) {
    // constructor body
  }
  catch (...) { // handle exception
  }
```

- Functions can define what exceptions they throw:

```
void func() throw( MyError, std::string );
void funk() throw();
```

- Predefined exceptions: *bad_alloc*, *bad_exception*, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section 10.4.8.
- There is an implicit `try/except` block around your main. You can replace the handler for that. See the *exception* header file.
- Keyword *noexcept*:

```
void f() noexcept { ... };
```

- There is no exception thrown when dereferencing a `nullptr`.

22.2.3 '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 native mechanism instead.

22.2.4 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.

22.2.5 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.

22.3 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 uninitialized data.

Chapter 23

Standard Template Library

The C++ language has a *Standard Template Library* (STL), which contains functionality that is considered standard, but that is actualy 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 11),
- strings (chapter 12),
- streams (chapter 13).

Using a template class typically involves

```
#include <something>
using std::function;
```

see section 19.1.

23.1 Complex numbers

Complex numbers use templating to set their precision.

```
#include <complex>
complex<float> f;
f.re = 1.; f.im = 2.;

complex<double> d(1.,3.);
```

Math operator like `+`, `*` are defined, as are math functions.

23.2 Containers

Vectors (section 11.3) and strings (chapter 12) are special cases of a STL *container*. Methods such as `push_back` and `insert` apply to all containers.

23.2.1 Maps: associative arrays

Arrays use an integer-valued index. Sometimes you may wish to use an index that is not ordered, or for which the ordering is not relevant. A common example is looking up information by string, such as finding the age of a person, given their name. This is sometimes called 'indexing by content', and the data structure that supports this is formally known as an *associative array*.

In C++ this is implemented through a `map`:

```
#include <map>
using std::map;
map<string,int> ages;
```

is set of pairs where the first item (which is used for indexing) is of type `string`, and the second item (which is found) is of type `int`.

A map is made by inserting the elements one-by-one:

```
#include <map>
using std::make_pair;
ages.insert(make_pair("Alice",29));
ages["Bob"] = 32;
```

You can range over a map:

```
for ( auto person : ages )
  cout << person.first << " has age " << person.second << endl;
```

A more elegant solution uses structured bindings (section 23.4):

```
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:

```
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 23.1. If you're doing the prime number project, you can now do the exercises in section 50.7.2.

23.2.2 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 11.8.1.

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

First, you can use them by themselves:

Code:

```
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 an operator, no a *pointer dereference*:

```
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;
```

23.2.2.1 Vector methods using iterators

Vector methods such as *erase* and *insert* use these iterators:

Methods `erase` and `insert` indicate their range with begin/end iterators

Code:

```
vector<int> v{1,3,5,7,9};
cout << "Vector: ";
for ( auto e : v ) cout << e << " ";
cout << endl;
auto first = v.begin();
first++;
auto last = v.end();
last--;
v.erase(first,last);
cout << "Erased: ";
for ( auto e : v ) cout << e << " ";
cout << endl;
```

Output
[stl] erase:

```
Vector: 1 3 5 7 9
Erased: 1 9
```

Note: end is exclusive.

23.2.2.2 Indexing and iterating

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.

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
```

23.2.2.3 Algorithms using iterators

Numerical *reductions* can be applied using iterators:

Default is sum reduction:

Code:

```
vector<int> v{1,3,5,7};
auto first = v.begin();
auto last  = v.end();
auto sum = accumulate(first,last,0);
cout << "sum: " << sum << endl;
```

Output
[stl] accumulate:

```
sum: 16
```

Supply multiply operator:

Code:

```
vector<int> v{1,3,5,7};
auto first = v.begin();
auto last  = v.end();
first++; last--;
auto product =
  accumulate
    (first,last,2,multiplies<>());
cout << "product: " << product << endl;
```

Output
[stl] product:

```
product: 30
```

Specific for the max reduction is `max_element`. This can be called without a comparator (for numerical max), or with a comparator for general maximum operations.
Example: maximum relative deviation from a quantity:

```
max_element(myvalue.begin(),myvalue.end(),
    [my_sum_of_squares] (double x,double y) -> bool {
        return fabs( (my_sum_of_squares-x)/x ) < fabs( (my_sum_of_squares-y)/y ); }
);
```

23.2.2.4 Forming sub-arrays

Iterators can be used to construct a `vector`. This can for instance be used to create a *subvector*:

Code:

```
  vector<int> vec{11,22,33,44,55,66};
  auto second = vec.begin(); second++;
  auto before = vec.end(); before--;
  //    vector<int> sub(second,before);
  vector<int> sub(vec.data()+1,vec.data()+vec.size()-1);
  cout << "no first and last: ";
  for ( auto i : sub ) cout << i << ", ";
  cout << endl;
  vec.at(1) = 222;
  cout << "did we get a change in the sub vector? " << sub.at(0) << endl;
/* ... */
  vector<int> vec{11,22,33,44,55,66};
  auto second = vec.begin(); second++;
  auto before = vec.end(); before--;
  //    vector<int> sub(second,before);
  vector<int> sub; sub.assign(second,before);
  cout << "vector at " << (long)vec.data() << endl;
  cout << "sub at " << (long)sub.data() << endl;

  cout << "no first and last: ";
  for ( auto i : sub ) cout << i << ", ";
  cout << endl;
  vec.at(1) = 222;
  cout << "did we get a change in the sub vector? " << sub.at(0) << endl;
```

Output
[iter] subvector:

```
no first and last: 22, 33, 44, 55,
did we get a change in the sub vector? 22
vector at 140380448228928
sub at 140380448238560
no first and last: 22, 33, 44, 55,
did we get a change in the sub vector? 22
```

23.3 Regular expression

The header `regex` gives C++ the functionality for *regular expression* matching. For instance, `regex_match` returns whether or not a string matches an expression exactly:

Code:

```
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
[regexp] regexp:

```
Looks like a name:
Victor: yes
aDam: no
DoD: no
```

(Note that the regex matches substrings, but `regex_match` only returns true for a match on the whole string.

For finding substrings, use `regex_search`:

- the function itself evaluates to a `bool`;
- there is an optional return parameter of type *smatch* ('string match') with information about the match.

The *smatch* object has these methods:

- `smatch::position` states where the expression was matched,

- while `smatch::str` returns the string that was matched.
- `smatch::prefix` has the string preceeding the match; with `smatch::prefix().size()` you get the number of characters preceeding the match, that is, the location of the match.

Code:

```
{
  auto findthe = regex("the");
  auto found = regex_search
    ( sentence,findthe );
  assert( found==true );
  cout << "Found <<the>>" << endl;
}
{
  smatch match;
  auto findthx = regex("o[^o]+o");
  auto found = regex_search
    ( sentence, match ,findthx );
  assert( found==true );
  cout << "Found <<o[^o]+o>>"
       << " at " << match.position(0)
       << " as <<" << match.str(0) << ">>"
       << " preceeded by <<" << match.prefix() << ">>"
       << endl;
}
```

Output

```
[regexp] search:
Found <<the>>
Found <<o[^o]+o>> at 12 as <<own fo>> preceeded
```

23.3.1 Regular expression syntax

C++ uses a variant of the International regular expression syntax. http://ecma-international.org/ecma-262/5.1/#sec-15.10. Consult that document for escape characters and more.

If your regular expression is getting too complicated with escape characters and such, consider using the *raw string literal* construct.

23.4 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*: an entity that comprises several components, possibly of different type, and which unlike a **struct** you do not need to define beforehand.

```
std::tuple<int,double> id = \
    std::make_tuple<int,double>(3,5.12);
double result = std::get<1>(id);
std::get<0>(id) += 1;
```

This does not look terribly elegant. Fortunately, C++17 can use denotations and the `auto` keyword to make this considerably shorter. Consider the case of a function that returns a tuple. You could use `auto` to deduce the return type:

```
auto maybe_root1(float x) {
  if (x<0)
    return make_tuple
      <bool,float>(false,-1);
  else
    return make_tuple
      <bool,float>(true,sqrt(x));
};
```

but more interestingly, you can use a *tuple denotation*:

```
tuple<bool,float>
    maybe_root2(float x) {
  if (x<0)
    return {false,-1};
  else
    return {true,sqrt(x)};
};
```

The calling code is particularly elegant:

Code:

```
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 23.2.1):

```
for ( const auto &[key,value] : mymap ) ....
```

23.5 Union-like stuff: tuples, optionals, variants

There are cases where you need a value that is one type or another, for instance, a number if a computation succeeded, and an error indicator if not.

The simplest solution is to have a function that returns both a bool and a number:

```
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = sqrt(x);
  return true;
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
      cout << "Root is " << x << endl;
    else
      cout << "could not take root of " << x << endl;
```

We will now consider some more idiomatically C++17 solutions to this.

23.5.1 Tuples

Using tuples (section 23.4) the solution to the above 'a number or an error' now becomes:

```
tuple<bool,float> RootAndValid(float x) {
  if (x<0)
    return {false,x};
  else
    return {true,sqrt(x)};
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto [ok,root] = RootAndValid(x) ; ok )
      cout << "Root is " << root << endl;
    else
      cout << "could not take root of " << x << endl;
```

23.5.2 Optional

The most elegant solution to 'a number or an error' is to have a single quantity that you can query whether it's valid. For this, the C++17 standard introduced the concept of a *nullable type*: a type that can somehow convey that it's empty.

Here we discuss `std::optional`.

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

- You can create an optional quantity with a function that returns either a value of the indicated type, or `{}`, which is a synonym for `nullopt`.

  ```
  optional<float> f {
    if (something) return 3.14;
    else return {};
  }
  ```

- You can test whether the optional quantity has a quantity with `has_value`, in which case you can extract the quantity with `value`:

  ```
  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.

23.5.3 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:

```
variant<int,double,string> union_ids;
```

We can use the `index` function to see what variant is used (0,1,2 in this case) and `get` the value accordingly:

```
union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
  cout << "Double case: " << std::get<double>(union_ids) << endl;
}
```

Getting the wrong variant leads to a `bad_variant_access` exception:

```
union_ids = 17;
cout << "Using option " << union_ids.index() << ": " << get<int>(union_ids) << endl;
try {
  cout << "Get as double: " << get<double>(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:

```
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 23.2. 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.

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`.

23.5.4 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`:

```
std::any a{12};
std::any_cast<int>(a); // succeeds
std::any_cast<string>(a); // fails
```

23.6 Algorithms

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.

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

Code:

```
vector<int> integers{1,2,3,5,7,10};
auto any_even = any_of
  ( integers.begin(),integers.end(),
    [=] (int i) -> bool { return i%2==0; }
    );
if (any_even) cout << "there was an even" << endl;
else          cout << "none were even" << endl;
```

Output
[range] anyof:

```
there was an even
```

An example of `for_each`:

Code:

```
#include <algorithm>
using std::for_each;
  /* ... */
  vector<int> ints{3,4,5,6,7};
  for_each
    ( ints.begin(),ints.end(),
      [] (int x) -> void {
        if (x%2==0)
          cout << x << endl;
      } );
```

Output
[stl] printeach:

```
4
6
```

See section 23.2.2 for iterators, in particular section 23.2.2.3 for algorithms with iterators, and section 24.3 for the use of lambda expressions.

23.7 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.

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;

cout << numeric_limits<long>::max();
```

Code:

```
cout << "Signed int: "
     << numeric_limits<int>::min() << " "
     << numeric_limits<int>::max()
     << endl;
cout << "Unsigned    "
     << numeric_limits<unsigned int>::min() << " "
     << numeric_limits<unsigned int>::max()
     << endl;
cout << "Single      "
     << numeric_limits<float>::min() << " "
     << numeric_limits<float>::max()
     << endl;
cout << "Double      "
     << numeric_limits<double>::min() << " "
     << numeric_limits<double>::max()
     << endl;
```

Output
[stl] limits:

```
Signed int: -2147483648 2147483647
Unsigned    0 4294967295
Single      1.17549e-38 3.40282e+38
Double      2.22507e-308 1.79769e+308
```

- *min* is the smallest positive number;
- use *lowest* for 'most negative'.
- There is an *epsilon* function for machine precision:

Code:

```
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
Double lowest -1.79769e+308 and epsilon 2.22045
```

Exercise 23.3. Write a program to discover what the maximal n is so that $n!$, that is, n-factorial, can be represented in an `int`, `long`, or `long long`. Can you write this as a templated function?

Operations such as dividing by zero lead to floating point numbers that do not have a valid value. For efficiency of computation, the processor will compute with these as if they are any other floating point number.

There are tests for detecting whether a number is *Inf* or *NaN*. However, using these may slow a computation down.

The functions *isinf* and *isnan* are defined for the floating point types (`float`, `double`, `long double`), returning a `bool`.

```
#include <math.h>
isnan(-1.0/0.0);   // false
isnan(sqrt(-1.0)); // true
isinf(-1.0/0.0);   // true
isinf(sqrt(-1.0)); // false
```

23.7.1 Storage

- An `int` used to be 16 bits in the olden days, but these days 32 bits is more common.
- A `long int`**`long`** has at least the range of an `int`, and at least 32 bits.
- A `long long int`**`long long`** has at least the range of a `long`, and at least 64 bits.

23.8 Random numbers

The STL has a *random number generator* that is more general and more flexible than the C version (section 7.7.1).

- There are several generators that give uniformly distributed numbers;
- then there are distributions that translate this to non-uniform or discrete distributions.

First you declare an engine; later this will be transformed into a distribution:

```
std::default_random_engine generator;
```

This generator will start at the same value everytime. You can seed it:

```
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:

```
std::uniform_real_distribution<float> distribution(0.,1.);
```

A roll of the dice would result from:

```
std::uniform_int_distribution<int> distribution(1,6);
```

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// set the default random number generator
std::default_random_engine generator{r()};

// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

cout << "first rand: " << distribution(generator) << endl;
```

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

Another distribution is the *Poisson distribution*:

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

23.9 Time

23.10 Ranges

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

We need to introduced two new concepts.

A `range` is an iteratable object. The containers of the pre-17 STL are ranges, but some new ones have been added.

First of all, ranges provide a clearer syntax:

```
vector data{2,3,1};
sort( begin(data),end(data) ); // open to accidents
ranges::sort(data);
```

A `view` is somewhat similar to a range, in the sense that you can iterate over it. The difference is that, unlike for instance a `vector`, a view is not a completely formed object. A view doesn't own any data, and any elements you view in it get formed as you iterate over it. This is sometimes called *lazy evaluation* or *lazy execution*. Stated differently, its elements are constructed as they are being requested by the iteration over the view.

Views are composable: you can take one view, and pipe it into another one. If you need the resulting object, rather than the elements as a stream, you can call `to_vector`.

First two simple examples of views:

1. one formed by `transform`, which applies a function to each element of the range or view in sequence;
2. one formed by `filter`, which only yields those elements that satisfy some boolean test.

(We use an auxiliary function to turn a vector into a string.)

Code:

```
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original vector: "
     << vector_as_string(v) << 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:

```
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; } );
cout << "Times two over five: "
     << vector_as_string
          ( times_two_over_five | ranges::to_vector )
     << endl;
```

Output [range] ft2:

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

Exercise 23.4. 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:

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

23.10.1 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.

```
auto result { view::ints(10) // VLE ???
    | views::filter( [] ( const auto& value ) {
                        return value%2==0; } )
    | views::take(10) };
```

Chapter 24

Obscure stuff

24.1 Auto

This is not actually obscure, but it intersects many other topics, so we put it here for now.

24.1.1 Declarations

Sometimes the type of a variable is obvious:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
                ( 20, new myclass(1.3) );
```

(Pointer to vector of 20 shared pointers to `myclass`, initialized with unique instances.) You can write this as:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
            ( 20, new myclass(1.3) );
```

Return type can be deduced in C++17:

```
auto equal(int i,int j) {
  return i==j;
};
```

Return type of methods can be deduced in C++17:

```
class A {
private: float data;
public:
  A(float i) : data(i) {};
  auto &access() {
    return data; };
  void print() {
    cout << "data: " << data << endl; };
};
```

`auto` discards references and such:

Code:

```
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output [auto] plainget:

```
data: 5.7
```

Combine `auto` and references:

Code:

```
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output [auto] refget:

```
data: 6.7
```

For good measure:

Code:

```
A my_a(5.7);
const auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output [auto] constrefget:

```
make[3]: *** No rule to make target `error_constrefget.
```

24.1.2 Iterating

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
  s += copy_of_int;
for ( auto &ref_to_int : myvector )
  ref_to_int = s;
for ( const auto& copy_of_thing : myvector )
  s += copy_of_thing.f();
```

is actually short for:

```
for ( std::vector<int>
      iterator it=myvector.begin() ;
      it!=myvector.end() ; ++it )
  s += *it ; // note the deref
```

Range iterators can be used with anything that is iteratable (vector, map, your own classes!)

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

```
    auto first = myarray.begin();
    first += 2;
    auto last  = myarray.end();
    last  -= 2-;
    myarray.erase(first,last);
```

24.1.3 `decltype`: declared type

There are places where you want the compiler to deduce the type of a variable, but where this is not immediately possible. Suppose that in

```
auto v = some_object.get_stuff();
f(v);
```

you want to put a `try ... catch` block around just the creation of `v`. This does not work:

```
try { auto v = some_object.get_stuff();
} catch (...) {}
f(v);
```

because the `try` block is a scope. It also doesn't work to write

```
auto v;
try { v = some_object.get_stuff();
} catch (...) {}
f(v);
```

because there is no indication what type `v` is created with.

Instead, it is possible to query the type of the expression that creates `v` with `decltype`:

```
decltype(some_object.get_stuff()) v;
try { auto v = some_objects.get_stuff();
} catch (...) {}
f(v);
```

11.8.3

24.2 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 iteratable like this.)

This is not magic: it is possible to iterate over any class: a *class* is *iteratable* that has a number of conditions satisfied.

The class needs to have:

- a method `begin` with prototype

  ```
  iteratableClass iteratableClass::begin()
  ```

  That gives an object in the initial state, which we will call the 'iterator object'; likewise
- a method `end`

  ```
  iteratableClass iteratableClass::end()}
  ```

that gives an object in the final state; furthermore you need

- an increment operator

```
void iteratableClass::operator++()
```

that advances the iterator object to the next state;

- a test

```
bool iteratableClass::operator!=(const iteratableClass&)
```

to determine whether the iteration can continue; finally

- a dereference operator

```
iteratableClass::operator*()
```

that takes the iterator object and returns its state.

Let's make a class, called a `bag`, that models a set of integers, and we want to enumerate them. For simplicity sake we will make a set of contiguous integers:

```
class bag {
  // basic data
private:
  int first,last;
public:
  bag(int first,int last) : first(first),last(last) {};
```

When you create an iterator object it will be copy of the object you are iterating over, except that it remembers how far it has searched:

```
private:
  int seek{0};
```

The `begin` method gives a `bag` with the seek parameter initialized:

```
public:
  bag &begin()  {
    seek = first; return *this;
  };
  bag end() {
    seek = last; return *this;
  };
```

These routines are public because they are (implicitly) called by the client code.

The termination test method is called on the iterator, comparing it to the `end` object:

```
bool operator!=( const bag &test ) const {
  return seek<=test.last;
};
```

Finally, we need the increment method and the dereference. Both access the `seek` member:

```
void operator++() { seek++; };
int operator*() { return seek; };
```

We can iterate over our own class:

Code:

```
bag digits(0,9);

bool find3{false};
for ( auto seek : digits )
  find3 = find3 || (seek==3);
cout << "found 3: " << boolalpha
     << find3 << endl;

bool find15{false};
for ( auto seek : digits )
  find15 = find15 || (seek==15);
cout << "found 15: " << boolalpha
     << find15 << endl;
```

Output
[loop] bagfind:

```
found 3: true
found 15: false
```

(for this particular case, use `std::any_of`)

If we add a method `has` to the class:

```
bool has(int tst) {
  for (auto seek : *this )
    if (seek==tst) return true;
  return false;
};
```

we can call this:

```
cout << "f3: " << digits.has(3) << 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 24.1. You can now do exercise 50.19, implementing a prime number generator with this mechanism.

If you think you understand `const`, consider that the `has` method is conceptually `cost`. But if you add that keyword, the compiler will complain about that use of `*this`, since it is altered through the `begin` method.

Exercise 24.2. Find a way to make `has` a `const` method.

24.3 Lambdas

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

```
[capture] ( inputs ) -> outtype { definition };
```

Example:

```
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Store lambda in a variable:

```
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << endl;
```

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

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

and fix the exponent:

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

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

The lambda notation can be even shorter if the output is void and there are no parameters:

```
[] { /* computation */ }
```

24.3.1 Lambda members of classes

Storing a lambda in a class is hard because it has unique type. Solution: use `std::function`.

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

Code:

```
cout << "Give a divisor: "; cin >> divisor; cout << endl;
cout << ".. using " << divisor << endl;
SelectedInts multiples
  ( [divisor] (int i) -> bool {
      return i%divisor==0; } );
for (int i=1; i<50; i++)
  multiples.add(i);
```

Output [func] lambdafun:

```
Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```

Exercise 24.3. Refer to 7.16 for background, and note that finding x such that $f(x) = a$ is equivalent to applying Newton to $f(x) - a$.

Implement a class `valuefinder` and its `double find(double)` method.

```
class valuefinder {
private:
  function< double(double) >
      f,fprime;
  double tolerance{.00001};
public:
  valuefinder
  ( function< double(double) > f,
    function< double(double) > fprime )
    : f(f),fprime(fprime) {};
```

used as

```
double root = newton_root.find(number);
```

Exercise 24.4. Can you write a derived class `rootfinder` used as

```
squarefinder newton_root;
double root = newton_root.find(number);
```

24.3.2 Generic lambdas

The `auto` keyword can be used, almost abused, for *generic lambda*s:

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

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

24.3.3 Lambda in algorithm

The `algorithm` header 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.

24.3.4 Lambda variables by reference

Normally, captured variables are copied by value. You can capture them by reference by prefixing an ampersand to them

```
auto maybeinc =
  [&count] (int i) mutable {
      if (f(i)) count++; }
```

The lambda expression also needs to be marked **mutable** because by default it is a const.

Code:

```
vector<int> moreints{8,9,10,11,12};
int count{0};
for_each
  ( moreints.begin(),moreints.end(),
    [&count] (int x) mutable {
      if (x%2==0)
        count++;
    } );
cout << "number of even: " << count << endl;
```

Output
[stl] counteach:

```
awk: illegal statement
 input record number 6, file
 source line number 1
make[3]: *** [run_counteach] Error 2
```

Lambdas without captures can be converted to a function pointer.

24.4 Casts

In C++, constants and variables have clear types. For cases where you want to force the type to be something else, there is the *cast* mechanism. With a cast you tell the compiler: treat this thing as such-and-such a type, no matter how it was defined.

In C, there was only one casting mechanism:

```
sometype x;
othertype y = (othertype)x;
```

This mechanism is still available as the **reinterpret_cast**, which does 'take this byte and pretend it is the following type':

```
sometype x;
auto y = reinterpret_cast<othertype>(x);
```

The inheritance mechanism necessitates another casting mechanism. An object from a derived class contains in it all the information of the base class. It is easy enough to take a pointer to the derived class, the bigger object, and cast it to a pointer to the base object. The other way is harder.

Consider:

```
class Base {};
class Derived : public Base {};
Base *dobject = new Derived;
```

Can we now cast dobject to a pointer-to-derived ?

- **static_cast** assumes that you know what you are doing, and it moves the pointer regardless.

- `dynamic_cast` checks whether `dobject` was actually of class `Derived` before it moves the pointer, and returns `nullptr` otherwise.

Remark 10 *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 easily recognized.*

Further reading https://www.quora.com/How-do-you-explain-the-differences-among-static_cast-reinterpret_cast-const_cast-and-dynamic_cast-to-a-new-C++-programmer/answer/Brian-Bi

24.4.1 Static cast

One use of casting is to convert to constants to a 'larger' type. For instance, allocation does not use integers but *size_t*.

```
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
cout << "overflow: " << overflow << 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] intlong:

```
long number:      100000000000
assigned to int: 1215752192
```

There are no runtime tests on static casting.

Static casts are a good way of casting back void pointers to what they were originally.

24.4.2 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: missing snippet polymain The function can discover what derived class the base pointer refers to:

```
void f( Base *obj ) {
  Derived *der =
    dynamic_cast<Derived*>(obj);
  if (der==nullptr)
    cout << "Could not be cast to Derived"
         << endl;
  else
    der->print();
};
  /* ... */
    Base *object = new Derived();
    f(object);
    Base *nobject = new Erived();
    f(nobject);
```

If we have a pointer to a derived object, stored in a pointer to a base class object, it's possible to turn it safely into a derived pointer again:

Code:

```
void f( Base *obj ) {
  Derived *der =
    dynamic_cast<Derived*>(obj);
  if (der==nullptr)
    cout << "Could not be cast to Derived"
         << endl;
  else
    der->print();
};
  /* ... */
    Base *object = new Derived();
    f(object);
    Base *nobject = new Erived();
    f(nobject);
```

Output
[cast] deriveright:

```
Construct derived!
Could not be cast to Derived
```

On the other hand, a `static_cast` would not do the job:

Code:

```
void g( Base *obj ) {
  Derived *der =
    static_cast<Derived*>(obj);
  der->print();
};
  /* ... */
    Base *object = new Derived();
    g(object);
    Base *nobject = new Erived();
    g(nobject);
```

Output
[cast] derivewrong:

```
Construct derived!
Construct erived!
```

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.

24.4.3 Const cast

With `const_cast` you can add or remove const from a variable. This is the only cast that can do this.

24.4.4 Reinterpret cast

The `reinterpret_cast` is the crudest cast, and it corresponds to the C mechanism of 'take this byte and pretend it of type whatever'. There is a legitimate use for this:

```
void *ptr;
ptr = malloc( how_much );
auto address = reinterpret_cast<long int>(ptr);
```

so that you can do arithmetic on the address. For this particular case, `intptr_t` is actually better.

24.4.5 A word about void pointers

A traditional use for casts in C was the treatment of *void pointer*s. The need for this is not as severe in C++ as it was before.

A typical use of void pointers appears in the PETSc [3, 4] library. Normally when you call a library routine, you have no further access to what happens inside that routine. However, PETSc has the functionality for you to specify a monitor so that you can print out internal quantities.

```
int KSPSetMonitor(KSP ksp,
  int (*monitor)(KSP,int,PetscReal,void*),
  void *context,
  // one parameter omitted
  );
```

Here you can declare your own monitor routine that will be called internally: the library makes a *callback* to your code. Since the library can not predict whether your monitor routine may need further information in order to function, there is the `context` argument, where you can pass a structure as void pointer.

This mechanism is no longer needed in C++ where you would use a *lambda* (section 24.3):

```
KSPSetMonitor( ksp,
  [mycontext] (KSP k,int ,PetscReal r) -> int {
    my_monitor_function(k,r,mycontext); } );
```

24.5 lvalue vs rvalue

The terms 'lvalue' and 'rvalue' sometimes appear in compiler error messages.

```
int foo() {return 2;}

int main()
{
    foo() = 2;

    return 0;
```

```
}

# gives:
test.c: In function 'main':
test.c:8:5: error: lvalue required as left operand of assignment
```

See the 'lvalue' and 'left operand'? To first order of approximation you're forgiven for thinking that an *lvalue* is something on the left side of an assignment. The name actually means 'locator value': something that's associated with a specific location in memory. Thus an lvalue is, also loosely, something that can be modified.

An *rvalue* is then something that appears on the right side of an assignment, but is really defined as everything that's not an lvalue. Typically, rvalues can not be modified.

The assignment `x=1` is legal because a variable `x` is at some specific location in memory, so it can be assigned to. On the other hand, `x+1=1` is not legal, since `x+1` is at best a temporary, therefore not at a specific memory location, and thus not an lvalue.

Less trivial examples:

```
int foo() { x = 1; return x; }
int main() {
  foo() = 2;
}
```

is not legal because `foo` does not return an lvalue. However,

```
class foo {
private:
  int x;
public:
  int &xfoo() { return x; };
};
int main() {
  foo x;
  x.xfoo() = 2;
```

is legal because the function `xfoo` returns a reference to the non-temporary variable `x` of the `foo` object.

Not every lvalue can be assigned to: in

```
const int a = 2;
```

the variable `a` is an lvalue, but can not appear on the left hand side of an assignment.

24.5.1 Conversion

Most lvalues can quickly be converted to rvalues:

```
int a = 1;
int b = a+1;
```

Here `a` first functions as lvalue, but becomes an rvalue in the second line.

The ampersand operator takes an lvalue and gives an rvalue:

```
int i;
int *a = &i;
&i = 5; // wrong
```

24.5.2 References

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```
std::string &s = std::string(); // wrong
```

is illegal. The type of `s` is an 'lvalue reference' and it can not be assigned from an rvalue.

On the other hand

```
const std::string &s = std::string();
```

works, since `s` can not be modified any further.

24.5.3 Rvalue references

A new feature of C++ is intended to minimize the amount of data copying through *move semantics*.

Consider a copy assignment operator

```
BigThing& operator=( const BigThing &other ) {
  BigThing tmp(other); // standard copy
  std::swap( /* tmp data into my data */ );
  return *this;
};
```

This calls a copy constructor and a destructor on `tmp`. (The use of a temporary makes this safe under exceptions. The `swap` method never throws an exception, so there is no danger of half-copied memory.)

However, if you assign

```
thing = BigThing(stuff);
```

Now a constructor and destructor is called for the temporary rvalue object on the right-hand side.

Using a syntax that is new in *C++*, we create an *rvalue reference*:

```
BigThing& operator=( BigThing &&other ) {
  swap( /* other into me */ );
  return *this;
}
```

24.6 Move semantics

With an *overloaded operator*, such as addition, on matrices (or any other big object):

```
Matrix operator+(Matrix &a,Matrix &b);
```

the actual addition will involve a copy:

```
Matrix c = a+b;
```

Use a move constructor:

```
class Matrix {
private:
  Representation rep;
public:
  Matrix(Matrix &&a) {
    rep = a.rep;
    a.rep = {};
  }
};
```

24.7 Graphics

C++ has no built-in graphics facilities, so you have to use external libraries such as *OpenFrameworks*, https://openframeworks.cc.

24.8 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.

24.8.1 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 15.

24.8.2 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 11.2. The C++11 mechanism, using an *iterator* (section 23.2.2) is largely obviated.
- Lambdas. See section 24.3.

- Variadic templates.
- Unique pointer.

```
unique_ptr<int> iptr( new int(5) );
```

  This fixes problems with *auto_ptr*.
- *constexpr*

```
constexpr int get_value() {
  return 5*3;
}
```

24.8.3 C++14

C++14 can be considered a bug fix on C++11. It simplifies a number of things and makes them more elegant.

- Auto return type deduction:

```
  auto f() {
    SomeType something;
    return something;
  }
```

- Generic lambdas (section 24.3.2)

```
const auto count = std::count(begin(vec),end(vec),
    [] ( const auto i ) { return i<3; }
);
```

  Also more sophisticated capture expressions.
- Unique pointer

```
auto iptr( make_unique<int>(5) );
```

  This makes **new** and **delete** completely superfluous.
- *constexpr*

```
constexpr int get_value() {
  int val = 5;
  int val2 = 3;
  return val*val2
}
```

24.8.4 C++17

- Optional; section 23.5.2.
- Structured binding declarations as an easier way of dissecting tuples; section 23.4.
- Init statement in conditionals; section **??**.

24.8.5 C++20

- *modules*: these offer a better interface specification than using *header files*.
- *coroutines*, another form of parallelism.
- *concepts* including in the standard library via ranges; section 21.3.
- *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)
- *ranges*
- *calendars* and *time zones*
- *text formatting*
- `span`. See section 11.8.5.

Chapter 25

Graphics

The C++ language and standard library do not have graphics components. However, the following projects exist.

https://www.sfml-dev.org/

Chapter 26

C++ for C programmers

26.1 I/O

There is little employ for `printf` and `scanf`. Use **`cout`** (and `cerr`) and **`cin`** instead. There is also the *fmtlib* library.

Chapter 13.

26.2 Arrays

Arrays through square bracket notation are unsafe. They are basically a pointer, which means they carry no information beyond the memory location.

It is much better to use `vector`. Use range-based loops, even if you use bracket notation.

Chapter 11.

Vectors own their data exclusively, so having multiple C style pointers into the same data act like so many arrays does not work. For that, use the **`span`**; section 11.8.5.

26.2.1 Vectors from C arrays

Suppose you have to interface to a C code that uses `malloc`. Vectors have advantages, such as that they know their size, you may want to wrap these C style arrays in a vector object. This is easily done using a *range constructor*:

```
vector<double> x( pointer_to_first, pointer_after_last );
```

Such vectors can still be used dynamically, but this may give a memory leak and other possibly unwanted behavior:

Code:

```
    float *x;
    x = (float*)malloc(length*sizeof(float));
/* ... */
    vector<float> xvector(x,x+length);
    cout << "xvector has size: " << xvector.size() << 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
[array] cvector:

```
xvector has size: 53
Push back was successful
pushed element: 5
original array: 1.02032e+29
```

26.3 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 11.8.3. (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

```
double *x;
if (something1) x = malloc(10);
if (something2) x[0];
```

which may give a memory error. Instead, declaration of the name and allocation of the storage are one indivisible action.

On the other hand:

```
vector<double> x(10);
```

declares the variable `x`, allocates the dynamic storage, and initializes it.

26.4 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 12.

26.5 Pointers

Many of the uses for *C pointer*s, which are really addresses, have gone away.

- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see above.
- Traversing arrays and vectors can be done with ranges; section 11.2.

- 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 15.2. The C pointers are now called *bare pointer*s, and they can still be used for 'non-owning' occurrences of pointers.

26.5.1 Parameter passing

No longer by address: now true references! Section 7.4.

26.6 Objects

Objects are structures with functions attached to them. Chapter 10.

26.7 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter 19.

26.8 Templates

If you find yourself writing the same function for a number of types, you'll love templates. Chapter 21.

26.9 Obscure stuff

26.9.1 Lambda

Function expressions. Section 24.3.

26.9.2 Const

Functions and arguments can be declared const. This helps the compiler. Section 17.1.

26.9.3 Lvalue and rvalue

Section 24.5.

Chapter 27

C++ review questions

27.1 Arithmetic

1. Given
   ```
   int n;
   ```
   write code that uses elementary mathematical operators to compute n-cubed: n^3.
   Do you get the correct result for all n? Explain.
2. What is the output of:
   ```
   int m=32, n=17;
   cout << n%m << endl;
   ```

27.2 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.

27.3 Functions

Exercise 27.1. The following code snippet computes in a loop the recurrence

$$v_{i+1} = av_i + b, \qquad v_0 \text{ given.}$$

Write a recursive function

```
float v = value_n(n,a,b,v0);
```

that computes the value v_n for $n \geq 0$.

27.4 Vectors

Exercise 27.2. The following program has several syntax and logical errors. The intended purpose is to read an integer N, and sort the integers $1, \ldots, N$ into two vectors, one for the odds and one for the evens. The odds should then be multiplied by two.

Your assignment is to debug this program. For 10 points of credit, find 10 errors and correct them. Extra errors found will count as bonus points. For logic errors, that is, places that are syntactically correct, but still 'do the wrong thing', indicate in a few words the problem with the program logic.

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

int main() {
  vector<int> evens,odd;
  cout << "Enter an integer value " << endl;
  cin << N;
  for (i=0; i<N; i++) {
    if (i%2=0) {
      odds.push_back(i);
    else
      evens.push_back(i);
  }
  for ( auto o : odds )
    o /= 2
  return 1
}
```

27.5 Vectors

Exercise 27.3. Take another look at exercise 27.1. 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?

27.6 Objects

Exercise 27.4. Let a class `Point` class be given. How would you design a class `SetOfPoints` (which models a set of points) so that you could write

```
Point p1,p2,p3;
SetOfPoints pointset;
// add points to the set:
pointset.add(p1); pointset.add(p2);
```

Give the relevant data members and methods of the class.

Exercise 27.5. You are programming a video game. There are moving elements, and you want to have an object for each. Moving elements need to have a method `move` with an argument that indicates a time duration, and this method updates the position of the element, using the speed of that object and the duration.

Supply the missing bits of code.

```
class position {
  /* ... */
public:
  position() {};
  position(int initial) { /* ... */ };
  void move(int distance) { /* ... */ };
};
class actor {
protected:
  int speed;
  position current;

public:
  actor() { current = position(0); };
  void move(int duration) {
    /* THIS IS THE EXERCISE: */
    /* write the body of this function */
  };
};
class human : public actor {
public:
  human() // EXERCISE: write the constructor
};
class airplane : public actor {
public:
  airplane() // EXERCISE: write the constructor
};

int main() {
  human Alice;
```

```
        airplane Seven47;
        Alice.move( 5 );
        Seven47.move( 5 );
```

PART III

FORTRAN

Chapter 28

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`

28.1 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 `f90` are the extensions for old-style fixed format; and
- `F` and `F90` are the extensions for new free format.

The postfix `90` indicates that the *C preprocessor* is applied to the file. For this course we will use the `F90` extension.

28.2 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 28.1. Add the line

```
print *,"Hello world!"
```

to the empty program, and compile and run it.

Fortran ignores *case*. Both keywords such as `Begin` or **`Program`** can just as well be written as `bEGIN` or `PrOgRaM`.

A program optionally has a **`stop`** statement, which can return a message to the OS.

Code:

```
Program SomeProgram
  stop 'the code stops here'
End Program SomeProgram
```

Output
[basicf] stop:

```
STOP the code stops here
```

28.3 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.

28.3.1 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
```

(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.)

- No includes before the program
- Program has a name (emacs tip: type `end<TAB>`)
- There is an **End**, rather than curly braces.
- Declarations first, not interspersed.

28.3.2 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
```

- If a statement spans more than one line, all but the first line need to have an explicit *continuation character*, the ampersand:

```
x = very &
  long &
  expression
```

28.3.3 Comments

Fortran knows only single-line *comments*, indicated by an exclamation point:

```
x = 1 ! set x to one
```

Everything from the exclamation point onwards is ignored.

Maybe not entirely obvious: you can have a comment after a continuation character:

```
x = f(a) & ! term1
  + g(b)   ! term2
```

28.4 Variables

Unlike in C++, where you can declare a variable right before you need it, Fortran wants its variables declared near the top of the program or subprogram:

```
Program YourProgram
  implicit none
  ! variable declaration
  ! executable code
End Program YourProgram
```

A variable declaration looks like:

```
type [ , attributes ] :: name1 [ , name2, .... ]
```

where

- we use the common grammar shorthand that `[ something ]` stands for an optional 'something';
- *type* is most commonly **integer**, **real**(4), **real**(8), **logical**. See below; section 28.4.1.
- *attributes* can be **dimension**, **allocatable**, **intent**, *parameters* et cetera.
- *name* is something you come up with. This has to start with a letter.
- Numeric: **Integer**, **Real**, **Complex**.
- Logical: **Logical**.
- Character: **Character**. Strings are realized as arrays of characters.

Sometimes an identifier corresponds to a constant:
use the **parameter** attribute

```
real,parameter :: pi = 3.141592
```

This can not be changed like an ordinary variable.

In chapter 37 you will see that **parameter**s are often used for defining the size of an array.

Further specifications for numerical precision are discussed in section 28.4.2. Strings are discussed in chapter 33.

28.4.1 Declarations

Fortran has a somewhat unusual treatment of data types: if you don't specify what data type a variable is, Fortran will deduce it from a default or user rules. This is a very dangerous practice, so we advocate putting a line

```
implicit none
```

immediately after any program or subprogram header.

You can the number of bytes for numerical types in the declaration:

```
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
complex*32 :: c32
```

28.4.2 Precision

In Fortran you can actually 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.

- To query the type of a variable, use the function **kind**, which returns an integer.

Likewise, you can specify the precision of a constant. Writing `3.14` will usually be a single precision real.

Code:

```
real(8) :: x,y,z
x = 1.
y = .1
z = x+y
print *,z
x = 1.d0
y = .1d0
z = x+y
print *,z
```

Output
[basicf] e0:

```
1.1000000014901161
1.1000000000000001
```

You can query how many bytes a data type takes with **kind**.

Number of bytes determines numerical precision:

- Computations in 4-byte have relative error $\approx 10^{-6}$
- Computations in 8-byte have relative error $\approx 10^{-15}$

Also different exponent range: max $10^{\pm 50}$ and $10^{\pm 300}$ respectively.

F08: *storage_size* reports number of bits.
F95: **bit_size** works on integers only.
c_sizeof reports number of bytes, requires *iso_c_binding* module.

Force a constant to be **real**(8):

```
real(8) :: x,y
x = 3.14d0
y = 6.022e-23
```

- Use a compiler flag such as `-r8` to force all reals to be 8-byte.
- Write `3.14d0`
- `x = real(3.14, kind=8)`

Complex constants are written as a pair of reals in parentheses.
There are some basic operations.

Code:

```
Complex :: &
     fourtyfivedegrees = (1.,1.), &
     other
print *,fourtyfivedegrees
other = 2*fourtyfivedegrees
print *,other
```

Output
[basicf] complex:

```
(1.00000000,1.00000000)
(2.00000000,2.00000000)
```

28.4.3 Initialization

Variables can be initialized in their declaration:

```
integer :: i=2
real(4) :: x = 1.5
```

That this is done at compile time, leading to a common error:

```
subroutine foo()
  implicit none
  integer :: i=2
  print *,i
  i = 3
end subroutine foo
```

On the first subroutine call `i` is printed with its initialized value, but on the second call this initialization is not repeated, and the previous value of `3` is remembered.

28.5 Input/Output, or I/O as we say

- Input:

```
READ *,n
```

- Output:

```
PRINT *,n
```

There is also **`Write`**.
The 'star' indicates that default formatting is used.
Other syntax for read/write with files and formats.

28.6 Expressions

- Pretty much as in C++
- Exception: `r**a` for power r^a.
- Modulus is a function: **`MOD`**`(7,3)`.

- Long form:
  `.and. .not. .or.`
  `.lt. .le. .eq. .ne. .ge. .gt.`
  `.true. .false.`
- Short form:
  `< <= == /= > >=`

Conversion is done through functions.

- **`INT`**: truncation; **`NINT`** rounding
- **`REAL`**, *`FLOAT`*, *`SNGL`*, **`DBLE`**
- **`CMPLX`**, **`CONJG`**, *`AIMIG`*

http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-conv.htm

Complex numbers exist

Strings are delimited by single or double quotes.
For more, see chapter 33.

28.7 Review questions

Exercise 28.2. What is the output for this fragment, assuming `i, j` are integers?

```
integer :: idiv
!! ...
i = 3 ; j = 2 ; idiv = i/j
print *,idiv
```

Exercise 28.3. What is the output for this fragment, assuming `i, j` are integers?

```
real    :: fdiv
!! ...
i = 3 ; j = 2 ; fdiv = i/j
print *,fdiv
```

Exercise 28.4. In declarations

```
real(4) :: x
real(8) :: y
```

what do the `4` and `8` stand for?

What is the practical implication of using the one or the other?

Exercise 28.5. 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 28.6. In the following code, if `value` is nonzero, what do expect about the output?

```
real(8) :: value8,should_be_value
real(4) :: value4
!! ...
print *,".. original value was:",value8
value4 = value8
print *,".. copied to single:",value4
should_be_value = value4
print *,".. copied back to double:",should_be_value
print *,"Difference:",value8-should_be_value
```

Chapter 29

Conditionals

29.1 Forms of the conditional statement

The Fortran conditional statement uses the `if` keyword:

Single line conditional:

```
if ( test ) statement
```

The full if-statement is:

```
if ( something ) then
  !! something_doing
else
  !! otherwise_else
end if
```

The 'else' part is optional; you can nest conditionals.

You can label conditionals, which is good for readability but adds no functionality:

```
checkx: if ( ... some test on x ... ) then
checky:   if ( ... some test on y ... ) then
                ... code ...
            end if checky
          else checkx
              ... code ...
          end if checkx
```

29.2 Operators

Operator	old style	meaning	example
==	.eq.	equals	x==y-1
/=	.ne.	not equals	x*x*!=5
>	.gt.	greater	y>x-1
>=	.ge.	greater or equal	sqrt(y)>=7
<	.lt.	less than	
<=	.le.	less equal	
	.and. .or.	and, or	x<1 .and. x>0
	.not.	not	.not.(x>1 .and. x<2)
	.eqv.	equiv	$(x \wedge y) \vee (\neg x \wedge \neg y)$
	.neqv.	not equiv	$(x \wedge \neg y) \vee (\neg x \wedge y)$

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

Exercise 29.1. Read in three grades: Algebra, Biology, Chemistry, each on a scale $1 \cdots 10$. Compute the average grade, with the conditions:

- Algebra is always included.
- Biology is only included if it increases the average.
- Chemistry is only included if it is 6 or more.

29.3 Select statement

The Fortran equivalent of the C++ `case` statement is `select`. It takes single values or ranges; works for integers and characters.

Test single values or ranges, integers or characters:

```
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.

29.4 Boolean variables

The Fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`.

Exercise 29.2. Print a boolean variable. What does the output look like in the true and false case?

29.5 Review questions

Exercise 29.3. What is a conceptual difference between the C++ `switch` and the Fortran `Select` statement?

Chapter 30

Loop constructs

30.1 Loop types

Fortran has the usual indexed and 'while' loops. There are variants of the basic loop, and both use the `do` keyword. The simplest loop has

- a loop variable, which needs to be declared;
- a lower bound and upper bound.

We'll see more types of loops below.

```
integer :: i

do i=1,10
  ! code with i
end do
```

You can include a step size (which can be negative) as a third parameter:

```
do i=1,10,3
  ! code with i
end do
```

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.

The while loop has a pre-test:

```
do while (i<1000)
  print *,i
  i = i*2
end do
```

You can label loops, which improves readability, but see also below.

```
outer: do i=1,10
    inner: do j=1,10
    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 11 *There is a legacy mechanism of label-terminated loops. Please do not use them.*

30.2 Interruptions of the control flow

For indeterminate looping, you can use the **while** test, or leave out the loop parameter altogether. In that case you need the **exit** statement to stop the iteration.

Loop without counter or while test:

```
do
  call random_number(x)
  if (x>.9) exit
  print *,"Nine out of ten exes agree"
end do
```

Skip rest of current iteration:

```
do i=1,100
  if (isprime(i)) cycle
  ! do something with non-prime
end do
```

Cycle and exit can apply to multiple levels, if the do-statements are labeled.

```
outer : do i = 1,10
inner : do j = 1,10
    if (i+j>15) exit outer
    if (i==j) cycle inner
end do inner
end do outer
```

30.3 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:

```
print *,(2*i,i=1,20)
```

You can iterate multiple expressions:

```
print *,(2*i,2*i+1,i=1,20)
```

These loops can be nested:

```
print *,( (i*j,i=1,20), j=1,20 )
```

This construct is especially useful for printing arrays.

Exercise 30.1. Use the implied do-loop mechanism to print a triangle:

```
1
2 2
3 3 3
4 4 4 4
```

up to a number that is input.

30.4 Review questions

Exercise 30.2. What is the output of:

```
do i=1,11,3
  print *,i
end do
```

What is the output of:

```
do i=1,3,11
  print *,i
end do
```

Chapter 31

Scope

31.1 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.

31.1.1 Variables local to a program unit

Variables declared in a subprogram have similar scope rules as in C++:

- Their visibility is controlled by their textual scope:

```
Subroutine Foo()
  integer :: i
  ! 'i' can now be used
  call Bar()
  ! 'i' still exists
End Subroutine Foo
Subroutine Bar() ! no parameters
  ! The 'i' of Foo is unknown here
End Subroutine Bar
```

- Their dynamic scope is the lifetime of the program unit in which they are declared:

```
Subroutine Foo()
  call Bar()
  call Bar()
End Subroutine Foo
Subroutine Bar()
  Integer :: i
  ! 'i' is created every time Bar is called
End Subroutine Bar
```

31.1.1.1 Variables in a module

Variables in a module (section 35.2) have a lifetime that is independent of the calling hierarchy of program units: they are *static variables*.

31.1.1.2 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.

31.1.2 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 32

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*, but this is not a language keyword.

If you structure your code in a single file, this is the recommended structure:

```
Program foo
  < declarations>
  < executable statements >
  Contains
    < procedure definitions >
End Program foo
```

That is, procedure 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 35.2.
- 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 40.1.

32.1 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:

```
subroutine foo( <parameters> )
<variable declarations>
<executable statements>
end subroutine foo
```

and

```
returntype function 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.

```
subroutine foo()
  print *,"foo"
  if (something) return
  print *,"bar"
end subroutine foo
```

The `return` statement is optional in the first case. The `return` statement is different from C++ in that it does not indicate the return result of a function.

Exercise 32.1. Rewrite the above subroutine `foo` without a `return` statement.

A subroutine is invoked with a `call` statement:

```
call foo()
```

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
```

Code:

```
program adding
  implicit none
  integer :: i=5
  call addint(i,4)
  print *,i
contains
  subroutine addint(inoutvar,addendum)
    implicit none
    integer :: inoutvar,addendum
    inoutvar = inoutvar + addendum
  end subroutine addint
end program adding
```

Output
[funcf] addone:

```
9
```

Parameters are always ‘by reference’!

Declare function as `Recursive Function`

Code:

```
recursive integer function fact(invalue) &
     result (val)
  implicit none
  integer,intent(in) :: invalue
  if (invalue==0) then
     val = 1
  else
     val = invalue * fact(invalue-1)
  end if
end function fact
```

Output
[funcf] fact:

```
echo 7 | ./fact
           7 factorial is        5040
```

Note the **result** clause. This prevents ambiguity.

32.2 Return results

While a **subroutine** can only return information through its parameters, a *function* procedure returns an explicit result:

```
logical function test(x)
  implicit none
  real :: x

  test = some_test_on(x)
  return ! optional, see above
end function test
```

You see that the result is not returned in the **return** statement, but rather through assignment to the function name. The **return** statement, as before, is optional and only indicates where the flow of control ends.

A *function* in Fortran is a procedure that return a result to its calling program, much like a non-void function in C++

- **subroutine** vs **function**:
  compare *void* functions vs non-void in C++.
- Return type, keyword **function**, name, parameters
- Function body has statements
- Result is returned by assigning to the function name
- Use: `y = f(x)`

Code:

```
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
```

Output
[funcf] plusone:

```
6
```

A function is not invoked with `call`, but rather through being used in an expression:

```
if (test(3.0) .and. something_else) ...
```

You now have the following cases to make the function known in the main program:

- If the function is in a `contains` section, its type is known in the main program.
- If the function is in a module (see section 35.2 below), it becomes known through a `use` statement.

  F77 note: Without modules and `contains` sections, you need to declare the function type explitly in the calling program. The safe way is through using an `interface` specification.

Exercise 32.2. 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:

Code:

```
program readpos
  implicit none
  real(4) :: userinput
  print *,"Type a positive number:"
  userinput = read_positive()
  print *,"Thank you for",userinput
contains
  real(4) function read_positive()
    implicit none
  !! ...
  end function read_positive
end program readpos
```

Output
[funcf] readpos:

```
Type a positive number:
No, not  -5.00000000
No, not   0.00000000
No, not  -3.14000010
Thank you for   2.48000002
```

32.2.1 The 'result' keyword

Apart from assigning to the function name, there is a second mechanism for returning a function result, namely through the `Result` keyword.

```
function some_function() result(x)
  implicit none
  real :: x
```

```
  !! stuff
  x = ! some computation
end function
```

You see that here

- the assignment to the name is missing,
- the function name is not typed; but
- instead there is a typed local variable that is marked to be the result.

32.2.2 The 'contains' clause

```
Program NoContains
  implicit none
  call DoWhat()
end Program NoContains

subroutine DoWhat(i)
  implicit none
  integer :: i
  i = 5
end subroutine DoWhat
```

Warning only, crashes.

```
Program ContainsScope
  implicit none
  call DoWhat()
contains
  subroutine DoWhat(i)
    implicit none
    integer :: i
    i = 5
  end subroutine DoWhat
end Program ContainsScope
```

Error, does not compile

Code:

```
Program NoContainTwo
  implicit none
  integer :: i=5
  call DoWhat(i)
end Program NoContainTwo

subroutine DoWhat(x)
  implicit none
  real :: x
  print *,x
end subroutine DoWhat
```

Output
[funcf] nocontaintype:

```
7.00649232E-45
```

At best compiler warning if all in the same file
For future reference: if you see very small floating point numbers, maybe you have made this error.

32.3 Arguments

Arguments are declared in procedure body:

```
subroutine f(x,y,i)
  implicit none
  integer,intent(in) :: i
  real(4),intent(out) :: x
  real(8),intent(inout) :: y
  x = 5; y = y+6
```

```
end subroutine f
! and in the main program
call f(x,y,5)
```

declaring the 'intent' is optional, but highly advisable.

- Everything is passed by reference.
  Don't worry about large objects being copied.
- Optional intent declarations:
  Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.

The term *dummy argument* is what Fortran calls the parameters in the procedure definition. The arguments in the procedure call are the *actual argument*s.

Compiler checks your intent against your implementation. This code is not legal:

```
subroutine ArgIn(x)
  implicit none
  real,intent(in) :: x
  x = 5 ! compiler complains
end subroutine ArgIn
```

Self-protection: if you state the intended behaviour of a routine, the compiler can detect programming mistakes.
Allow compiler optimizations:

```
x = f()
call ArgOut(x)
print *,x
```

Call to `f` removed

```
do i=1,1000
  x = ! something
  y1 = .... x ....
  call ArgIn(x)
  y2 = ! same expression as y1
```

`y2` is same as `y1` because `x` not changed

(May need further specifications, so this is not the prime justification.)

Exercise 32.3. Write a subroutine `trig` that takes a number α as input and passes $\sin\alpha$ and $\cos\alpha$ back to the calling environment.

32.4 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 `entry` statement is so bizarre that I refuse to discuss it.

32.5 More about arguments

- Use the name of the *formal parameter* as keyword.
- Keyword arguments have to come last.

Code:

```
  call say_xy(1,2)
  call say_xy(x=1,y=2)
  call say_xy(y=2,x=1)
  call say_xy(1,y=2)
  ! call say_xy(y=2,1) ! ILLEGAL
contains
  subroutine say_xy(x,y)
    implicit none
    integer,intent(in) :: x,y
    print *,"x=",x,", y=",y
  end subroutine say_xy
```

Output
[funcf] keyword:

```
 x=           1 , y=           2
 x=           1 , y=           2
 x=           1 , y=           2
 x=           1 , y=           2
```

- Extra specifier: **Optional**
- Presence of argument can be tested with **Present**

Fortran behaves slightly counterintuitive if you try to initialize local variables: they initialization is only executed once, and the second time you call the procedure it will have retained its previous value. (Technically: the variable has an implicit *Save* attribute.)

This may trip you up as the following example shows:

Local variable is initialized only once,
second time it uses its retained value.

Code:

```
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
```

Output
[funcf] save:

```
Comparing:  1   3
            3
Comparing: -2  -4
            3
```

Chapter 33

String handling

33.1 String denotations

A string can be enclosed in single or double quotes. That makes it easier to have the other type in the string.

```
print *,'This string was in single quotes'
print *,'This string in single quotes contains a single '' quote'
print *,"This string was in double quotes"
print *,"This string in double quotes contains a double "" quote"
```

33.2 Characters

33.3 Strings

The length of a Fortran string is specified with the `len` keyword when the string is created:

```
character(len=50) :: mystring
mystring = "short string"
```

The `len` function also gives the length of the string, but note that that is the length with which it was allocated, not how much non-blank content you put in it.

Code:

```
character(len=12) :: strvar
!! ...
strvar = "word"
print *,len(strvar),len(trim(strvar))
```

Output

[stringf] strlen:

```
          12           4
```

To get the more intuitive length of a string, that is, the location of the last non-blank character, you need to `trim` the string.

Intrinsic functions: `LEN(string)`, `INDEX(substring,string)`, `CHAR(int)`, `ICHAR(char)`, `TRIM(string)`

Code:

```
character(len=10) :: firstname,lastname
character(len=15) :: shortname,fullname
!! ...
firstname = "Victor"; lastname = "Eijkhout"
shortname = firstname // lastname
print *,"without trimming: ",shortname
fullname = trim(firstname) // " " // trim(lastname)
print *,"with trimming: ",fullname
```

Output
[stringf] concat:

```
without trimming: Victor    Eijkh
with trimming: Victor Eijkhout
```

33.4 Strings versus character arrays

33.5 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.

33.5.1 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
```

33.5.2 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:

- '`a`n' specifies a string of n characters. If the actual string is longer, it is truncated in the output.
- '`i`n' 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`.
- '`f`$m.n$' specifies a fixed point representation of a real number, with m total positions (including the decimal point) and n digits in the fractional part.
- '`e`$m.n$' 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

```
print '( 3i4 )', i1,i2,i3
print '( 3(i2,":",f7.4) )', i1,r1,i2,r2,i3,r2
```

- If `abc` is a format string, then `10(abc)` gives 10 repetitions. There is no line break.
- If there is more data than specified in the format, the format is reused in a new print statement. This causes line breaks.
- The `/` (slash) specifier causes a line break.

- There may be a 80 character limit on output lines.

Exercise 33.1. Use formatted I/O to print the number $0 \cdots 99$ as follows:

```
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

Chapter 34

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 name` / `End Type` block.
Component declarations inside the block:

```
type mytype
  integer :: number
  character :: name
  real(4) :: value
end type mytype
```

Declare a type object in the main program:

```
Type(mytype) :: object1,object2
```

Initialize with type name:

```
object1 = mytype( 1, 'my_name', 3.7 )
```

Copying:

```
object2 = object1
```

Access structure members with `%`

```
Type(mytype) :: typed_object
typed_object%member = ....
```

```
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
(or use a **module**; see later)

You can have arrays of types:

```
type(my_struct) :: data
type(my_struct),dimension(1) :: data_array
```

Structures can be passed as subprogram argument, just like any other datatype. This example:

- 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 )
end function length
```

Function call

```
print *,"Length:",length(p2)
```

Exercise 34.1. Write a program that has the following:

- A type `Point` that contains real numbers `x,y`;
- a type `Rectangle` that contains two `Points`, corresponding to the lower left and upper right point;
- a function `area` that has one argument: a `Rectangle`.

Test your program.

Chapter 35

Modules

Fortran has a clean mechanism for importing data (including numeric constants such as π), functions, types that are defined in another file.

Modules look like a program, but without executable code:

```
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 imported through `use` statement;
comes before `implicit none`

Code:

```
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] typemod:

```
   2.50000000       3.70000005
 length:   4.46542263
   6.2831854820251465
```

Exercise 35.1. Take exercise 34.1 and put all type definitions and all functions in a module.

35.1 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.

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.

35.2 Modules

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.

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`**

A module is made available with the **`use`** keyword, which needs to go before the **`implicit none`**.

```
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
```

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.

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 35.2. Write a module *PointMod* that defines a type *Point* and a function *distance* to make this code work:

```
use pointmod
implicit none
type(Point) :: p1,p2
real(8) :: p1x,p1y,p2x,p2y
read *,p1x,p1y,p2x,p2y
p1 = point(p1x,p1y)
p2 = point(p2x,p2y)
print *,"Distance:",distance(p1,p2)
```

Put the program and module in two separate files and compile thusly:

```
ifort -g -c pointmod.F90
ifort -g -c pointmain.F90
ifort -g  -o pointmain pointmod.o pointmain.o
```

35.2.1 Polymorphism

```
module somemodule

INTERFACE swap
MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
END INTERFACE

  contains
  subroutine swapreal
  ...
  end subroutine swapreal
  subroutine swapint
  ...
  end subroutine swapint
```

35.2.2 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 36

Classes and objects

36.1 Classes

Fortran classes are based on **type** objects, a little like the analogy between C++ `struct` and `class` constructs.
New syntax for specifying methods.

You define a type as before, with its data members, but now the type has a **contains** for the methods:

```
Module multmod

  type Scalar
     real(4) :: value
   contains
     procedure,public :: &
          printme,scaled
  end type Scalar

contains ! methods
  !! ...
end Module multmod
```

```
Program Multiply
  use multmod
  implicit none

  type(Scalar) :: x
  real(4) :: y
  x = Scalar(-3.14)
  call x%printme()
  y = x%scaled(2.)
  print '(f7.3)',y

end Program Multiply
```

```
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
```

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
```

- 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. ⇐

```
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
```

Exercise 36.1. Take the point example program and add a distance function:

```
Type(Point) :: p1,p2
! ... initialize p1,p2
dist = p1%distance(p2)
! ... print distance
```

Exercise 36.2. 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`?

36.1.1 Final proceduares: destructors

The Fortran equivalent of *destructors* is a *final procedure*, designated by the *final* keyword.

```
 contains
   final :: &
        print_final
end type Scalar
```

A final procedure has a single argument of the type that it applies to:

```
subroutine print_final(me)
  implicit none
  type(Scalar) :: me
  print '("On exit: value is",f7.3)',me%value
end subroutine print_final
```

The final procedure is invoked when a derived type object is deleted, except at the conclusion of a program:

```
  call tmp_scalar()
contains
  subroutine tmp_scalar()
    type(Scalar) :: x
    real(4) :: y
    x = Scalar(-3.14)
  end subroutine tmp_scalar
```

36.1.2 Inheritance

Inheritance:

```
type,extends(baseclas) :: derived_class
```

Pure virtual:

```
type,abstract
```

http://fortranwiki.org/fortran/show/Object-oriented+programming

Chapter 37

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.

37.1 Static arrays

The preferred way for specifying an array size is:

Preferred way of creating arrays through `dimension` keyword:

```
real(8), dimension(100) :: x,y
```

One-dimensional arrays of size 100.
Older mechanism works too:

```
integer :: i(10,20)
```

Two-dimensional array of size 10×20.
These arrays are statically defined, and only live inside their program unit.

Such an array, with size explicitly indicated, is called a *static array* or *automatic array*. (See section 37.4 for dynamic arrays.)

Array indexing in Fortran is 1-based by default:

```
integer,parameter :: N=8
real(4),dimension(N) :: x
do i=1,N
  ... x(i) ...
```

Note the use of `parameter`: *compile-time constant*

Unlike C++, Fortran can specify the lower bound explicitly:

```
real,dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Safer:

Code:

```
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.

37.1.1 Initialization

There are various syntaxes for *array initialization*, including the use of *implicit do-loops*:

```
real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
!! ...
real5 = [ (1.01*i,i=1,size(real5,1)) ]
!! ...
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

37.1.2 Array sections

Fortran is more sophisticated than C++ in how it can handle arrays as a whole. For starters, you can assign one array to another:

```
real*8, dimension(10) :: x,y
x = y
```

This obviously requires the arrays to have the same size. You can assign subarrays, or *array sections*, as long as they have the same shape. This uses a colon syntax.

- `:` 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`.

Code:

```
real(8),dimension(5) :: x = &
     [.1d0, .2d0, .3d0, .4d0, .5d0]
!! ...
x(2:5) = x(1:4)
print '(f5.3)',x
```

Output
[arrayf] sectionassign:

```
0.100
0.100
0.200
0.300
0.400
```

Exercise 37.1. 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 an copied to some temporary locations before being written to the left-hand side. In practice, this may require large temporary arrays (and negatively affect performance by lessening *locality*) so you hope that the compiler does something smarter. However, the exercise showed that an array assignment can not trivially be converted to a simply loop.

Exercise 37.2. Can you formalize the sort of array statement for which a simple translation to a loop changes the semantics? (In compiler terminology this is called a *dependence*.)

Array operations can be more sophistaced than assigning to a whole array or a section of it. For instance, you can use a stride:

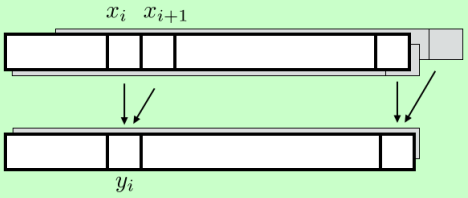
Code:

```
integer,dimension(5) :: &
      y = [0,0,0,0,0]
integer,dimension(3) :: &
      z = [3,3,3]
!! ...
y(1:5:2) = z(:)
print '(i3)',y
```

Output
[arrayf] sectionmg:

```
 3
 0
 3
 0
 3
```

Exercise 37.3.



Code $\forall_i$: $y_i = (x_i + x_{i+1})/2$:

- First with a do loop; then
- in a single array assignment statement by using sections.

Initialize the array `x` with values that allow you to check the correctness of your code.

37.1.3 Integer arrays as indices

```
integer,dimension(4) :: i = [2,4,6,8]
real(4),dimension(10) :: x
print *,x(i)
```

37.2 Multi-dimensional

Arrays above had '*rank* one'. The rank is defined as the number of indices you need to address the elements. Calling this the 'dimension' of the array can be confusing, but we will talk about the first and second dimension of the array.

A rank-two array, or matrix, is defined like this:

```
real(8),dimension(20,30) :: array
array(i,j) = 5./2
```

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 37.1.

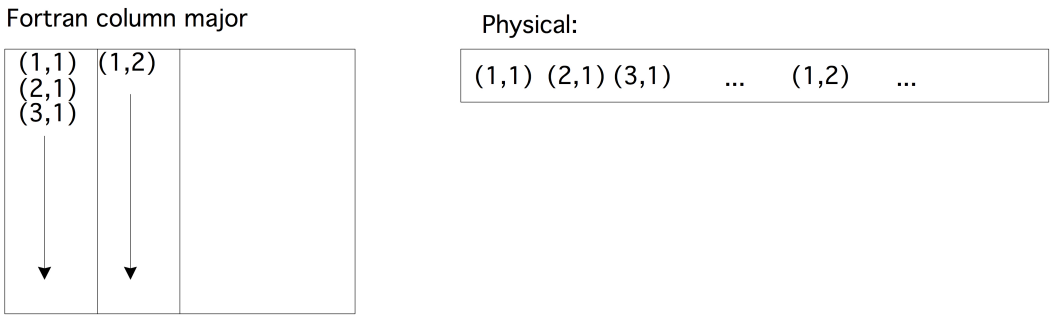


Figure 37.1: Column-major storage in Fortran

To traverse the elements as they are stored in memory, you would need the following code:

```
do col=1,size(A,2)
  do row=1,size(A,1)
    .... A(row,col) .....
  end do
end do
```

This is sometimes described as 'First index varies quickest'.

Exercise 37.4. Can you describe in words how memory elements are access if you would write

```
do row=1,size(A,1)
  do col=1,size(A,2)
    .... A(row,col) .....
  end do
end do
```

?

You can make sections in multi-dimensional arrays: you need to indicate a range in all dimensions.

```
real(8),dimension(10) :: a,b
a(1:9) = b(2:10)
```

or

```
logical,dimension(25,3) :: a
logical,dimension(25)   :: b
a(:,2) = b
```

You can also use strides.

Fill array by rows:

$$\begin{matrix} 1 & 2 & \dots & N \\ N+1 & \dots & & \\ & \dots & & \\ & & & MN \end{matrix}$$

Code:

```
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        11.000000
```

37.2.1 Querying an array

We have the following properties of an array:

- The bounds are the lower and upper bound in each dimension. For instance, after

  ```
  integer,dimension(-1:1,-2:2) :: symm
  ```

  the array `symm` has a lower bound of -1 in the first dimension and -2 in the second. The functions **Lbound** and **Ubound** give these bounds as array or scalar:

  ```
  array_of_lower = Lbound(symm)
  upper_in_dim_2 = Ubound(symm,2)
  ```

Code:

```
real(8),dimension(2:N+1) :: Afrom2 = &
     [1,2,3,4,5]
!! ...
lo = lbound(Afrom1,1)
hi = ubound(Afrom1,1)
print *,lo,hi
print '(i3,":",f5.3)', &
     (i,Afrom1(i),i=lo,hi)
```

Output
[arrayf] fsection2:

```
           1           5
  1:1.000
  2:2.000
  3:3.000
  4:4.000
  5:5.000
```

- The *extent* is the number of elements in a certain dimension, and the *shape* is the array of extents.
- The **size** is the number of elements, either for the whole array, or for a specified dimension.

  ```
  integer :: x(8), y(5,4)
  size(x)
  size(y,2)
  ```

37.2.2 Reshaping

RESHAPE

```
array = RESHAPE( list,shape )
```

Example:

```
square = reshape( (/(i,i=1,16)/),(/4,4/) )
```

SPREAD

```
array = SPREAD( old,dim,copies )
```

37.3 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:

Passing array as one symbol:

Code:

```
real(8),dimension(:) :: x(N) &
     = [ (i,i=1,N) ]
real(8),dimension(:) :: y(0:N-1) &
     = [ (i,i=1,N) ]

sx = arraysum(x)
sy = arraysum(y)
print '("Sum of one-based array:",/,4x,f6.3)', sx
print '("Sum of zero-based array:",/,4x,f6.3)', sy
```

Output
[arrayf] arraypass1d:

```
Sum of one-based array:
    55.000
Sum of zero-based array:
    55.000
```

Note declaration as `dimension(:)`
actual size is queried

```
real(8) function arraysum(x)
  implicit none
  real(8),intent(in),dimension(:) :: x
  real(8) :: tmp
  integer i

  tmp = 0.
  do i=1,size(x)
     tmp = tmp+x(i)
  end do
  arraysum = tmp
end function arraysum
```

The array inside the subroutine is known as a *assumed-shape array* or *automatic array*.

37.4 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.

```
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)
```

37.4.1 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 32.2.1:

```
function create_array(n) result(v)
  implicit none
  integer,intent(in) :: n
  real,dimension(:),allocatable :: v
  integer  :: i
  allocate(v(n))
  v = [ (i+.5,i=1,n) ]
end function create_array
```

37.5 Array output

Use implicit do-loops; section 30.3.

37.6 Operating on an array

37.6.1 Arithmetic operations

Between arrays of the same shape:

```
A = B+C
D = D*E
```

(where the multiplication is by element).

37.6.2 Intrinsic functions

The following intrinsic functions are avaible for arrays:

- **Abs** creates the matrix of pointwise absolute values.
- **MaxVal** finds the maximum value in an array.
- **MinVal** finds the minimum value in an array.
- **Sum** returns the sum of all elements.
- **Product** return the product of all elements.
- **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.

- Functions such as **Sum** operate on a whole array by default.
- To restrict such a function to one subdimension add a keyword parameter **DIM**:

  ```
  s = Sum(A,DIM=1)
  ```

  where the keyword is optional.
- Likewise, the operation can be restricted to a *MASK*:

  ```
  s = Sum(A,MASK=B)
  ```

Exercise 37.5. The 1-norm of a matrix is defined as the maximum of all sums of absolute values in any column:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

while the infinity-norm is defined as the maximum row sum:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Compute these norms using array functions as much as possible, that is, try to avoid using loops.

For bonus points, write Fortran **Function**s that compute these norms.

Exercise 37.6. Compare implementations of the matrix-matrix product.

1. Write the regular `i,j,k` implementation, and store it as reference.
2. Use the **`DOT_PRODUCT`** function, which eliminates the `k` index. How does the timing change? Print the maximum absolute distance between this and the reference result.
3. Use the **`MATMUL`** function. Same questions.
4. Bonus question: investigate the `j,k,i` and `i,k,j` variants. Write them both with array sections and individual array elements. Is there a difference in timing?

Does the optimization level make a difference in timing?

37.6.3 Restricting with `where`

If an array operation should not apply to all elements, you can specify the ones it applies to with a **`where`** statement.

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )
  sequence of array statements
ELSEWHERE
  sequence of array statements
END WHERE
```

37.6.4 Global condition tests

Reduction of a test on all array elements: **`all`**

```
REAL(8),dimension(N,N) :: A
LOGICAL :: positive,positive_row(N),positive_col(N)
positive = ALL( A>0)
positive_row = ALL( A>0,1 )
positive_col = ALL( A>0,2 )
```

Exercise 37.7. Use array statements (that is, no loops) to fill a two-dimensional array `A` with random numbers between zero and one. Then fill two arrays `Abig` and `Asmall` with the elements of `A` that are great than 0.5, or less than 0.5 respectively:

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i,j) = \begin{cases} 0 & \text{if } A(i,j) \geq 0.5 \\ A(i,j) & \text{otherwise} \end{cases}$$

Using more array statements, add `Abig` and `Asmall`, and test whether the sum is close enough to `A`.

Similar to `all`, there is a function `any` that tests if any array element satisfies the test.

```
if ( Any( Abs(A-B)>
```

37.7 Array operations

37.7.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

37.7.1.1 Slicing

If your loop assigns to an array from another array, you can use section notation:

```
a(:) = b(:)
c(1:n) = d(2:n+1)
```

37.7.1.2 'forall' keyword

The `forall` keyword also indicates an array assignment:

```
forall (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end forall
```

You can tell that this is for arrays only, because the loop index has to be part of the left-hand side of every assignment.

What happens if you apply `forall` to a statement with loop-carried dependencies? Consider first the traditional loops

```
A = [1,2,3,4,5]
do i=1,4
   A(i+1) = A(i)
end do
print '(5(i2x))',A
```

```
A = [1,2,3,4,5]
do i=4,1,-1
   A(i+1) = A(i)
end do
print '(5(i2x))',A
```

Can you predict the output? Now consider the following:

Code:

```
A = [1,2,3,4,5]
forall (i=1:4)
   A(i+1) = A(i)
end forall
print '(5(i2x))',A
```

Output
[arrayf] forallf:

```
 1  1  2  3  4
```

Code:

```
A = [1,2,3,4,5]
do i=4,1,-1
   A(i+1) = A(i)
end do
print '(5(i2x))',A
```

Output
[arrayf] forallb:

```
 1  1  2  3  4
```

What does this tell you about the execution?

In other words, this mechanism is prone to misunderstanding and therefore now deprecated. It is not a parallel loop! For that, the following mechanism is preferred.

37.7.1.3 Do concurrent

The *do concurrent* is a true do-loop. With the `concurrent` keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
do concurrent (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end do
```

(Do not use `for all`)

37.7.2 Loops without dependencies

Here are some illustrations of simple array copying with the above mechanisms.

```
do i=2,n
   counted(i) = 2*counting(i-1)
end do
```

```
Original     1   2   3   4   5   6   7   8   9  10
Recursive    0   2   4   6   8  10  12  14  16  18
```

```
counted(2:n) = 2*counting(1:n-1)
```

```
Original     1   2   3   4   5   6   7   8   9  10
Section      0   2   4   6   8  10  12  14  16  18
```

```
forall (i=2:n)
   counted(i) = 2*counting(i-1)
end forall
```

```
Original     1   2   3   4   5   6   7   8   9  10
Forall       0   2   4   6   8  10  12  14  16  18
```

```
do concurrent (i=2:n)
   counted(i) = 2*counting(i-1)
end do
```

```
Original     1   2   3   4   5   6   7   8   9  10
Concurrent   0   2   4   6   8  10  12  14  16  18
```

Exercise 37.8. Create arrays `A,C` of length `2N`, and `B` of length `N`. Now implement

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \ldots N$$

and

$$C_i = A_{i/2}, \quad i = 1, \ldots 2N$$

using all four mechanisms. Make sure you get the same result every time.

37.7.3 Loops with dependencies

For parallel execution of a loop, all iterations have to be independent. This is not the case if the loop has a *recurrence*, and in this case, the 'do concurrent' mechanism is not appropriate. Here are the above four constructs, but applied to a loop with a dependence.

```
do i=2,n
   counting(i) = 2*counting(i-1)
end do
```

```
Original   1   2   3   4   5   6   7   8   9  10
Recursiv   1   2   4   8  16  32  64 128 256 512
```

The slicing version of this:

```
counting(2:n) = 2*counting(1:n-1)
```

```
Original   1   2   3   4   5   6   7   8   9  10
Section    1   2   4   6   8  10  12  14  16  18
```

acts as if the right-hand side is saved in a temporary array, and subsequently assigned to the left-hand side.

Using 'forall' is equivalent to slicing:

```
forall (i=2:n)
   counting(i) = 2*counting(i-1)
end forall
```

```
Original     1   2   3   4   5   6   7   8   9  10
Forall       1   2   4   6   8  10  12  14  16  18
```

On the other hand, 'do concurrent' does not use temporaries, so it is more like the sequential version:

```
do concurrent (i=2:n)
   counting(i) = 2*counting(i-1)
end do
```

```
Original     1   2   3   4   5   6   7   8   9  10
Concurrent   1   2   4   8  16  32  64 128 256 512
```

Note that the result does not have to be equal to the sequential execution: the compiler is free to rearrange the iterations any way it sees fit.

37.8 Review questions

Exercise 37.9. Let the following declarations be given, and assume that all arrays are properly initialized:

```
real                    :: x
real, dimension(10)     :: a, b
real, dimension(10,10)  :: c, d
```

Comment on the following lines: are they legal, if so what do they do?

1. `a = b`
2. `a = x`
3. `a(1:10) = c(1:10)`

How would you:

1. Set the second row of `c` to `b`?
2. Set the second row of `c` to the elements of `b`, last-to-first?

Chapter 38

Pointers

Pointers in C/C++ are based on memory addresses; Fortran pointers on the other hand, are more abstract.

38.1 Basic pointer operations

- Pointer points at an object
- Access object through pointer
- You can change what object the pointer points at.

```
real,pointer :: point_at_real
```

Pointers could also be called 'aliases': they act like an alias for an object of elementary or derived data type. You can access the object through the alias. The difference with actually using the object, is that you can decide what object the pointer points at.

Fortran pointers are often automatically *dereferenced*: if you print a pointer you print the object it references, not some representation of the pointer.

The `pointer` definition

```
real,pointer :: point_at_real
```

defined a pointer that can point at a real variable.

- You have to declare that a variable is pointable:

  ```
  real,target :: x
  ```

- Set the pointer with => notation:

  ```
  point_at_real => x
  ```

Now using `point_at_real` is the same as using `x`.

```
print *,point_at_real ! will print the value of x
```

Pointers can not just point at anything: the thing pointed at needs to be declared as `target`

```
real,target :: x
```

and you use the => operator to let a pointer point at a target:

```
point_at_real => x
```

If you use a pointer, for instance to print it

```
print *,point_at_real
```

it behaves as if you were using the value of what it points at.

Code:

```
real,target :: x,y
real,pointer :: that_real

x = 1.2
y = 2.4
that_real => x
print *,that_real
that_real => y
print *,that_real
y = x
print *,that_real
```

Output
[pointerf] realp:

```
1.20000005
2.40000010
1.20000005
```

1. The pointer points at x, so the value of x is printed.
2. The pointer is set to point at y, so its value is printed.
3. The value of y is changed, and since the pointer still points at y, this changed value is printed.

```
real,pointer :: point_at_real,also_point
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

```
real,pointer :: point_at_real,also_point
```

you can make the target of the one to also be the target of the other:

```
also_point => 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.

Using ordinary assignment does not work, and will give strange memory errors.

Exercise 38.1. Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

Code:

```
real,dimension(10),target :: array &
     = [1.1, 2.2, 3.3, 4.4, 5.5, &
        9.9, 8.8, 7.7, 6.6, 0.0]
real,pointer :: biggest_element

print '(10f5.2)',array
call SetPointer(array,biggest_element)
print *,"Biggest element is",biggest_element
biggest_element = 0
print '(10f5.2)',array
```

Output
[pointerf] arpointf:

```
 1.10 2.20 3.30 4.40 5.50 9.90 8.80 7.70 6.60 0
 Biggest element is   9.89999962
 1.10 2.20 3.30 4.40 5.50 0.00 8.80 7.70 6.60 0
```

- `Nullify`: zero a pointer
- `Associated`: test whether assigned

Associate unnamed memory:

```
Integer,Pointer,Dimension(:) :: array_point
Allocate( array_point(100) )
```

This is automatically deallocated when control leaves the scope.

38.2 Pointers and arrays

You can set a pointer to an array element or a whole array.

```
real(8),dimension(10),target :: array
real(8),pointer              :: element_ptr
real(8),pointer,dimension(:) :: array_ptr

element_ptr => array(2)
array_ptr   => array
```

More surprising, you can set pointers to array slices:

```
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.

38.3 Example: linked lists

For pictures of linked lists, see section 62.1.2.

- Linear data structure
- more flexible 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 38.2. 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.

- Node: value field, and pointer to next node.
- List: pointer to head node.

```
type node
   integer :: value
   type(node),pointer :: next
end type node

type list
   type(node),pointer :: head
end type list

type(list) :: the_list
nullify(the_list%head)
```

A list is based on a simple data structure, a node, which contains a value field and a pointer to another node.

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.

Here are the basic definitions of a node, and a list which is basically a repository for the head node:

```
type node
   integer :: value
   type(node),pointer :: next
end type node

type list
   type(node),pointer :: head
end type list

type(list) :: the_list
nullify(the_list%head)
```

First element becomes the list head:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
the_list%head => new_node
```

Initially, the list is empty, meaning that the 'head' pointer is un-associated. The first time we add an element to the list, we create a node and assign it as the head of the list:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
the_list%head => new_node
```

Keep the list sorted: new largest element attached at the end.

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
current%next => new_node
```

Exercise 38.3. Write function `attach` that takes an integer, and attaches a new node at the end of the list with that value.

Adding a value to a list can be done two ways. If the new element is larger than all elements in the list, a new node needs to be appended to the last one. Let's assume we have managed to let `current` point at the last node of the list, then here is how to attaching a new node from it:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
current%next => new_node
```

Find the insertion point:

```
current => the_list%head ; nullify(previous)
do while ( current%value<value &
     .and. associated(current%next) )
   previous => current
   current => current%next
end do
```

Inserting an element in the list is harder. First of all, you need to find the two nodes, `previous` and `current`, between which to insert the new node:

```
current => the_list%head ; nullify(previous)
do while ( current%value<value &
     .and. associated(current%next) )
   previous => current
   current => current%next
end do
```

The actual insertion requires rerouting some pointers:

```
allocate(new_node)
new_node%value = value
new_node%next => current
previous%next => new_node
```

Exercise 38.4. 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 38.5. Write a `length` function for the linked list.
Try it both with a loop, and recursively.

Chapter 39

Input/output

39.1 Types of I/O

Fortran can deal with input/output in ASCII format, which is called *formatted* I/O, and binary, or *unformatted* I/O. Formatted I/O can use default formatting, but you can specify detailed formatting instructions, in the I/O statement itself, or separately in a `Format` statement.

Fortran I/O can be described as *list-directed I/O*: both input and output commands take a list of item, possibly with formatting specified.

- `Print` simple output to terminal
- `Write` output to terminal or file ('unit')
- `Read` input from terminal or file
- `Open`, `Close` for files and streams
- `Format` format specification that can be used in multiple statements.

- Formatted: ascii output. This is good for reporting, but not for numeric data storage.
- Unformatted: binary output. Great for further processing of output data.
- Beware: binary data is machine-dependent. Use *hdf5* for portable binary.

39.2 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.

39.2.1 Print on one line

The statement

```
print *,item1,item2,item3
```

will print the items on one line, as far as the line length allows.

Parametrized printing with an *implicit do loop*:

```
print *,( i*i,i=1,n)
```

All values will be printed on the same line.

39.2.2 Printing arrays

If you print a variable that is an array, all its elements will be printed, in *column-major* ordering if the array is multi-dimensional.

You can also control the printing of an array by using an *implicit do loop*:

```
print *,( A(i),i=1,n)
```

39.3 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:

```
write (unit,format) data
```

where **format** and **data** are as described above. The new element is the *unit*, which is a numerical indication of an output device, such as a file.

39.3.1 Units

```
Open(11)
```

will result in a file with a name typically `fort11`.

```
Open(11,FILE="filename")
```

Many other options for error handling, new vs old file, etc.

After this:

```
Write (11,fmt) data
```

Again options for errors and such.

39.3.2 Other write options

```
write(unit,fmt,ADVANCE="no") data
```

will not issue a newline.

open Close

39.4 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*.

Indicated by lack of format specification:

```
write (*) data
```

Note: may not be portable between machines.

Chapter 40

Leftover topics

40.1 Interfaces

If you want to use a procedure in your main program, the compiler needs to know the signature of the procedure: how many arguments, of what type, and with what intent. You have seen how the `contains` clause can be used for this purpose if the procedure resides in the same file as the main program.

If the procedure is in a separate file, the compiler does not see definition and usage in one go. To allowed the compiler to do checking on proper usage, we can use an `interface` block. This is placed at the calling site, declaring the signature of the procedure.

Main program:

```
interface
   function f(x,y)
     real*8 :: f
     real*8,intent(in) :: x,y
   end function f
end interface

real*8 :: in1=1.5, in2=2.6, result

result = f(in1,in2)
```

Procedure:

```
function f(x,y)
  implicit none
  real*8 :: f
  real*8,intent(in) :: x,y
```

The `interface` block is not required (an older `external` mechanism exists for functions), but is recommended. It is required if the function takes function arguments.

40.1.1 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.

40.2 Random numbers

In this section we briefly discuss the Fortran *random number generator*. The basic mechanism is through the library subroutine `random_number`, which has a single argument of type `REAL` with `INTENT(OUT)`:

```
real(4) :: randomfraction
call random_number(randomfraction)
```

The result is a random number from the uniform distribution on $[0, 1)$.

Setting the *random seed* is slightly convoluted. The amount of storage needed to store the seed can be processor and implementation-dependent, so the routine `random_seed` can have three types of named argument, exactly one of which can be specified at any one time. The keyword can be:

- `SIZE` for querying the size of the seed;
- `PUT` for setting the seed; and
- `GET` for querying the seed.

A typical fragment for setting the seed would be:

```
integer :: seedsize
integer,dimension(:),allocatable :: seed

call random_seed(size=seedsize)
allocate(seed(seedsize))
seed(:) = ! your integer seed here
call random_seed(put=seed)
```

40.3 Timing

Timing is done with the `system_clock` routine.

- This call gives an integer, counting clock ticks.
- To convert to seconds, it can also tell you how many ticks per second it has: its *timer resolution*.

```
integer :: clockrate,clock_start,clock_end
call system_clock(count_rate=clockrate)
print *,"Ticks per second:",clockrate

call system_clock(clock_start)
! code
call system_clock(clock_end)
print *,"Time:",(clock_end-clock_start)/REAL(clockrate)
```

Chapter 41

Fortran review questions

41.1 Fortran versus C++

Exercise 41.1. 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.

41.2 Basics

Exercise 41.2.

- What does the `Parameter` keyword do? Give an example where you would use it.
- Why would you use a `Module`?
- What is the use of the `Intent` keyword?

41.3 Arrays

Exercise 41.3. You are looking at historical temperature data, say a table of the high and low temperature at January 1st of every year between 1920 and now, so that is 100 years.

Your program accepts data as follows:

```
Integer :: year, high, low

!! code omitted

read *,year,high,low
```

where the temperatures are rounded to the closest degree (Centigrade of Fahrenheit is up to you.)

Consider two scenarios. For both, give the lines of code for 1. the array in which you store the data, 2. the statement that inserts the values into the array.

- Store the raw temperature data.
- Suppose you are interested in knowing how often certain high/low temperatures occurred. For instance, 'how many years had a high temperature of 32F /0 C'.

41.4 Subprograms

Exercise 41.4. Write the missing procedure `pos_input` that

- reads a number from the user
- returns it
- and returns whether the number is positive

in such a way to make this code work:

Code:

```
program looppos
  implicit none
  real(4) :: userinput
  do while (pos_input(userinput))
     print &
      '("Positive input:",f7.3)',&
      userinput
  end do
  print &
      '("Nonpositive input:",f7.3)',&
      userinput
  !! ...
end program looppos
```

Output
[funcf] looppos:

```
Running with the following inputs:

5
1
-7.3

/bin/sh: ./looppos: No such file or directory
make[3]: *** [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 42

Basics of Julia

42.1 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.

- A program contains statements. A simple statement is one that fits on one line. No need for a semicolon: the line end terminates it.

```
a = 1
```

- You can multiple statements on one line by using semicolons:

```
a = 1; b = 2
```

- Compound statements, such as conditionals, take up mutiple lines;

```
if a == 1
  println("It was one")
end
```

- Comments are 'Note to self', short:

```
c = 3 # set c to three
```

The basics of interacting with a program are through the output statements `print` and `println`, and input `readline`.

```
a = 1; b = 2
print("a and b: ")
print(a)
print(" ")
println(b)
```

or simpler:

```
println("a and b: $a $b")
```

42.2 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 placed in memory.

Variable names are more liberal than in other programming languages: many *Unicode* characters are allowed.

42.2.1 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.

The main numerical types are `Int64` and `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' `Real`, which is a supertype of both; see later.

Types can be determined by looking at the rhs expression, or explicitly forced:

Code:

```
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))")
```

Output
[basicj] scalar:

```
1 has type Int64
1 has type Int64
1.0 has type Float64
1.0 has type Float64
```

Types have a hierarchy.
The subtype relation is denoted with the `<:` relation:

```
Integer <: Number
```

This can also be used in functions:

```
function somecompute(x::T) where T <: Real
```

42.2.1.1 Boolean values

So far you have seen integer and real variables. There are also boolean values, type `Bool`, which represent truth values.

```
primitive type Bool <: Integer 8 end
```

There are only two values: `true` and `false`.

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

To create an array of booleans you can

```
alltrue = trues(n)
allfalse = falses(n)
```

42.2.1.2 Characters and strings

- Characters, of type `Char`, in single quotes
- Strings, of type `String`, in double quotes.

In addition to the regular 128 *ASCII* characters, *Unicode* characters can be entered through a backslash notation: `\gamma`. This follows LaTeX naming conventions.

42.2.2 Truth values

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

- Testing: `==` `!=` `<` `>` `<=` `>=`
- Not, and, or: `!` `&&` `||`

As in other languages, booleans can be implicitly interpreted as integer, with `true` corresponding to 1 and `false` to 0. However, conversion the other way is not as generous.

```
1+true
# evaluates to 2
2 || 3
# is a TypeError
```

Logical expressions in C++ are evaluated using *shortcut operators*: you can write

```
x>=0 && 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.

Bitwise: `&` `|` `^`

42.3 Expressions

Division `2/3` gives a `Float64` number. You can cast to integer:

```
half = Int64( (27+2)/2 )
```

42.3.1 Terminal input

To make a program run dynamic, you can set starting values from keyboard input. For this, use `readline`, which takes keyboard input. To assign this input to a variable you need to parse that as the desired type, using the function `parse`.

```
n = parse(Int64,readline(stdin))
```

42.4 Tuples

```
(a,b) = (1,2)
```

even

```
(a,b) = (b,a)
```

Chapter 43

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*.

43.1 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 43.1. For what values of x will the left code print 'b'?
For what values of x will the right code print 'b'?

```
x = ## some float value
if x > 1
   println("a")
   if x > 2
     println("b")
   end
end
```

```
x = ## some float value
if x > 1
   println("a"
elseif x > 2
   println("b")
end
```

43.2 Advanced topics

43.2.1 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.

43.2.2 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 = foo ? 5 : 6;
```

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

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

Chapter 44

Looping

See chapter 6 for a general introduction to loops.

44.1 Basics

Julia loops have a structure:

```
for var = some_range
  # statements
end

# or

for var in some_range
  # statements
end
```

Code:

```
function sum_of_cubes(n)
  cubesum = 0.
  for num in 1:n
    cubesum += num*num*num
  end
  return cubesum
end

n = 10
sumofcubes = sum_of_cubes(n)
println("Sum of $n cubes: $sumofcubes")
```

Output
[loopj] loop:

```
0
Sum of 10 squares: 385
Sum of 10 cubes: 3025.0
```

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`:

Code:

```
sum = 0; n = 10
println(sum)
for num in 1:n
  global sum
  sum += num*num
end
println("Sum of $n squares: $sum")
```

Output
[loopj] loop:

```
0
Sum of 10 squares: 385
Sum of 10 cubes: 3025.0
```

Note that we said 'scalar' variable. This does not hold for array variable: those are always global.

44.1.1 Stride

Loops by default have a stride of 1. To use something else, write

```
for odd=1:2:10
  f(odd)
end
```

or

```
for down=10:-1:1
  f(down)
end
```

Exercise 44.1. Read an integer value with `cin`, and print 'Hello world' that many times.

Exercise 44.2. Extend exercise 6.1: the input 17 should now give lines

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

Loops such as

```
for  i in 1:10
```

have well-defined starting and ending values. However, loops can also iterate over more abstractly defined iteration spaces. You will see this later.

44.1.2 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.

```
for i in 1:10
  for j in 1:10
  ...
  end
end
```

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

For a more compact syntax:

```
for i in 1:10, j in 1:10
  ...
end
```

This syntax makes it easier to `break` out of multiple loops.

Chapter 45

Functions

Read the introduction of chapter 7 for a general discussion of what functions are about.

Here is a single-line function definition:

Code:

```
double(x) = 2*x
println(double(6))
```

Output
[funcj] simpleone:

```
12
```

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* . Thus, a function such as `sin` can apply to both a scalar and matrix.

This (multiline) function prints out its argument type:

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:

```
Arg type Int64: 15
30
Arg type Float64: 15.1
30.2
Arg type Array{Int64,1}: [1, 2]
[2, 4]
```

You can enforce an argument type

```
myfunction( x::Float64 ) = #
```

Chapter 46

Arrays

Reference: https://docs.julialang.org/en/v1/manual/arrays/

46.1 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)
```

46.1.1 Element access

Julia uses the square bracket notation

```
ar[5] = 6
```

with bounds that are 1-based.

46.1.2 Vectors and matrices

The keywords `Vector` and `Matrix` are synonyms for

```
Array{T,1}
Array{T,2}
```

respectively.

46.1.3 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:

```
function look_at_matrix( A::AbstractMatrix )
  (size,type) = ( size(A,1),eltype(A) )
```

46.2 Functions

46.2.1 Built-in functions

`collect`: turn the elements of a collection or iterator into an array. Example:

```
integers = collect( 1:n )
```

46.2.2 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:

Code:

```
x = [2,4,-6,8,1,-3,5,7,9]
println("$x")
xmax = abs.(x)
println("$xmax")
```

Output
[arrayj] pointwise:

```
[2, 4, -6, 8, 1, -3, 5, 7, 9]
[2, 4, 6, 8, 1, 3, 5, 7, 9]
```

46.2.3 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:

Extend with `push!`:

Code:

```
foo = [1,2,3];
push!(foo, 4, 5, 6);
```

Output
[arrayj] push:

```
[1, 2, 3, 4, 5, 6]
```

46.3 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

(0,100,200)
(1,101,201)
(2,102,202)
(3,103,203)
(4,104,204)
(5,105,205)
(6,106,206)
(7,107,207)
(8,108,208)
(9,109,209)
(10,110,210)
```

You can iterators:

```
ro = 0:2:100
[i for i in ro]
```

46.4 Multi-dimensional cases

Julia has support for multi-dimensional arrays. Array types look like

```
Array{Int64,2}
```

for a two-dimensional array of `Int64` objects.

Arrays are organized by columns. For instance using `reshape` to make a two-dimensional array from one-dimensional one:

```
a = reshape(1:100, (10, 10))
for n in a
    print(n, " ")
end
```

Chapter 47

Structs

47.1 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.

```
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.

47.2 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.

```
mutable struct vector
  x::Float64
  y::Float64
  label::Int64
end
```

47.3 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.

47.3.1 Structures versus objects

Multiple dispatch on structures are Julia's alternative to **OOP!** (**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 47.1. If you are doing the prime project (chapter 50) you can now do exercise 50.8.

Chapter 48

More Julia

48.1 Assertions

Test on dynamic conditions

```
function twice_root( x::T )
  @assert( x>0 )
  return 2*sqrt()
end
```

PART V

EXERCISES AND PROJECTS

Chapter 49

Style guide for project submissions

The purpose of computing is insight, not numbers. (Richard Hamming)

Your project writeup is at least as 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 First of all, observe correct spelling and grammar. Use full sentences.

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.

Summary Summarize your work and findings.

Chapter 50

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.

50.1 Arithmetic

Before doing this section, make sure you study section 4.5.

Exercise 50.1. 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.

50.2 Conditionals

Before doing this section, make sure you study section 5.1.

Exercise 50.2. Read two numbers and print a message stating whether the second is as divisor of the first:

Code:

```
int number,divisor;
bool is_a_divisor;
/* ... */
if (
/* ... */
    ) {
  cout << "Indeed, " << divisor
       << " is a divisor of "
       << number << endl;
} else {
  cout << "No, " << divisor
       << " is not a divisor of "
       << number << endl;
}
```

Output
[primes] division:

```
( echo 6 ; echo 2 ) | 1divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) | 1divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

50.3 Looping

Before doing this section, make sure you study section 6.1.

Exercise 50.3. Read an integer and determine whether it is prime by testing for the smaller numbers whether they are a divisor of 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 50.4. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

Exercise 50.5. Read in an integer r. If it is prime, print a message saying so. If it is not prime, find integers $p \leq q$ so that $r = p \cdot q$ and so that p and q are as close together as possible. For instance, for $r = 30$ you should print out `5,6`, rather than `3,10`. You are allowed to use the function `sqrt`.

50.4 Functions

Before doing this section, make sure you study section 7.

Above you wrote several lines of code to test whether a number was prime.

Exercise 50.6. Write a function `test_if_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```
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?

50.5 While loops

Before doing this section, make sure you study section 6.2.

Exercise 50.7. 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.)

50.6 Structures

Before doing this section, make sure you study section 9.1, 14.1.

A `struct` functions to bundle together a number of data item. We only discuss this as a preliminary to classes.

Exercise 50.8. Rewrite the exercise that found a predetermined number of primes, putting the `number_of_primes_` and `last_number_tested` variables in a structure. Your main program should now look like:

```
cin >> nprimes;
struct primesequence sequence;
while (sequence.number_of_primes_found<nprimes) {
  int number = nextprime(sequence);
  cout << "Number " << number << " is prime" << endl;
}
```

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?

50.7 Classes and objects

Before doing this section, make sure you study section 10.1.

In exercise 50.8 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 50.9. Write a class `primegenerator` that contains

- members `how_many_primes_found` and `last_number_tested`,
- a method `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << endl;
}
```

In the previous exercise you defined the `primegenerator` class, and you made one object of that class:

```
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other.

Exercise 50.10. 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.

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 50.11. The *Goldbach conjecture* says that every even number $2n$ (starting at 4), is the sum of two primes $p + q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r \text{ prime}} \exists_{p,q \text{ prime}} : r = (p+q)/2 \text{ is prime.}$$

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?

50.7.1 Exceptions

Before doing this section, make sure you study section 22.2.2.

Exercise 50.12. Revisit the pime generator class (exercise 50.9) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 23.7.)

Code:

```
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
```

50.7.2 Prime number decomposition

Before doing this section, make sure you study section 23.2.1.

Design a class `Integer` which stores its value as its prime number decomposition. For instance,

$$180 = 2^2 \cdot 3^3 \cdot 5 \qquad \Rightarrow \qquad \texttt{[2:2, 3:2, 5:1]}$$

You can implement this decomposition itself as a *vector*, (the i-th location stores the exponent of the i-th prime) but let's use a `map` instead.

Exercise 50.13. 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:

```
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 50.14. Extend the previous exercise to having multiplicity > 1 for the prime factors.

Code:

```
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 `Integer`s.

Implement a class `Rational` for rational numbers, which are implemented as two `Integer` objects. This class should have methods for addition and multiplication. Write these through operator overloading if you've learned this.

Make sure you always divide out common factors in the numerator and denominator.

50.8 Eratosthenes sieve

The Eratosthenes sieve is an algorithm for prime numbers that step-by-step filters out the multiples of any prime it finds.

1. Start with the integers from 2: $2, 3, 4, 5, 6, \ldots$
2. The first number, 2, is a prime: record it and remove all multiples, giving

   $$3, 5, 7, 9.11, 13, 15, 17 \ldots$$

3. The first remaining number, 3, is a prime: record it and remove all multiples, giving

   $$5, 7, 11, 13, 17, 19, 23, 25, 29 \ldots$$

4. The first remaining number, 5, is a prime: record it and remove all multiples, giving

   $$7, 11, 13, 17, 19, 23, 29, \ldots$$

50.8.1 Arrays implementation

The sieve is easily implemented with an array that stores all integers.

Exercise 50.15. Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers. Apply the sieve algorithm to find the prime numbers.

50.8.2 Streams implementation

The disadvantage of using an array is that we need to allocate an array. What's more, the size is determined by how many integers we are going to test, not how many prime numbers we want to generate. We are going to take the idea above of having a generator object, and apply that to the sieve algorithm: we will now have multiple generator objects, each taking the previous as input and erasing certain multiples from it.

Exercise 50.16. Write a `stream` class that generates integers and use it through a pointer.

Code:

```
for (int i=0; i<7; i++)
  cout << "Next int: "
       << the_ints->next() << 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 50.17. Write a class `filtered_stream` with a constructor

```
filtered_stream(int filter,shared_ptr<stream> input);
```

that

1. Implements `next`, giving filtered values,
2. by calling the `next` method of the input stream and filtering out values.

Code:

```
auto integers =
  make_shared<stream>();
auto odds =
  shared_ptr<stream>
  ( new filtered_stream(2,integers) );
for (int step=0; step<5; step++)
  cout << "next odd: "
       << odds->next() << endl;
```

Output

[sieve] odds:

```
next odd: 3
next odd: 5
next odd: 7
next odd: 9
next odd: 11
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

Exercise 50.18. Write a program that generates prime numbers as follows.

- Maintain a `current` stream, that is initially the stream of prime numbers.
- Repeatedly:
  - Record the first item from the current stream, which is a new prime number;
  - and set `current` to a new stream that takes `current` as input, filtering out multiples of the prime number just found.

50.9 Range implementation

Before doing this section, make sure you study section 24.2.

Exercise 50.19. Make a `primes` class that can be ranged:

Code:

```
primegenerator allprimes;
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout << endl;
```

Output

[primes] range:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
```

Chapter 51

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 10.

51.1 Basic functions

Exercise 51.1. 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 51.2. Write a function with inputs x, y, θ that alters x and y corresponding to rotating the point (x, y) over an angle θ.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should behave like:

Code:

```
const float pi = 2*acos(0.0);
float x{1.}, y{0.};
rotate(x,y,pi/4);
cout << "Rotated halfway: ("
     << x << "," << y << ")" << endl;
rotate(x,y,pi/4);
cout << "Rotated to the y-axis: ("
     << x << "," << y << ")" << endl;
```

Output
[geom] rotate:

```
Rotated halfway: (0.707107,0.707107)
Rotated to the y-axis: (0,1)
```

51.2 Point class

Before doing this section, make sure you study section 10.1.

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 51.3. Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `printout` uses `cout` to display the point.
- `angle` computes the angle of vector (x, y) with the x-axis.

Exercise 51.4. Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

```
p.distance(q)
```

computes the distance between them.

Hint: remember the 'dot' notation for members.

Exercise 51.5. Write a method `halfway_point` that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p + q)/2$:

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions `Add` and `Scale` and combine these. (Later you will learn about operator overloading.)

How would you print out a *Point* to make sure you compute the halfway point correctly?

Exercise 51.6. Make a default constructor for the point class:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

but which gives an indication that it is undefined:

Code:

```
Point p3;
cout << "Uninitialized point:"
     << endl;
p3.printout();
cout << "Using uninitialized point:"
     << endl;
auto p4 = Point(4,5)+p3;
p4.printout();
```

Output
[geom] linearnan:

```
Uninitialized point:
Point: nan,nan
Using uninitialized point:
Point: nan,nan
```

Hint: see section 4.6.2.

Exercise 51.7. Revisit exercise 51.2 using the `Point` class. Your code should now look like:

```
newpoint = point.rotate(alpha);
```

Exercise 51.8. Advanced. Can you make a `Point` class that can accomodate any number of space dimensions? Hint: use a `vector`; section 11.3. Can you make a constructor where you do not specify the space dimension explicitly?

51.3 Using one class in another

Before doing this section, make sure you study section 10.2.

Exercise 51.9. Make a class `LinearFunction` with a constructor:

```
LinearFunction( Point input_p1,Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 51.10. Make a class `LinearFunction` with two constructors:

```
LinearFunction( Point input_p2 );
LinearFunction( Point input_p1,Point input_p2 );
```

where the first stands for a line through the origin.
Implement again the `evaluate` function so that

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 51.11. Revisit exercises 51.2 and 51.7, introducing a `Matrix` class. Your code can now look like

```
newpoint = point.apply(rotation_matrix);
```

or

```
newpoint = rotation_matrix.apply(point);
```

Can you argue in 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.

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 51.12.

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   Rectangle(Point bl,float w,float h);
   ```

   The logical implementation is to store these quantities. Implement methods

   ```
   float area(); float rightedge(); float topedge();
   and write a main program to test these.
   ```

2. Add a second constructor

   ```
   Rectangle(Point bl,Point tr);
   ```

   Can you figure out how to use member initializer lists for the constructors?
3. Make a copy of your file, 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.

51.4 Is-a relationship

Before doing this section, make sure you study section 10.3.

Exercise 51.13. 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 51.14. 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.

51.5 More stuff

Before doing this section, make sure you study section 14.3.

The `Rectangle` class stores at most one corner, but may be convenient to sometimes have an array of all four corners.

Exercise 51.15. Add a method

```
const vector<Point> &corners()
```

to the `Rectangle` class. The result is an array of all four corners, not in any order. Show by a compiler error that the array can not be altered.

Chapter 52

Infectuous disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

52.1 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.

52.1.1 Other ways of modeling

Instead of capturing every single person in code, a 'contact network' model, it is possible to use an ODE approach to disease modeling. You would then model the percentage of infected persons by a single scalar, and derive relations for that and other scalars [2].

`http://mathworld.wolfram.com/Kermack-McKendrickModel.html`

This is known as a 'compartmental model', where each of the three SIR states is a compartment: a section of the population. Both the contact network and the compartmental model capture part of the truth. In

fact, they can be combined. We can consider a country as a set of cities, where people travel between any pair of cities. We then use a compartmental model inside a city, and a contact network between cities.

In this project we will only use the network model.

52.2 Coding up 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 52.1. Write a `Person` class with methods:

- `status_string()` : returns a description of the person's state as a `string`;
- `update()` : update the person's status to the next day;
- `infect(n)` : infect a person, with the disease to run for n days;
- `is_stable()` : return a `bool` indicating whether the person has been sick and is recovered.

Your main program could for instance look like:

```
Person joe;

int step = 1;
for ( ; ; step++) {

  joe.update();
  float bad_luck = (float) rand()/(float)RAND_MAX;
  if (bad_luck>.95)
    joe.infect(5);

  cout << "On day " << step << ", Joe is "
       << joe.status_string() << endl;
  if (joe.is_stable())
    break;
}
```

Here is a suggestion how you can model disease status. Use a single integer with the following interpretation:

- healthy but not vaccinated, value 0,
- recovered, value -1,
- vaccinated, value -2,
- and sick, with n days to go before recovery, modeled by value n.

The `Person::update` method then updates this integer.

Remark 12 *Consider a proint of programming style. Now that you've modeled the state of a person with an integer, you can use that as*

```
void infect(n) {
    if (state==0)
        state = n;
}
```

But you can also write

```
bool is_susceptible() {
  return state==0;
}
void infect(n) {
    if (is_susceptible())
        state = n;
}
```

Which do you prefer and why?

52.3 Population

Next we need a `Population` class. Implement a population as a `vector` consisting of `Person` objects. Initially we only infect one person, and there is no transmission of the disease.

The trace output should look something like:

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

Exercise 52.2. Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:

  ```
  Population population(npeople);
  ```

- Write a method that infects a random person:

  ```
  population.random_infection();
  ```

- 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.

52.4 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 52.3. Read in a number $0 \leq p \leq 1$ representing the probability of disease transmission upon contact. Incorporate this into the program: in each step the direct neighbours of an infected person can now get sick themselves.

```
population.set_probability_of_transfer(probability);
```

Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

Exercise 52.4. 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?

52.5 Spreading

To make the simulation more realistic, we let every sick person come into contact with a fixed number of random people every day. This gives us more or less the *SIR model*; https://en.wikipedia.org/wiki/Epidemic_model.

Set the number of people that a person comes into contact with, per day, to 6 or so. (You can also let this be an upper bound for a random value, but that does not essentially change the simulation.) You have already programmed the probability that a person who comes in contact with an infected person gets sick themselves. Again start the simulation with a single infected person.

Exercise 52.5. 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?

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.

52.6 Diseases without vaccine: Ebola and Covid-19

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.

52.7 Project writeup and submission

52.7.1 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. This requires you to use *separate compilation* for building the program, and you need a *header* file. See section 18.1.2 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*.

52.7.2 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 last exercise asks you to explore the program behaviour as a function of one or more parameters. Include a table to report on the behaviour you found. You can use Matlab or Matplotlib in Python (or even Excell) to plot your data, but that is not required.

52.8 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 λ_i is the product of the number of infected people and a constant that reflects the number of meetings and the infectuousness 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 52.6. Code this scheme. What is the effect of varying dt?

Exercise 52.7. For the disease to become an epidemic, the number of newly infected has to be larger than the number of recovered. That is,

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma/\lambda.$$

Can you observe this in your simulations?

The parameter γ has a simple interpretation. Suppose that a person stays ill for δ 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, γ is the reciprocal of the duration of the infection in a given person.

Chapter 53

Google PageRank

53.1 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 datacenter. 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 53.1. Make a class `Page` which initially just contains the name of the page. Write a method to display the page. Since we will be using pointers quite a bit, let this be the intended code for testing:

```
auto homepage = make_shared<Page>("My Home Page");
cout << "Homepage has no links yet:" << 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:

```
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 53.2. Add some more links to your homepage. Write a method `random_click` for the `Page` class. Intended code:

```
for (int iclick=0; iclick<20; iclick++) {
  auto newpage = homepage->random_click();
  cout << "To: " << newpage->as_string() << endl;
}
```

How do you handle the case of a page without links?

53.2 Clicking around

Exercise 53.3. Now make a class `Web` which foremost contains a bunch (technically: a `vector`) of pages. Or rather: of pointers to pages. Since we don't want to build a whole internet by hand, let's have a method `create_random_links` which makes a random number of links to random pages. Intended code:

```
Web internet(netsize);
internet.create_random_links(avglinks);
```

Now we can start our simulation. Write a method `Web::random_walk` that takes a page, and the length of the walk, and simulates the result of randomly clicking that many times on the current page. (Current page. Not the starting page.)

Let's start working towards PageRank. First we see if there are pages that are more popular than others. You can do that by starting a random walk once on each page. Or maybe a couple of times.

Exercise 53.4. 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 53.5. Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

```
    vector<int> landing_counts(internet.number_of_pages(),0);
    for ( auto page : internet.all_pages() ) {
      for (int iwalk=0; iwalk<5; iwalk++) {
        auto endpage = internet.random_walk(page,2*avglinks,tracing);
        landing_counts.at(endpage->global_ID())++;
      }
    }
```

Display the results and analyze. You may find that you finish on certain pages too many times. What's happening? Fix that.

53.3 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 53.6. 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.

53.4 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() << endl;
```

Output

[google] pdfsetup:

```
Initial distribution: 0:0.00, 1:0.02, 2:0.07,
```

Exercise 53.7. 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 53.8. Write the method

```
ProbabilityDistribution Web::globalclick
    ( ProbabilityDistribution currentstate );
```

Test it by

- start with a distribution that is nonzero in exactly one page;
- print the new distribution corresponding to one click;
- do this for several pages and inspect the result visually.

Then start with a random distribution and run a couple of iterations. How fast does the process converge? Compare the result to the random walk exercise above.

Exercise 53.9. 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 53.10. 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?

53.5 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 53.11. 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 54

Redistricting

In this project you can explore 'gerrymandering', the strategic drawing of districts to give a minority population a majority of districts[1].

54.1 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*.

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.

54.2 Basic functions

54.2.1 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 54.1. Implement a `Voter` class. You could for instance let ± 1 stand for `A/B`, and 0 for undecided.

Code:

```
  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;
```

Output

```
[gerry] voters:
Voter 5 is positive:
5:+

Voter 6 is negative:
6:-

Voter 7 is weird:
Illegal affiliation value: 3
Error in creating voter 7
```

missing snippet voterneg missing snippet voterwrong

54.2.2 Populations

Exercise 54.2. 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:

```
    cout << "Making district with one B voter" << endl;
    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
```

Exercise 54.3. Implement a `Population` class that will initially model a whole state.

Code:

```
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.

54.2.3 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 54.4. Write a class `Districting` that stores a `vector` of `District` objects. Write `size` and `lean` methods:

Code:

```
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 54.5. Write methods to extend a `Districting`:

```
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) );
gerry = gerry.extend_with_new_district( Voter(4,+1) );
cout << ".. number of districts: " << gerry.size() << endl;
cout << ".. lean: " << gerry.lean() << endl;
```

54.3 Strategy

Now we need a method for districting a population:

```
Districting Population::minority_rules( int ndistricts );
```

Rather than generating all possible partitions of the population, we take an incremental approach (this is related to the solution strategy called *dynamic programming*):

- The basic question is to divide a population optimally over n districts;
- We do this recursively by first solving a division of a subpopulation over $n-1$ districts,
- and extending that with the remaining population as one district.

This means that you need to consider all the ways of having the 'remaining' population into one district, and that means that you will have a loop over all ways of splitting the population, outside of your recursion; see figure 54.1.

- For all $p = 0, \ldots n-1$ considering splitting the state into $0, \ldots, p-1$ and $p, \ldots, n-1$.
- Use the best districting of the first group, and make the last group into a single district.
- Keep the districting that gives the strongest minority rule, over all values of p.

You can now realize the above simple example:

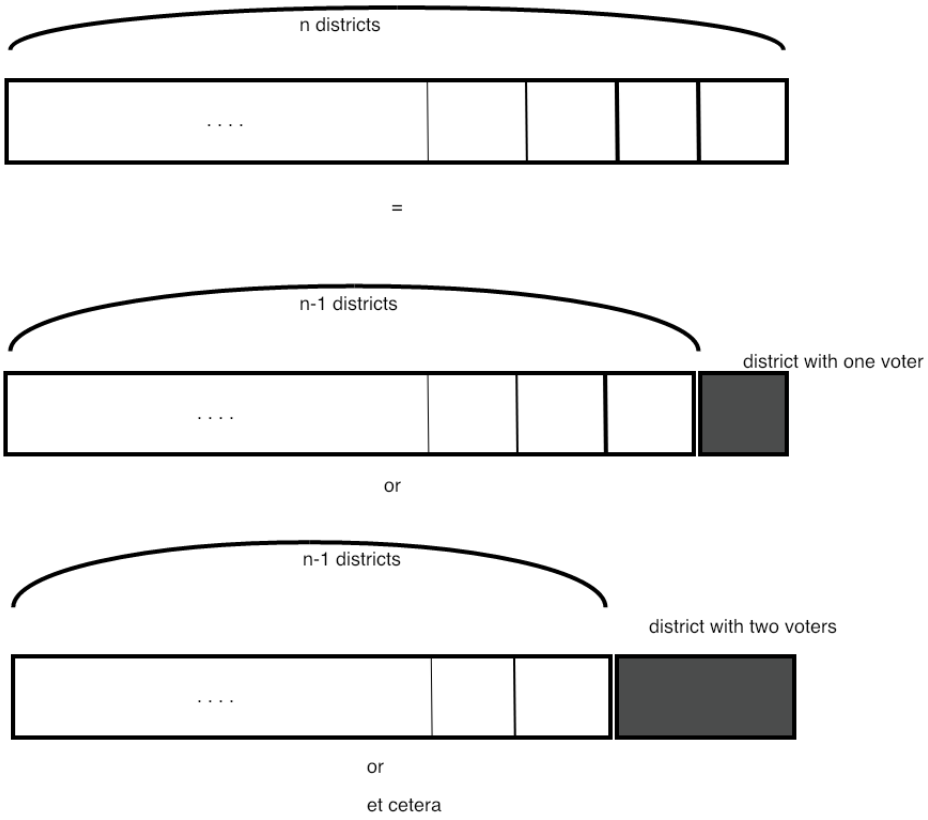


Figure 54.1: Multiple ways of splitting a population

```
AAABB => AAA|B|B
```

Exercise 54.6. Implement the above scheme.

Code:

```
Population five("+++--");
cout << "Redistricting population: " << endl
     << five.print() << endl;
cout << ".. majority rule: "
     << five.rule() << 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 accomodate $n - 1$ voters.

Exercise 54.7. Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

54.4 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 54.8. Improve your implementation by storing and reusing results for the initial sub-populations.

In a way, we solved the program backward: we looked at making a district out of the last so-many voters, and then recursively solving a smaller problem for the first however-many voters. But in that process, we decided what is the best way to assign districts to the first 1 voter, first 2, first 3, et cetera. Actually, for more than one voter, say five voters, we found the result on the best attainable minority rule assigning these five voters to one, two, three, four districts.

The process of computing the 'best' districting forward, is known as *dynamic programming*. The fundamental assumption here is that you can use intermediate results and extend them, without having to reconsider the earlier problems.

Consider for instance that you've considered districting ten voters over up to five districts. Now the majority for eleven voters and five districts is the minimum of

- ten voters and five districts, and the new voter is added to the last district; or
- ten voters and four districts, and the new voter becomes a new district.

Exercise 54.9. Code a dynamic programming solution to the redistricting problem.

54.5 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 54.10. 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 54.11. Look up the definition of efficiency gap (and ‘wasted votes’), and implement it in your code.

Chapter 55

Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

55.1 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.

55.2 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.

55.2.1 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 55.1. Code a class `Address` with the above functionality, and test it.

Code:

```
    Address one(1.,1.),
      two(2.,2.);
    cerr << "Distance: "
         << one.distance(two)
         << "\n";
```

Output

```
[amazon] address:
Address
Distance: 1.41421
.. address
Address 1 should be closest to the depot. Check

Route from depot to depot:  (0,0) (2,0) (1,0)
 has length 8: 8
Greedy scheduling:  (0,0) (1,0) (2,0) (3,0) (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)
 length=8
flip middle two addresses:  (0,0) (-2,0) (1,0)
 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 o
Single route has length: 2078.43
 .. new route accepted with length 2076.65
Final route has length 2076.65 over initial 207
TSP route has length 1899.4 over initial 2078.4

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

Two routes
Total route has greedy length 2677.46 and TSP l
greedy routes have length 1817.21, 1932.43
start with 1817.21,1932.43
Pass 0
.. down to 1813.43,1926.55
.. down to 1811.74,1924.52
.. down to 1803.16,1893.5
.. down to 1786.61,1890.77
.. down to 1784.83,1888.22
.. down to 1779.53,1885.13
.. down to 1779.1,1884.69
.. down to 1774.08,1849.38
.. down to 1763.01,1849.15
.. down to 1759.65,1835.71
.. down to 1750.57,1830.57
.. down to 1748.53,1827.73
.. down to 1747.9,1814.22
.. down to 1744.81,1808.56
```

Next we need a class `AddressList` that contains a list of addresses.

Exercise 55.2. 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 closests to another address, presumably not on the list.

55.2.2 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.

55.2.3 Greedy construction of a route

Next we need to construct a route. Rather than solving the full TSP, we start by employing a *greedy search* strategy:

> Given a point, find the next point by some local optimality test, such as shortest distance. Never look back to revisit the route you have constructed so far.

Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let's write a method

```
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

Exercise 55.3. Write the `greedy_route` method for the `AddressList` class.

1. Assume that the route starts at the depot, which is located at $(0, 0)$. Then incrementally construct a new list by:
2. Maintain an `Address` variable `we_are_here` of the current location;
3. repeatedly find the address closest to `we_are_here`.

Extend this to a method for the `Route` class by working on the subvector that does not contain the final element.

Test it on this example:

Code:

```
Route deliveries;
deliveries.add_address( Address(0,5) );
deliveries.add_address( Address(5,0) );
deliveries.add_address( Address(5,5) );
cerr << "Travel in order: " << deliveries.length() << "\n";
assert( deliveries.size()==5 );
auto route = deliveries.greedy_route();
assert( route.size()==5 );
auto len = route.length();
cerr << "Square route: " << route.as_string()
     << "\n has length " << len << "\n";
```

Output
[amazon] square5:

```
Travel in order: 24.1421
Square route:  (0,0) (0,5) (5,5) (5,0) (0,0)
 has length 20
```

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each adress 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.

55.3 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.

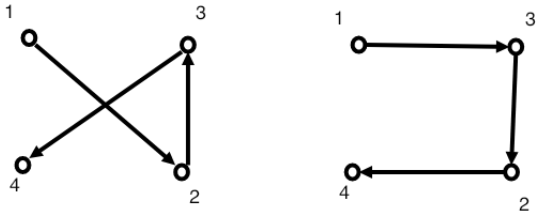


Figure 55.1: Illustration of the 'opt2' idea of reversing part of a path

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 55.1 shows that the path $1--2--3--4$ can be made shorter by reversing part of it, giving $1--3--2--4$. Since recognizing where a path crosses itself can be hard, or even impossible for graphs that don't have Cartesian coordinates associated, we adopt a scheme:

```
for all nodes m<n on the path [1..N]:
  make a new route from
```

```
  [1..m-1] + [m--n].reversed + [n+1..N]
if the new route is shorter, keep it
```

Exercise 55.4. 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 55.5. What is the runtime complexity of this heuristic solution?

Exercise 55.6. 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.

55.4 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 55.2: instead of modifying one path, we could switch bits out bits between one path and another. When you write the code, take into account that the other path may be running backwards! This means that based on split points in the first and second path you know have four resulting modified paths to consider.

Exercise 55.7. Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 55.3.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

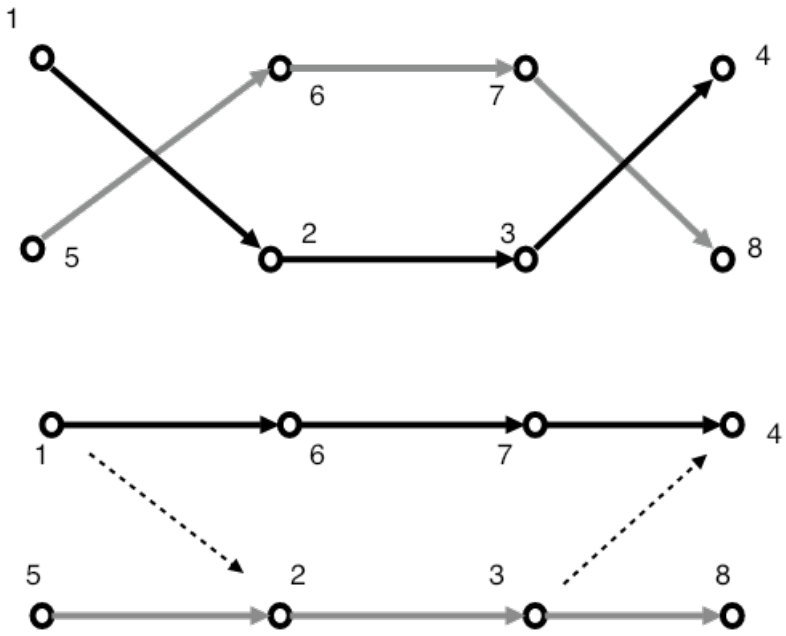


Figure 55.2: Extending the ‘opt2’ idea to multiple paths

55.5 Amazon prime

In section 55.4 you made the assumption that it doesn’t matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

Exercise 55.8. 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.

55.6 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 55.9. Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

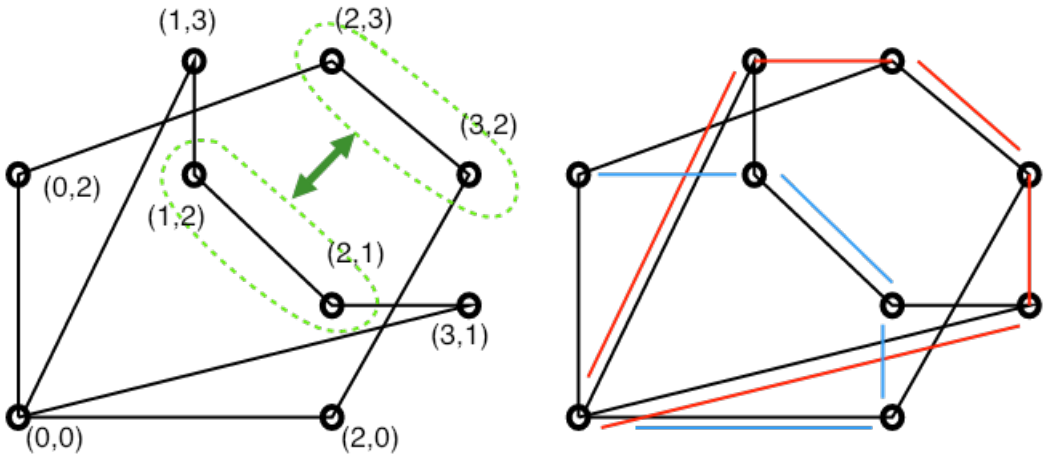


Figure 55.3: Multiple paths test case

Chapter 56

DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

56.1 Basic functions

Refer to section 23.4.

First we set up some basic mechanisms.

Exercise 56.1. There are four bases, `A,C,G,T`, and each has a complement: `A` ↔ `T`, `C` ↔ `G`. Implement this through a `map`, and write a function

```
char BaseComplement(char);
```

Exercise 56.2. 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.

56.2 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 **SCS!** (**SCS!**).

56.2.1 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.

56.2.2 De Bruijn graph assembly

56.3 '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.

56.3.1 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 56.3. Code up the naive algorithm for matching a read. Test it on fake reads obtained by copying a substring from the genome. Use the genome in `phix.fa`.

Now read the *Fastq* file `ERR266411_1.first1000.fastq`. Fastq files contains groups of four lines: the second line in each group contains the reads. How many of these reads are matched to the genome?

Reads are not necessarily a perfect match; in fact, each fourth line in the fastq file gives an indication of the 'quality' of the corresponding read. How many matches do you get if you take a substring of the first 30 or so characters of each read?

56.3.2 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 **BM!** (**BM!**), we still find match locations left-to-right, but we do our matching right-to-left.

```
   vvvv match location
antidisestablishmentarianism
   blis
    ^bad character
```

The mismatch is an ‘l’ in the pattern, which does not match a ‘d’ in the text. Since there is no ‘d’ in the pattern at all, we move the pattern completely past the mismatch:

```
   vvvv match location
antidisestablishmentarianism
       blis
```

in fact, we move it further, to the first match on the first character of the pattern:

```
           vvvv match location
antidisestablishmentarianism
           blis
           ^ first character match
```

The case where we have a mismatch, but the character in the text does appear in the pattern is a little trickier: we find the next occurrence of the mismatched character in the pattern, and use that to determine the shift distance.

```
shoobeedoobeeboobah
edoobeeboob
       ^ mismatch
 ^ other occurrence of 'd'
```

Note that this can be a considerably smaller shift than in the previous case.

```
       v
shoobeedoobeeboobah
      edoobeeboob
       ^ match the bad character 'd'
      ^ new location
```

Exercise 56.4. Discuss how efficient you expect this heuristic to be in the context of genomics versus text searching. (See above.)

Good suffix rule

The ‘good suffix’ consists of the matched characters after the bad character. When moving the read, we try to keep the good suffix intact:

```
desistrust
  listrest
        ^^good suffix

desistrust
      listrest
        ^^next occurrence of suffix
```

Chapter 57

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.

57.1 Mathematical preliminaries

The matrix-matrix product $C \leftarrow A \cdot B$ is defined as

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

Straightforward code for this would be:

```
for (i=0; i<a.m; i++)
  for (j=0; j<b.n; j++)
    s = 0;
    for (k=0; k<a.n; k++)
      s += a[i,k] * b[k,j];
    c[i,j] = s;
```

However, this is not the only way to code this operation. The loops can be permuted, giving a total of six implementations.

Exercise 57.1. 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 2×2 block form:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{57.1}$$

(Convince yourself that this actually computes the same product $C = A \cdot B$.)

57.2 Matrix storage

The simplest way to store an $M \times N$ matrix is as an array of length MN. Inside this array we can decide to store the rows end-to-end, or the columns. 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 algbra 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 57.1.

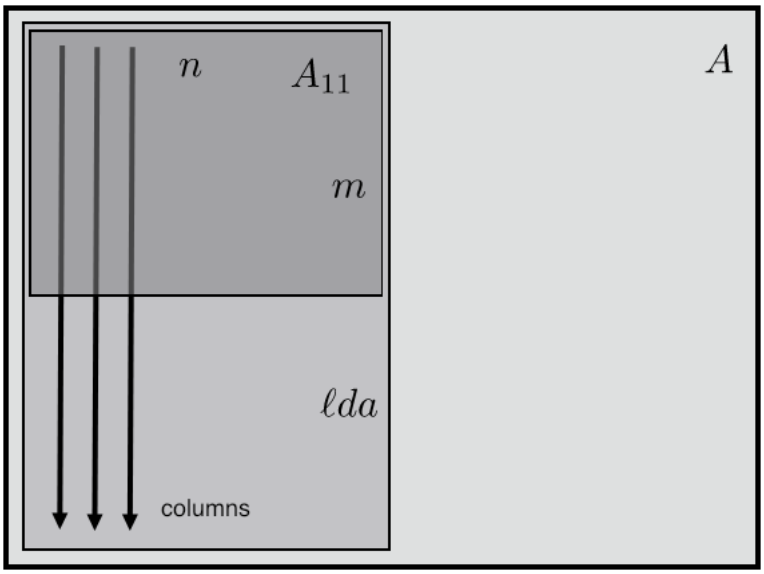


Figure 57.1: Submatrix out of a matrix, with m,n,lda of the submatrix indicated

Exercise 57.2. 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 11.8.5.

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 23.5.3), 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.

```
// example values for M,LDA,N
M = 2; LDA = M+2; N = 3;
```

```
// create a vector to contain the data
vector<double> one_data(LDA*N,1.);
// create a matrix using the vector data
Matrix one(M,LDA,N,one_data.data());
```

(If you have not previously programmed in C, you need to get used to the `double*` mechanism. Read up on section 11.8.2.)

Exercise 57.3. Start implementing the `Matrix` class with a constructor

```
Matrix::Matrix(int m,int lda,int n,double *data)
```

and private data members:

```
private:
  int m,n,lda;
  span<double> data;
```

Write a method

```
double& Matrix::at(int i,int j);
```

that you can use as a safe way of accessing elements.

Let's start with simple operations.

Exercise 57.4. 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 20) to introduce alternatives:

```
#ifdef DEBUG
  c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
  cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

where you access the data directly with

```
auto get_double_data() {
  double *adata;
  adata = data.data();
  return adata;
};
```

Exercise 57.5. Implement this. Use a cpp `#define` macro for the optimized indexing expression. (See section 20.1.2.)

57.2.1 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:

```
Matrix Left(int j);
Matrix Right(int j);
Matrix Top(int i);
Matrix Bot(int i);
```

where, for instance, `Left(5)` gives the columns with $j < 5$.

Exercise 57.6. Implement these methods and test them.

57.3 Multiplication

You can now write a first multiplication routine, for instance with a prototype

```
void Matrix::MatMult( Matrix& other,Matrix& out );
```

Alternatively, you could write

```
Matrix Matrix::MatMult( Matrix& other );
```

but we want to keep the amount of creation/destruction of objects to a minimum.

57.3.1 One level of blocking

Next, write

```
void Matrix::BlockedMatMult( Matrix& other,Matrix& out );
```

which uses the 2×2 form above.

57.3.2 Recursive blocking

The final step is to make the blocking recursive.

Exercise 57.7. Write a method

```
void RecursiveMatMult( Matrix& other,Matrix& out );
```

which

- Executes the 2×2 block product, using again `RecursiveMatMult` for the blocks.
- When the block is small enough, use the regular `MatMult` product.

57.4 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 57.8. 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 57.9. 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?

57.4.1 Parallelism (optional)

The four clauses of equation 57.1 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`.

57.4.2 Comparison (optional)

The final question is: how close are you getting to the best possible speed? Unfortunately you are still a way off. You can explore that as follows.

Your computer is likely to have an optimized implementation, accessible through:

```
#include <cblas.h>

cblas_dgemm
  ( CblasColMajor, CblasNoTrans, CblasNoTrans,
    m,other.n,n, alpha,adata,lda,
    bdata,other.lda,
    beta,cdata,out.lda);
```

which computes $C \leftarrow \alpha A \cdot B + \beta C$.

Exercise 57.10. 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.

Chapter 58

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 59

Cryptography

59.1 Basic ideas

https://simple.wikipedia.org/wiki/RSA_algorithm

https://simple.wikipedia.org/wiki/Exponentiation_by_squaring

Chapter 60

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 37): 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.

60.1 Reading the data

In the repository you find two text files

`GLB.Ts+dSST.txt` `GLB.Ts.txt`

that contain temperature deviations from the 1951–1980 average. Deviations are given for each month of each year 1880–2018. These data files and more can be found at https://data.giss.nasa.gov/gistemp/.

Exercise 60.1. Start by making a listing of the available years, and an array `monthly_deviation` of size $12 \times \text{nyears}$, where nyears is the number of full years in the file. Use formats and array notation.

The text files contain lines that do not concern you. Do you filter them out in your program, or are you using a shell script? Hint: a judicious use of `grep` will make the Fortran code much easier.

60.2 Statistical hypothesis

We assume that mr Shah was really saying that climate has a 'stationary distribution', meaning that highs and lows have a probability distribution that is independent of time. This means that in n data points, each point has a chance of $1/n$ to be a record high. Since over $n+1$ years each year has a chance of $1/(n+1)$, the $n+1$st year has a chance $1/(n+1)$ of being a record.

We conclude that, as a function of n, the chance of a record high (or low, but let's stick with highs) goes down as $1/n$, and that the gap between successive highs is approximately a linear function of the year[1].

This is something we can test.

Exercise 60.2. Make an array `previous_record` of the same shape as `monthly_deviation`. This array records (for each month, which, remember, we treat like independent measurements) whether that year was a record, or, if not, when the previous record occurred:

$$\mathrm{PrevRec}(m,y) = \begin{cases} y & \text{if } \mathrm{MonDev}(m,y) = \max_{m'}\big(\mathrm{MonDev}(m',y)\big) \\ y' & \text{if } \mathrm{MonDev}(m,y) < \mathrm{MonDev}(m,y') \\ & \text{and } \mathrm{MonDev}(m,y') = \max_{m''<m'}\big(\mathrm{MonDev}(m'',y)\big) \end{cases}$$

Again, use array notation. This is also a great place to use the `Where` clause.

Exercise 60.3. 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 60.4. Copy the code from `http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf` and adapt for our purposes: find the best fit for the slope and intercept for a linear function describing the gaps between records.

You'll find that the gaps are decidedly not linearly increasing. So is this negative result the end of the story, or can we do more?

1. Technically, we are dealing with a uniform distribution of temperatures, which makes the maxima and minima have a beta-distribution.

Exercise 60.5. 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?

PART VI

ADVANCED TOPICS

Chapter 61

Programming strategies

61.1 A philosophy of programming

Yes, your code will be executed by the computer, but:

- You need to be able to understand your code a month or year from now.
- Someone else may need to understand your code.
- ⇒ make your code readable, not just efficient

- Don't waste time on making your code efficient, until you know that that time will actually pay off.
- Knuth: 'premature optimization is the root of all evil'.
- ⇒ first make your code correct, then worry about efficiency

- Variables, functions, objects, form a new 'language':
  code in the language of the application.
- ⇒ your code should look like it talks about the application, not about memory.
- Levels of abstraction: implementation of a language should not be visible on the use level of that language.

61.2 Programming: top-down versus bottom up

The exercises in chapter 50 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 50.9.

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...

61.2.1 Worked out example

Take a look at exercise 6.10. We will solve this in steps.

1. State the problem:
```
// find the longest sequence
```

2. Refine by introducing a loop
```
// find the longest sequence:

// Try all starting points
// If it gives a longer sequence report
```

3. Introduce the actual loop:
```
// Try all starting points
for (int starting=2; starting<1000; starting++) {
// If it gives a longer sequence report
}
```

4. Record the length:
```
// Try all starting points
int maximum_length=-1;
for (int starting=2; starting<1000; starting++) {
  // If the sequence from 'start' gives a longer sequence report:
```

```
  int length=0;
  // compute the sequence from 'start'
  if (length>maximum_length) {
    // Report this sequence as the longest
  }
}
```

5. Refine computing the sequence:

```
// compute the sequence from 'start'
int current=starting;
while (current!=1) {
  // update current value
  length++;
}
```

6. Refine the update of the current value:

```
// update current value
if (current%2==0)
  current /= 2;
else
  current = 3*current+1;
```

61.3 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.

61.4 Documentation

Take a look at Doxygen.

61.5 Testing

If you write your program modularly, it is easy (or at least: easier) to test the components without having to wait for an all-or-nothing test of the whole program. In an extreme form of this you would write your code by *test-driven development*: for each part of the program you would first write the test that it would satisfy.

In a more moderate approach you would use *unit testing*: you write a test for each program bit, from the lowest to the highest level.

And always remember the old truism that 'by testing you can only prove the presence of errors, never the absence.

61.6 Best practices: C++ Core Guidelines

The C++ language is big, and some combinations of features are not advisable. Around 2015 a number of *Core Guidelines* were drawn up that will greatly increase code quality. Note that this is not about performance: the guidelines have basically no performance implications, but lead to better code.

For instance, the guidelines recommend to use default values as much as possible when dealing with multiple constructors:

```
class Point { // not this way
private:
  double d;
public:
  Point( double x,double y,double fudge ) {
    auto d = ( x*x + y*y ) * (1+fudge); };
  Point( double x,double y ) {
    auto d = ( x*x + y*y ); };
};
```

This is bad because of code duplication. Slightly better:

```
class Point { // not this way
private:
  double d;
public:
  Point( double x,double y,double fudge ) {
    auto d = ( x*x + y*y ) * (1+fudge); };
  Point( double x,double y ) : Point(x,y,0.) {};
};
```

which wastes a couple of cycles if fudge is zero. Best:

```
class Point { // not this way
private:
  double d;
public:
  Point( double x,double y,double fudge=0. ) {
    auto d = ( x*x + y*y ) * (1+fudge); };
};
```

Chapter 62

Tiniest of introductions to algorithms and data structures

62.1 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.

62.1.1 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 elememt if the new last element is removed.

The actions of adding and removing the last element are known as *push* and *pop* respectively.

Exercise 62.1. Write a class that implements a stack of integers. It should have methods

```
void push(int value);
int pop();
```

62.1.2 Linked lists

Before doing this section, make sure you study section 15.

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 11.3.2.)

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

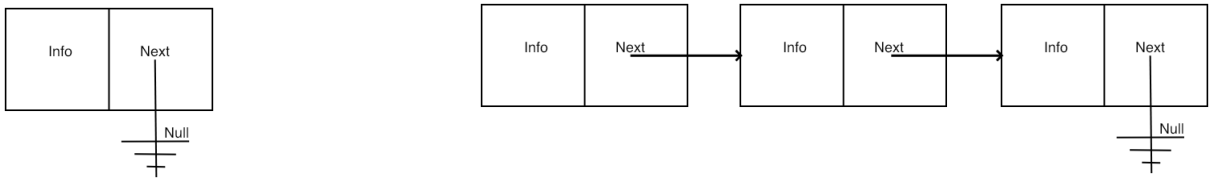


Figure 62.1: Node data structure and linked list of nodes

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.

We illustrate this in figure 62.1.

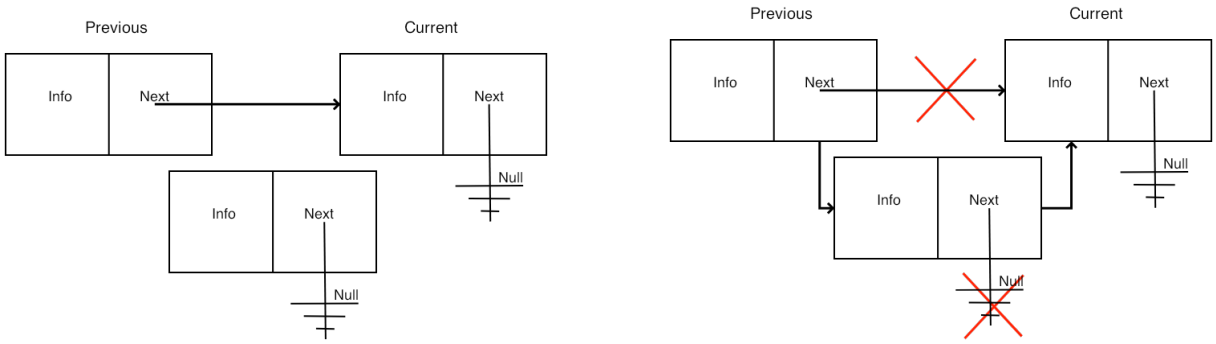


Figure 62.2: Insertion in a linked list

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 62.2.

62.1.2.1 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.

A linked list has as its only member a pointer to a node:

```
class List {
private:
  unique_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

A node has information fields, and a link to another node:

```
class Node {
  friend class List;
private:
  int datavalue{0},datacount{0};
  unique_ptr<Node> next{nullptr};
public:
  friend class List;
  Node() {}
  Node(int value,unique_ptr<Node> tail=nullptr)
    : datavalue(value),datacount(1),next(move(tail)) {};
  ~Node() { cout << "deleting node " << datavalue << endl; };
```

A Null pointer indicates the tail of the list.

62.1.2.2 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.

```
int recursive_length() {
  if (head==nullptr)
    return 0;
  else
    return head->listlength();
};
```

```
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.

Use a *bare pointer*, which is appropriate here because it doesn't own the node.

```
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.)

Exercise 62.2. Write a function

```
bool List::contains_value(int v);
```

to test whether a value is present in the list.

Try both recursive and iterative.

62.1.2.3 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.

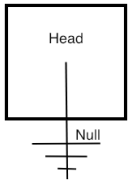
Our choice of using unique pointers dictates a certain design.

We will write functions

```
void List::insert(int value);
void Node::insert(int value);
```

that add the value to the list. The `List::insert` value can put a new node in front of the first one; the `Node::insert` assumes the the value is on the current node, or gets inserted after it.

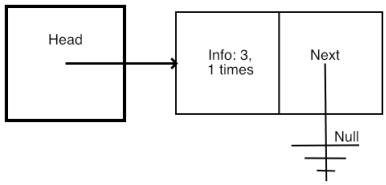
There are a lot of cases here. You can try this by an approach called Test-Driven Development (TDD): first you decide on a test, then you write the code that covers that case.



Step 1: dealing with an empty list

Exercise 62.3. Write a `List::length` method, so that this code gives the right output:

```
List mylist;
cout << "Empty list has length: "
     << mylist.listlength_iterative() << endl;
cout << endl;
```

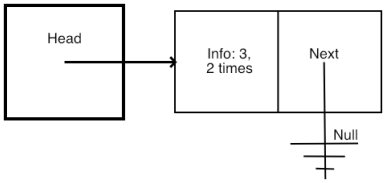


Step 2: insert the first element

Exercise 62.4. 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.

```
mylist.insert(3);
```

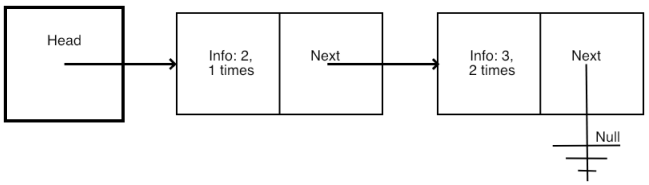
```
cout << "After one insertion the length is: "
     << mylist.listlength_iterative() << endl;
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << endl;
else
  cout << "Hm. Should contain 3" << endl;
if (mylist.contains_value(4))
  cout << "Hm. Should not contain 4" << endl;
else
  cout << "Indeed: does not contain 4" << endl;
cout << endl;
```



Step 3: inserting an element that already exists

Exercise 62.5. 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:

```
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;
```



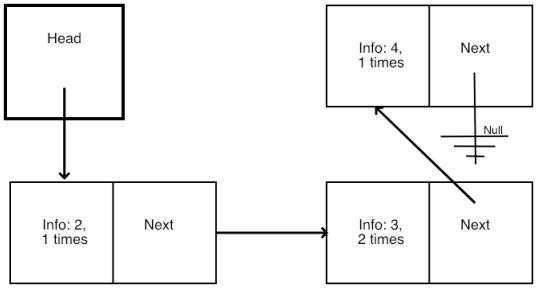
Step 4: inserting an element before another

Exercise 62.6. One of the remaining cases is inserting an element that goes at the head. Update your `insert` method to get this to work:

```
mylist.insert(2);
cout << "Inserting 2 goes at the head; now the length is: "
     << mylist.listlength_iterative() << endl;
if (mylist.contains_value(2))
  cout << "Indeed: contains 2" << endl;
```

```
else
  cout << "Hm. Should contain 2" << endl;
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << endl;
else
  cout << "Hm. Should contain 3" << endl;
cout << endl;
```

Step 5: inserting an element at the end



Exercise 62.7. Finally, if an item goes at the end of the list:

```
mylist.insert(4);
cout << "Inserting 4 goes at the tail; now the length is: "
     << mylist.listlength_iterative() << endl;
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;
cout << endl;
```

62.1.3 Trees

Before doing this section, make sure you study section 15.

A tree can be defined recursively:

- A tree is empty, or
- a tree is a node with some number of children trees.

Let's design a tree that stores and counts integers: each node has a label, namely an integer, and a count value that records how often we have seen that integer.

Our basic data structure is the node, and we define it recursively to have up to two children. This is a problem: you can not write

```
class Node {
private:
  Node left,right;
```

```
    }
```

because that would recursively need infinite memory. So instead we use pointers.

```
class Node {
private:
  int key{0},count{0};
  shared_ptr<Node> left,right;
  bool hasleft{false},hasright{false};
public:
  Node() {}
  Node(int i,int init=1 ) { key = i; count = 1; };
  void addleft( int value) {
    left = make_shared<Node>(value);
    hasleft = true;
  };
  void addright( int value ) {
    right = make_shared<Node>(value);
    hasright = true;
  };
  /* ... */
};
```

and we record that we have seen the integer zero zero times.

Algorithms on a tree are typically recursive. For instance, the total number of nodes is computed from the root. At any given node, the number of nodes of that attached subtree is one plus the number of nodes of the left and right subtrees.

```
int number_of_nodes() {
  int count = 1;
  if (hasleft)
    count += left->number_of_nodes();
  if (hasright)
    count += right->number_of_nodes();
  return count;
};
```

Likewise, the depth of a tree is computed as a recursive max over the left and right subtrees:

```
int depth() {
  int d = 1, dl=0,dr=0;
  if (hasleft)
    dl = left->depth();
  if (hasright)
    dr = right->depth();
  d = max(d+dl,d+dr);
  return d;
};
```

Now we need to consider how actually to insert nodes. We write a function that inserts an item at a node. If the key of that node is the item, we increase the value of the counter. Otherwise we determine whether to add the item in the left or right subtree. If no such subtree exists, we create it; otherwise we descend in the appropriate subtree, and do a recursive insert call.

```
void insert(int value) {
  if (key==value)
    count ++;
  else if (value<key) {
    if (hasleft)
      left->insert(value);
    else
      addleft(value);
  } else if (value>key) {
    if (hasright)
      right->insert(value);
    else
      addright(value);
  } else throw(1); // should not happen
};
```

62.2 Algorithms

This *really* **really** goes beyond this book.

- Simple ones: numerical
- Connected to a data structure: search

62.2.1 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.

62.2.1.1 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.

```
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 64.1.1) 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.

62.2.1.2 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 62.8. 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?

62.3 Programming techniques

62.3.1 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.

```
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];
}
```

Chapter 63

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.

63.1 Loops as quantors

Quite often, algorithms can be expressed mathematically. In that case you should make your program look like the mathematics.

63.1.1 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.

$$\neg \, \text{divides}(2, n) \cap \ldots \cap \neg \, \text{divides}(n-1, n)$$

And this sequence of 'and' conjunctions can be programmed:

```
for (int f=2; f<n; f++)
  isprime = isprime && not divides(f,p)
```

Now our only worry is how to initialize `isprime`. The initial value corresponds to an 'and' conjunction over an empty set, which is true, so:

```
bool isprime{true};
for (int f=2; f<n; f++)
  isprime = isprime && not divides(f,p)
```

63.1.2 Thereis-quantor

What if we had expressed primeness as:

$$\text{isprime}(n) \equiv \neg\exists_{2\leq f<n}\colon \text{divides}(f,n)$$

To get a pure quantor, and not a negated one, we write:

$$\text{isnotprime}(n) \equiv \exists_{2\leq f<n}\colon \text{divides}(f,n)$$

Spelling out the exists-quantor as

$$\text{isnotprime}(n) \equiv \text{divides}(2,n) \cup \ldots \cup \text{divides}(n-1,n)$$

we see that we need a loop where we test if any iteration satisifies a predicate. That is, we do an 'or'-reduction on the results of each iteration.

```
for (int f=2; f<n; f++)
  isnotprime = isnotprime or divides(f,p)
bool isprime = not isnotprime;
```

Again we take care to initialize the reduction variable correctly: applying $\exists_{s\in S}P(s)$ over an empty set S is `false`: bool isnotprimefalse;

```
for (int f=2; f<n; f++)
  isnotprime = isnotprime or divides(f,p)
bool isprime = not isnotprime;
```

63.2 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

$$P(\ell_k, m_k, k) = \left[\begin{array}{ll} & \ell_k < k \\ \wedge & c(\ell_k) = m_k \text{(only if } k > 0) \\ \wedge & \forall_{\ell<k}\colon c(\ell) \leq m_k \end{array}\right.$$

for $k = N$.

We develop the code that makes this predicate inductively true. We start out with

$$\ell_0 = -1, \quad m_0 = 0 \Rightarrow P(\ell_0, m_0, 0).$$

The inductive proof corresponds to a loop:

- we assume that at the start of the k-th iteration $P(\ell_k, m_k, k)$ is true;
- the iteration body is such that at the end of the k-th iteration $P(\ell_{k+1}, m_{k+1}, k+1)$ is true;
- this of course sets up the predicate at the start of the next iteration.

The loop structure is then:

```
k=0;
{P(l_k, m_k, k)}
while ( k<N ) {
  {P(l_k, m_k, k)}
  update;
  {P(l_{k+1}, m_{k+1}, k+1)}
  k = k+1;
}
```

The update has to extend the predicate from k to $k+1$. Let us consider the parts of it.

We need to establish

$$\forall_{\ell<k+1}\colon c(\ell) \leq m_{k+1}$$

We split the range $\ell < k+1$ into $\ell < k$ and $\ell = k$:

- the first part

  $$\forall_{\ell<k}\colon c(\ell) \leq m_{k+1}$$

  is true if $m_{k+1} \geq m_k$;
- the part

  $$\ell = k\colon c(\ell) \leq m_{k+1}$$

  states that $m_{k+1} \geq c(k)$.

Together we get that

$$m_{k+1} \geq \max(m_k, c(k))$$

Finally, the clause

$$c(\ell_{k+1}) = m_{k+1}$$

can be satisfied:

- If $c(k) > m_k$, we need to set $m_{k+1} = c(k)$ and $\ell_{k+1} = k$.
- (Stricly speaking, there is a possibility $m_{k+1} > c(k)$. This is not possible, because we can not satisfy $m_{k+1} = c(\ell_k)$ for any k.)
- If $c(k) \leq m_k$, we need to set $m_{k+1} \geq m_k$. Again, $m_{k+1} > m_k$ can not be satisfied by any ℓ_{k+1}, so we conclude $m_{k+1} = m_k$.

Chapter 64

Complexity

64.1 Order of complexity

64.1.1 Time complexity

Exercise 64.1. 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 64.2. How many operations, as a function of n, are performed in these two solutions?

64.1.2 Space complexity

Exercise 64.3. 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 64.4. How much space do the two solutions require?

PART VII

INDEX AND SUCH

Chapter 65

General index

Index

`#ifndef`, 195
`#define`, 193, 391
`#ifdef`, 195
`#ifndef`, 195
`#once`, 195
`#pack`, 195
`#typedef`, 194

abs, 47
abstraction, 69
Amazon
    delivery truck, 377
    prime, 377, 383
Amazon Prime, 31
Apple, 27
argument
    actual, 272
    default, 80
    dummy, 272
array, **117**
    associative, 208
    assumed-shape, 294
    automatic, 289, 294
    bounds checking, 118
    index, 118
    initialization, 119, 290
    operations, semantics of, 291
    rank, 292
    static, 289
    subscript, 118
ASCII, 321
assignment, **40**
asterisk
    in Fortran formatted I/O, 276

Boost, 141
bottom-up, 405
Boyer-Moore, 387
bubble sort, 134, 416
bug, 20
bus error, 118

C
    parameter passing, 171–174
    pointer, 167–174, 242
    preprocessor, 251
    string, 141, 242
C preprocessor, *see* preprocessor
C++, 235
    C++03, 236
    C++11, 182, 227, **236–237**
    C++14, 90, 182, **237**
    C++17, 37, 48, 56, 141, 181, 212, 214, **237**
    C++20, 133, 182, 188, 198, 204, 219, **237–238**
    Core Guidelines, **408**
    standard, 133
Caesar cypher, 139
calendars, 238
call-back, 233
calling environment, 151
capture, 228
case sensitive, 39
cast, 46, **230**
    lexical, 141
charconv, **141**
class, 95
    abstract, 107
    base, 105
    derived, 105
    iteratable, 225
closure, 85
code
    duplication, 71
    maintainance, 407

code reuse, 71
Collatz conjecture, 67
column-major, **292**, 310
compilation
    separate, 184, 282, 363
compile-time constant, 289
compiler, 28, 35
    and preprocessor, 193
    one pass, 73
compiling, 28
complex numbers, 41, **207**
concept, 199
concepts, 237
conditional, 51, 323
connected components, *see* graph, connected
const
    reference, 175
constructor, 96, 165, 243
    copy, 112, 176
    default, 101
    range, 241
container, **207**
contains
    for class functions, 285
    in modules, 282
continuation character, 253
coroutines, 237
covid-19, 363

data model, 42
datatype, 39, 320
debugger, 205
`define`, *see* `#pragma define`, *see* `#pragma define`
definition vs use, 83
dependence, 291
dereference, 168
dereferencing, 130
`dereferencing a`
    nullptr, 163
destructor, 87, **113**
    at end of scope, 86
    in Fortran, *see* final procedure
do
    concurrent, 66
do concurrent, 299
do loop
    implicit, 309, 310
        and array initialization, 290
    implied, 262
dynamic
    programming, 374

ebola, 363
ECMA, 212
efficiency gap, 374
Eigen, 128
emacs, **27**, 27, 37, 252
encoding
    extendible, 141
epsilon, **217**
error
    compile-time, 38
    run-time, 38
    syntax, 38
exception, **202**
    catch, 203
    catching, 202
    throwing, 202
executable, 28, 36, 282
exponent part, **42**
expression, 40
extent
    of array dimension, 293

Fasta, 385
Fastq, 386
file
    binary, 35, 38
    executable, 184
    handle, 113
    include, 95
    object, 184, 282
    source, 35
final, **286**
floating point error, 261
floating point number, **42**
flush, 148
fmtlib, 241
for
    indexed, 120
    range-based, 120
Fortran

90, 251
case ignores, 252
comments, 253
forward declaration, 183
of classes, 85
of functions, 85
function, 69, **269**, 269
argument, 73
arguments, 72
body, **72**
call, 69, 71
defines scope, 73
definition, 69
header, 73
parameter, 73
parameters, 72
pointwise application to array, 334
prototype, 73
result type, **72**
function try block, 204
functional programming, **74**, 151
functor, **111**

gerrymandering, 369
glyph, **141**
GNU, 35
Goldbach conjecture, 348, 349
Google, 365
graph
connected, 367
diameter, 367
greedy search, *see* search, greedy

has-a relation, 103
hdf5, 309
header, 363
header file, 184, 193
and global variables, 187
treatment by preprocessor, 187
vs modules, 237
header-only, **186**
heap, **132**, 133
fragmentation, **132**
hexadecimal, 167
Holmes
Sherlock, 139
homebrew, 27

host association, **266**

I/O
formatted, 309
list-directed, 309
unformatted, 309
if
ternary, 55, 325
`ifdef`, *see* `#pragma ifdef`
`ifndef`, *see* `#pragma ifndef`
incubation period, 363
index
section, 290
inheritance, **105**
initialization
variable, 39
initializer
in conditional, 56
initializer list, 98, 119, 120
inline, 272
is-a relation, 105
iteration
of a loop, *see* loop, iteration
iterator, 129, **208**, 236

julia
compared to pytyhon, 320

keywords, 38

label, 277
lambda, *see* closure, 233
expression, 227
generic, 229
make mutable, 230
lamda
const by default, 230
lazy evaluation, 219
lazy execution, 219
lexicographic ordering, 62
linear regression, 400
linker, 184, 282
Linux, 27
list
linked, **409–414**
in Fortran, 305–308
single-linked, 220

locality, 291
logging, 148
loop, **59**
  body, 59
  counter, 59
  for, 59
  header, 59
  inner, 62, 329
  iteration, 59
  nest, 62, 329
  outer, 62, 329
  variable, 60
  while, 59
lowest, **217**
lvalue, **234**

macports, 27
Make, 184
makefile, 185, 363
Manhattan distance, 377
mantissa, **42**
max, 47
member
  initializer list, **96**, 96, 126
  of struct, 89
members, *see* object, members
memoization, 374, **417**
memory
  bottleneck, 393
  leak, 113, 133, **172**, 172, 295
memory leak, 160
memory leaking, **174**
method, **99**
  abstract, 107
  overriding, 106
methods, *see* object, methods
Microsoft
  Windows, 27
  Word, 20
module, **188**
modules, 237
move semantics, 235
multiple dispatch, **331**, 338

namespace, **189**
Newton's method, 76
NP-hard, 381

NULL, 163
null terminator, 242

object
  members, 95
  methods, 95
  state of, 100
object file, *see* file, object
`once`, *see* `#pragma once`
OpenFrameworks, 236
operator
  bitwise, 53
  comparison, 53
  logic, 53
  overloading, **110**
    and copies, 235
    of parentheses, 111
  precedence, 53
  short-circuit, 55, 324
  shortcut, 45, 321
  spaceship, 237
opt2, 381
output
  binary, 311
  raw, 311
  unformatted, 311

`pack`, *see* `#pragma pack`
package manager, 27
Pagerank, 365
parameter, *see also* function, parameter
  formal, 273
  input, **77**
  output, **77**, 212
  pass by value, **75**
  passing, 74
    by reference, 151, 164, 243
    by value, 151
  passing by reference, 74, **77**, 91
    in C, 77
  passing by value, 74, 91
  throughput, **77**
Pascal's triangle, 134
pass by reference, *see* parameter, passing by reference
pass by value, *see* parameter, passing by value
path

Hamiltonian, 386
PETSc, 205
pipe, 219
pointer, 114
and heap, 132
arithmetic, 130, 170
bare, 162, 243, 411
dereference, 209
dereferencing, 303
null, 163, 410
opaque
in C++, 164
smart, 133, 157
unique, 162
void, 233
weak, 160, **163**
Poisson
distribution, 219
polymorphism, 110, 331
pop, 409
preprocessor
and header files, 187
conditionals, 194–195
macro
parametrized, 194
macros, 193–194
procedure, **267**
final, 286
internal, 266
procedures
internal, 272
module, 272
program
statements, 36
programming
dynamic, 372
parallel, 66
prototype, **183**
punch cards, 251
push, 409
putty, 27
python, 25

quicksort, 417

radix point, **42**
RAII, **242**
random
seed, 314
random number
generator, 81, 218
Fortran, 314
seed, 81
range-based for loop, *see* for, range-based
ranges, 238
recurrence, 300
recursion, *see* function, recursive
depth, 81
mutual, **79**
reduction, 210
reference, **76**, 151
argument, 164, 243
const, 124, 151, 155
to class member, 152
to class member, 152
reference count, 162
regular expression, 211
reserved words, 39
return, **72**
code, 38
makes copy, 155
statement, 38
root finding, 67
roundoff error analysis, 48
runtime error, 201
rvalue, **234**
reference, 235

scope, 72, **83**
and stack, 132
dynamic, 86
in conditional branches, 54
lexical, 83, 86
of function body, 73
search
greedy, **380**, 381
segmentation fault, 118
shape
of array, 293
shell
inspect return code, 38
side-effects, 177
significand, **42**

Single Source Shortest Path, 367
SIR model, 362
smartphone, 20
smatch, 211
source
  format
    fixed, 251
    free, 251
source code, 28
stack, 81, 87, **132**, 133, 295, **409**
  overflow, 81, 133, 295
Standard Template Library, 207
statement functions, 272
stream, 147
string, 137
  concatenation, 138
  null-terminated, 141
  raw literal, **138**, 212
  size, 138
  stream, 140
struct
  denotation, 92
structured binding, 213
subprogram, *see* function
syntax
  error, 201

templates
  and separate compilation, 186
Terminal, 27
test-driven development, 407
testing, 407
text
  formatting, 238
time zones, 238
timer
  resolution, 314
top-down, 405
tuple, **212**
  denotation, 213
type
  derived, 279
  nullable, 214
  signature
    ., 331
`typedef`, *see* `#pragma typedef`

Unicode, **141**, 320, 321
unit, 310
unit testing, 408
Unix, 27
UTF8, **141**

values
  boolean, 43
variable, 39, 320
  assignment, 39
  declaration, **39**, 39
  global, 187
    in header file, 187
  initialization, 44
  lifetime, 84
  numerical, 41
  shadowing, 84
  static, 86, 265
vector, 189, 350
  methods, 122
  subvector, 210
vi, 27
Virtualbox, 27
Visual Studio, 27
VMware, 27

X windows, 27
Xcode, 27

Chapter 66

Index of C++ keywords

Index

__FILE__, 204
__LINE__, 204
__cpluscplus, 236

algorithm, 47, 111, **216**, 229
all_of, **216**
any, **215**
any_cast, **215**
any_of, **216**, 216, 227, 229
array, **133**
assert, **202**
auto, 120, 236
auto_ptr, 236, 237

bad_alloc, 204
bad_exception, 204
basic_ios, 149
begin, **209**, 225
break, 63

cerr, **148**, 241
char, **137**
cin, 41, 44, 241
clang++, 36
cmath, 45, 47, 228
const, 175, 177, 181
const_cast, 181, 233
constexpr, **181**, 181, 237
continue, 64
cout, 44, 241

decltype, **225**
delete, 237
distance, **210**
dynamic_cast, 231

end, **209**, 225
endl, 148
EOF, **149**

eof, 149
erase, 209
errno, 205
exception, 204

false, 43, 45
filter, **219**
for_each, **216**, 216
friend, 108
from_chars, 141
function, 228

g++, 35, 36
gdb, 205
get, 161
get_if, **214**
getline, 148, 149

has_value, 214

icpc, 35, 36
if, 56
Inf, 48, 217
insert, 209
int, **42**
intptr_t, **233**
is_eof, 149
is_open, 149
isinf, 217
isnan, 204, 217
iterator, 209
itoa, 140

limits, 216
limits.h, 216
long, **42**, 218
long long, **42**, 218
longjmp, 205

malloc, 133, 165, **170**, 172, 174, 241–243
map, **208**
max_element, **210**
MAX_INT, 216
monostate, **215**
mutable, **181**, **230**

NaN, 48, 217
NDEBUG, **202**
new, 127, 160, 162, 172–174, 237
noexcept, 204
none_of, **216**
NULL, 141
nullopt, 214
nullptr, 163, 165, 231, 410
nullptr_t, 163
numeric_limits, 216

open, 149
optional, 214
override, 107

printf, 241
private, 96, 99, 101, 102
protected, **105**
public, 96, 101, 102
push_back, 123

rand, 81
RAND_MAX, 81
range, **219**
ranges, 219
rbegin, 224
regex, 211
regex_match, **211**
regex_search, **211**
reinterpret_cast, 230, **233**
rend, 224

scanf, 241
setjmp, 205
shared_ptr, 243, 410
short, **42**
size_t, 117, 231
sizeof, **170**, 172
source_location, 204
span, **133**, 133, 238, 241, 390
sprintf, 140
srand, 81
static_cast, 230, 232
std::any, 164
string, 242
stringstream, 140
struct, **89**, 212
swap, 235
switch, 54, 56

this, 114, 162
to_chars, 141
to_string, 140
to_vector, **219**
transform, **219**
true, 43, 45

union, 214
unique_ptr, 162, 243, 410
using, 38, 194

valgrind, 205
value, 214
value_or, **214**
variant, **214**
vector, 117, **122**, 133, 210, 219, 241
view, **219**
void, **73**, 73

weak_ptr, **163**
while, **65**

Chapter 67

Index of Fortran keywords

Index

.AND., 260
.and., 259
.eq., 259
.false., 260
.ge., 259
.gt., 259
.le., 259
.lt., 259
.ne., 259
.or., 259
.true., 260

Abs, 296
advance, 307
AIMIG, 256
all, 297, 298
allocatable, 254
allocate, 295
any, 298
Associated, 305

Begin, 252
bit_size, 255

c_sizeof, 255
call, 268, 270
case, 260
CHAR, 275
Character, 254
class, 285
Close, 309, 310
CMPLX, 256
Common, 266
Complex, 254
concurrent, 299
CONJG, 256
Contains, 266, 267
contains, 270, 272, 282, 285, 286, 313
Cshift, 296

data, 310
DBLE, 256
deallocate, 295
DIM, 296
dimension, 254, 289
dimension(:), 294
do, 261, 262, 265
DOT_PRODUCT, 297
Dot_Product, 296

end, 252
end do, 262, 265
End Program, 252
entry, 272
exit, 262
external, 313

F90, 251
FLOAT, 256
for all, 299
forall, 298
Format, 309
format, 310
Function, 268, 295
function, 269

gfortran, 252

ICHAR, 275
if, 259
ifort, 252
implicit none, 281, 282
in, 272
INDEX, 275
inout, 272
INT, 256
Integer, 254

integer, 254
intent, 254
Interface, 267
interface, 270, **313**, 313
iso_c_binding, 255

kind, **255**

Lbound, **293**
LEN, 275
len, 275
Logical, 254, 260
logical, 254

MASK, 296
MATMUL, 297
MatMul, 296
MaxLoc, 296
MaxVal, 296
MinLoc, 296
MinVal, 296
Module, 266, 267, 286
module, 280

NINT, 256
Nullify, 305

Open, 309
open, 310
Optional, 273
out, 272

parameter, 254
parameters, 254
pointer, 303
Present, 273
Print, 309
print, 309
private, 283
procedure, 286
Product, 296
Program, 252, 267
protected, 283
public, 283

random_number, 314
random_seed, 314
Read, 309
REAL, 256
Real, 254
real(4), 254
real(8), 254, 255
Recursive, 268
RESHAPE, 294
Result, 270, 295
result, 269
return, 268, 269

Save, 266, 273
Select, 260
select, 260
selected_int_kind, **254**
selected_real_kind, **254**
size, **293**
SNGL, 256
SPREAD, 294
stat=ierror, 295
stop, 252
storage_size, 255
Subroutine, 295
subroutine, 269
Sum, 296
system_clock, 314

target, 303
Transpose, 296
TRIM, 275
trim, 275
Type, 286
type, 279, 285

Ubound, **293**
use, 270, 281, 282

where, 297
while, 262
Write, 256, 307, 309
write, 310

Chapter 68

Index of Julia keywords

Index

AbstractMatrix, 334

collect, 334

Chapter 69

remaining index

Index

XQuartz, 27

Chapter 70

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11. IEEE Computer Society Press, Los Alamitos, California, 1990.

[2] Roy M. Anderson and Robert M. May. Population biology of infectious diseases: Part I. *Nature*, 280:361–367, 1979. doi:10.1038/280361a0.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc webpage. http://www.mcs.anl.gov/petsc, 2011.

[5] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006.

[6] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762?772, October 1977.

[7] Yen-Lin Chen, Chuan-Yen Chiang, Yo-Ping Huang, and Shyan-Ming Yuan. A project-based curriculum for teaching c++ object-oriented programming. In *Proceedings of the 2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*, UIC-ATC '12, pages 667–672, Washington, DC, USA, 2012. IEEE Computer Society.

[8] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling ?17, page 20?29, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[11] Juan M. Restrepo, , and Michael E. Mann. This is how c̈limate is always changing.̈ *APS Physics, GPS newletter*, February 2018.

[12] Lin S. and Kernighan B. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.