

Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

August 13, 2019

Contents

I	Introduction	15
1	Introduction	17
1.1	<i>Programming and computational thinking</i>	17
1.1.1	History	17
1.1.2	Is programming science, art, or craft?	20
1.1.3	Computational thinking	20
1.1.4	Hardware	23
1.1.5	Algorithms	23
1.2	<i>About the choice of language</i>	24
1.3	<i>Further reading</i>	25
2	Warming up	27
2.1	<i>Programming environment</i>	27
2.1.1	Language support in your editor	27
2.2	<i>Compiling</i>	28
2.3	<i>Your environment</i>	28
3	Teachers guide	29
3.1	<i>Justification</i>	29
3.2	<i>Timeline for a C++/F03 course</i>	29
3.2.1	Project-based teaching	30
3.2.2	Choice: Fortran or advanced topics	31
II	C++	33
4	Basic elements of C++	35
4.1	<i>From the ground up: Compiling C++</i>	35
4.1.1	A quick word about unix commands	36
4.1.2	C++ is a moving target	37
4.2	<i>Statements</i>	37
4.3	<i>Variables</i>	38
4.3.1	Variable declarations	39
4.3.2	Assignments	40
4.3.3	Datatypes	41
4.3.4	Initialization	43
4.4	<i>Input/Output, or I/O as we say</i>	43
4.5	<i>Expressions</i>	44

4.5.1	Numerical expressions	44
4.5.2	Truth values	44
4.5.3	Type conversions	45
4.6	<i>Advanced topics</i>	46
4.6.1	Library functions	46
4.6.2	Number values and undefined values	47
4.6.3	Constants	47
4.7	<i>Review questions</i>	48
4.8	<i>Sources used in this chapter</i>	48
5	Conditionals	49
5.1	<i>Conditionals</i>	49
5.2	<i>Operators</i>	50
5.3	<i>Switch statement</i>	51
5.4	<i>Scopes</i>	52
5.5	<i>Sources used in this chapter</i>	52
6	Looping	55
6.1	<i>The ‘for’ loop</i>	55
6.2	<i>Looping until</i>	59
6.3	<i>Advanced topics</i>	61
6.3.1	Parallelism	61
6.4	<i>Exercises</i>	62
6.5	<i>Sources used in this chapter</i>	62
7	Functions	65
7.1	<i>Function definition and call</i>	66
7.1.1	Another option for defining functions	67
7.2	<i>Why use functions?</i>	68
7.3	<i>Anatomy of a function definition and call</i>	69
7.3.1	Examples	69
7.4	<i>Void functions</i>	70
7.5	<i>Parameter passing</i>	71
7.5.1	Pass by value	71
7.5.2	Pass by reference	72
7.6	<i>Recursive functions</i>	75
7.6.1	Stack overflow	76
7.7	<i>Advanced function topics</i>	77
7.7.1	Default arguments	77
7.7.2	Polymorphic functions	77
7.8	<i>Library functions</i>	77
7.8.1	Random function	77
7.9	<i>Review questions</i>	78
7.10	<i>Sources used in this chapter</i>	78
8	Scope	83
8.1	<i>Scope rules</i>	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
8.5	Sources used in this chapter	88
9	Structures	91
9.1	Why structures?	91
9.2	The basics of structures	91
9.3	Sources used in this chapter	94
10	Classes and objects	99
10.1	What is an object?	99
10.1.1	Constructor	100
10.1.2	Methods	100
10.1.3	Initialization	101
10.1.4	Methods	102
10.1.5	Default constructor	103
10.1.6	Accessors	104
10.1.7	Examples	105
10.2	Inclusion relations between classes	105
10.2.1	Accessors and other methods	106
10.3	Inheritance	107
10.3.1	Methods of base and derived classes	107
10.3.2	Virtual methods	108
10.3.3	Advanced topics in inheritance	109
10.4	Advanced topics	109
10.4.1	Returning by reference	109
10.4.2	Accessor functions	110
10.4.3	Accessability	111
10.4.4	Polymorphism	111
10.4.5	Operator overloading	111
10.4.6	Functors	112
10.4.7	Copy constructor	112
10.4.8	Destructor	114
10.4.9	'this' pointer	115
10.4.10	Static members	115
10.5	Review question	116
10.6	Sources used in this chapter	116
11	Arrays	127
11.1	Introduction	127
11.1.1	Vector creation	127
11.1.2	Element access	128
11.1.3	Initialization	128
11.2	Ranging over an array	129
11.3	Vector are a class	131
11.3.1	Vector methods	132
11.3.2	Vectors are dynamic	132

11.4	<i>Vectors and functions</i>	133
11.4.1	Pass vector to function	133
11.4.2	Vector as function return	134
11.5	<i>Vectors in classes</i>	135
11.5.1	Timing	136
11.6	<i>Wrapping a vector in an object</i>	136
11.7	<i>Multi-dimensional cases</i>	137
11.7.1	Matrix as vector of vectors	137
11.7.2	A better matrix class	138
11.8	<i>Advanced topics</i>	138
11.8.1	Iterators	138
11.8.2	Old-style arrays	139
11.8.3	Stack and heap allocation	142
11.8.4	The Array class	142
11.8.5	Span	142
11.9	<i>Exercises</i>	143
11.10	<i>Sources used in this chapter</i>	144
12	Strings	155
12.1	Characters	155
12.2	<i>Basic string stuff</i>	155
12.3	Conversion	158
12.4	C strings	158
12.5	<i>Sources used in this chapter</i>	158
13	Input/output	163
13.1	<i>Formatted output</i>	163
13.1.1	Floating point output	165
13.1.2	Saving and restoring settings	166
13.2	<i>File output</i>	167
13.2.1	Binary output	167
13.3	<i>Output your own classes</i>	167
13.4	<i>Output buffering</i>	168
13.5	<i>Input</i>	168
13.5.1	File input	169
13.5.2	Input streams	170
13.6	<i>Sources used in this chapter</i>	170
14	References	179
14.1	<i>Reference</i>	179
14.2	<i>Pass by reference</i>	179
14.3	<i>Reference to class members</i>	180
14.4	<i>Reference to array members</i>	182
14.5	rvalue references	183
14.6	<i>Sources used in this chapter</i>	183
15	Pointers	187
15.1	<i>The ‘arrow’ notation</i>	187
15.2	<i>Making a shared pointer</i>	188
15.2.1	Pointers and arrays	188

15.2.2	Smart pointers versus address pointers	190
15.3	<i>Garbage collection</i>	190
15.4	<i>More about pointers</i>	191
15.4.1	Get the pointed data	191
15.4.2	Example: linked lists	192
15.5	<i>Advanced topics</i>	192
15.5.1	Unique pointers	192
15.5.2	Base and derived pointers	192
15.5.3	Shared pointer to ‘this’	193
15.5.4	Null pointer	193
15.5.5	Void pointer	193
15.5.6	Pointers to non-objects	193
15.6	<i>Sources used in this chapter</i>	194
16	C-style pointers and arrays	201
16.1	<i>What is a pointer</i>	201
16.2	<i>Pointers and addresses, C style</i>	201
16.3	Arrays and pointers	203
16.4	Pointer arithmetic	204
16.5	Multi-dimensional arrays	205
16.6	Parameter passing	205
16.6.1	Allocation	206
16.6.2	Use of new	208
16.7	Memory leaks	208
16.8	<i>Sources used in this chapter</i>	209
17	Const	213
17.1	<i>Const arguments</i>	213
17.2	<i>Const references</i>	213
17.3	<i>Const methods</i>	215
17.4	<i>Const and pointers</i>	215
17.5	<i>Sources used in this chapter</i>	216
18	Prototypes	223
18.1	<i>Prototypes for functions</i>	223
18.1.1	Separate compilation	224
18.1.2	Header files	225
18.1.3	C and C++ headers	225
18.2	<i>Prototypes for class methods</i>	226
18.3	<i>Header files and templates</i>	226
18.4	<i>Namespaces and header files</i>	227
18.5	<i>Global variables and header files</i>	227
19	Namespaces	229
19.1	<i>Solving name conflicts</i>	229
19.1.1	Namespace header files	230
19.2	<i>Best practices</i>	231
20	Preprocessor	233
20.1	<i>Textual substitution</i>	233
20.2	<i>Parametrized macros</i>	234

20.3	<i>Conditionals</i>	234
20.3.1	Check on a value	234
20.3.2	Check for macros	235
20.3.3	Including a file only once	235
20.4	<i>Other pragmas</i>	235
21	Templates	237
21.1	<i>Templated functions</i>	237
21.2	<i>Templated classes</i>	238
21.3	<i>Specific implementation</i>	238
21.4	<i>Templating over non-types</i>	238
22	Error handling	241
22.1	<i>General discussion</i>	241
22.2	<i>Mechanisms to support error handling and debugging</i>	242
22.2.1	Assertions	242
22.2.2	Exception handling	242
22.2.3	'Where does this error come from'	244
22.2.4	Legacy mechanisms	244
22.2.5	Legacy C mechanisms	244
22.3	<i>Tools</i>	245
23	Standard Template Library	247
23.1	<i>Complex numbers</i>	247
23.2	<i>Containers</i>	247
23.2.1	Maps: associative arrays	248
23.2.2	Iterators	248
23.3	<i>Union-like stuff: tuples, optionals, variants</i>	250
23.3.1	Tuples	250
23.3.2	Optional	252
23.4	<i>Algorithms</i>	252
23.5	<i>Limits</i>	253
23.5.1	Storage	254
23.6	<i>Random numbers</i>	254
23.7	<i>Time</i>	254
23.8	<i>Sources used in this chapter</i>	254
24	Obscure stuff	263
24.1	<i>Auto</i>	263
24.1.1	Declarations	263
24.1.2	Iterating	264
24.2	<i>Iterating over classes</i>	265
24.3	<i>Lambdas</i>	267
24.3.1	Lambda members of classes	268
24.3.2	Lambda in algorithm	269
24.3.3	Lambda variables by reference	269
24.4	<i>Casts</i>	270
24.4.1	Static cast	270
24.4.2	Dynamic cast	271
24.4.3	Const cast	272

24.4.4	A word about void pointers	272
24.5	<i>Ivalue vs rvalue</i>	273
24.5.1	Conversion	274
24.5.2	References	274
24.5.3	Rvalue references	274
24.6	<i>Move semantics</i>	275
24.7	<i>Graphics</i>	275
24.8	<i>Standards timeline</i>	275
24.8.1	C++11	276
24.8.2	C++14	276
24.8.3	C++17	277
24.8.4	C++20	277
24.9	<i>Sources used in this chapter</i>	277
25	C++ for C programmers	283
25.1	<i>I/O</i>	283
25.2	<i>Arrays</i>	283
25.2.1	Vectors from C arrays	283
25.3	<i>Dynamic storage</i>	284
25.4	<i>Strings</i>	284
25.5	<i>Pointers</i>	284
25.5.1	Parameter passing	284
25.6	<i>Objects</i>	284
25.7	<i>Namespaces</i>	285
25.8	<i>Templates</i>	285
25.9	<i>Obscure stuff</i>	285
25.9.1	Lambda	285
25.9.2	Const	285
25.9.3	Lvalue and rvalue	285
25.10	<i>Sources used in this chapter</i>	285

III	Fortran	287
26	Basics of Fortran	289
26.1	<i>Source format</i>	289
26.2	<i>Compiling Fortran</i>	290
26.3	<i>Main program</i>	290
26.3.1	Program structure	290
26.3.2	Statements	291
26.3.3	Comments	291
26.4	<i>Variables</i>	291
26.4.1	Declarations	292
26.4.2	Precision	292
26.4.3	Initialization	294
26.5	<i>Input/Output, or I/O as we say</i>	294
26.6	<i>Expressions</i>	294
26.7	<i>Review questions</i>	295

26.8	<i>Sources used in this chapter</i>	296
27	Conditionals	299
27.1	<i>Forms of the conditional statement</i>	299
27.2	<i>Operators</i>	299
27.3	<i>Select statement</i>	300
27.4	<i>Boolean variables</i>	300
27.5	<i>Review questions</i>	301
28	Loop constructs	303
28.1	<i>Loop types</i>	303
28.2	<i>Interruptions of the control flow</i>	304
28.3	<i>Implied do-loops</i>	304
28.4	<i>Review questions</i>	305
29	Scope	307
29.1	<i>Scope</i>	307
29.1.1	<i>Variables local to a program unit</i>	307
29.1.2	<i>Variables in an internal procedure</i>	308
30	Subprograms and modules	309
30.1	<i>Procedures</i>	309
30.1.1	<i>Subroutines and functions</i>	309
30.1.2	<i>Return results</i>	311
30.1.3	<i>Arguments</i>	313
30.1.4	<i>Types of procedures</i>	314
30.1.5	<i>More about arguments</i>	314
30.2	<i>Interfaces</i>	315
30.2.1	<i>Polymorphism</i>	315
30.3	<i>Sources used in this chapter</i>	316
31	String handling	321
31.1	<i>String denotations</i>	321
31.2	<i>Characters</i>	321
31.3	<i>Strings</i>	321
31.4	<i>Strings versus character arrays</i>	322
31.5	<i>Sources used in this chapter</i>	322
32	Structures, eh, types	323
33	Modules	325
33.1	<i>Modules for program modularization</i>	325
33.2	<i>Modules</i>	326
33.2.1	<i>Polymorphism</i>	327
33.2.2	<i>Operator overloading</i>	327
34	Classes and objects	329
34.1	<i>Classes</i>	329
35	Arrays	333
35.1	<i>Static arrays</i>	333
35.1.1	<i>Initialization</i>	334
35.1.2	<i>Array sections</i>	334
35.1.3	<i>Integer arrays as indices</i>	335
35.2	<i>Multi-dimensional</i>	335

35.2.1	<i>Querying an array</i>	337
35.2.2	<i>Reshaping</i>	337
35.3	<i>Arrays to subroutines</i>	337
35.4	<i>Allocatable arrays</i>	338
35.5	<i>Array output</i>	339
35.6	<i>Operating on an array</i>	339
35.6.1	<i>Arithmetic operations</i>	339
35.6.2	<i>Intrinsic functions</i>	339
35.6.3	<i>Restricting with where</i>	340
35.6.4	<i>Global condition tests</i>	340
35.7	<i>Array operations</i>	341
35.7.1	<i>Loops without looping</i>	341
35.7.2	<i>Loops without dependencies</i>	342
35.7.3	<i>Loops with dependencies</i>	342
35.8	<i>Review questions</i>	343
35.9	<i>Sources used in this chapter</i>	343
36	Pointers	345
36.1	<i>Basic pointer operations</i>	345
36.2	<i>Pointers and arrays</i>	347
36.3	<i>Example: linked lists</i>	347
36.4	<i>Sources used in this chapter</i>	350
37	Input/output	351
37.1	<i>Types of I/O</i>	351
37.2	<i>Print to terminal</i>	351
37.2.1	<i>Print on one line</i>	352
37.2.2	<i>Printing arrays</i>	352
37.2.3	<i>Formats</i>	352
37.3	<i>File and stream I/O</i>	354
37.3.1	<i>Units</i>	354
37.3.2	<i>Other write options</i>	354
37.4	<i>Unformatted output</i>	354
37.5	<i>Sources used in this chapter</i>	355
38	Leftover topics	357
38.1	<i>Random numbers</i>	357
38.2	<i>Timing</i>	357
IV	Exercises and projects	359
39	Exercises	361
39.1	<i>Arithmetic</i>	361
39.2	<i>Scope</i>	361
39.3	<i>Looping</i>	362
39.4	<i>Subprograms</i>	362
39.5	<i>Object oriented exercises</i>	363
39.6	<i>List access</i>	364
40	Prime numbers	365

40.1	<i>Arithmetic</i>	365
40.2	<i>Conditionals</i>	365
40.3	<i>Looping</i>	365
40.4	<i>Functions</i>	366
40.5	<i>While loops</i>	366
40.6	<i>Structures</i>	366
40.7	<i>Classes and objects</i>	367
40.8	<i>Eratosthenes sieve</i>	368
40.8.1	<i>Arrays implementation</i>	368
40.8.2	<i>Streams implementation</i>	368
40.9	<i>Range implementation</i>	369
41	Geometry	371
41.1	<i>Basic functions</i>	371
41.2	<i>Point class</i>	371
41.3	<i>Using one class in another</i>	372
41.4	<i>Is-a relationship</i>	374
41.5	<i>More stuff</i>	374
42	Infectuous disease simulation	375
42.1	<i>Model design</i>	375
42.1.1	<i>Other ways of modeling</i>	375
42.2	<i>Coding up the basics</i>	376
42.3	<i>Population</i>	377
42.4	<i>Contagion</i>	377
42.5	<i>Spreading</i>	378
42.6	<i>Project writeup and submission</i>	378
42.6.1	<i>Program files</i>	378
42.6.2	<i>Writeup</i>	378
43	PageRank	381
43.1	<i>Basic ideas</i>	381
43.2	<i>Clicking around</i>	382
43.3	<i>Graph algorithms</i>	382
43.4	<i>Page ranking</i>	383
43.5	<i>Graphs and linear algebra</i>	384
43.6	<i>Sources used in this chapter</i>	384
44	Redistricting	385
44.1	<i>Basic concepts</i>	385
44.2	<i>Basic functions</i>	386
44.2.1	<i>Voters</i>	386
44.2.2	<i>Populations</i>	386
44.2.3	<i>Districting</i>	387
44.3	<i>Strategy</i>	388
44.4	<i>Efficiency: dynamic programming</i>	389
44.5	<i>Extensions</i>	389
45	Amazon delivery truck scheduling	391
45.1	<i>Problem statement</i>	391
45.2	<i>Coding up the basics</i>	391

45.2.1	Address list	391
45.2.2	Add a depot	392
45.2.3	Greedy construction of a route	392
45.3	<i>Optimizing the route</i>	393
45.4	<i>Multiple trucks</i>	394
45.5	<i>Amazon prime</i>	394
45.6	<i>Dynamicism</i>	395
46	Memory allocation	397
47	DNA Sequencing	399
47.1	<i>Basic functions</i>	399
48	Cryptography	401
48.1	<i>Basic ideas</i>	401
49	Climate change	403
49.1	<i>Reading the data</i>	403
49.2	<i>Statistical hypothesis</i>	403
 V Advanced topics 405		
50	Programming strategies	407
50.1	<i>A philosophy of programming</i>	407
50.2	<i>Programming: top-down versus bottom up</i>	407
50.2.1	Worked out example	408
50.3	<i>Coding style</i>	409
50.4	<i>Documentation</i>	409
50.5	<i>Testing</i>	410
50.6	<i>Best practices: C++ Core Guidelines</i>	410
51	Tiniest of introductions to algorithms and data structures	411
51.1	<i>Data structures</i>	411
51.1.1	Stack	411
51.1.2	Linked lists	411
51.1.3	Trees	416
51.2	<i>Algorithms</i>	418
51.2.1	Sorting	418
51.3	<i>Programming techniques</i>	419
51.3.1	Memoization	419
52	Complexity	421
52.1	<i>Order of complexity</i>	421
52.1.1	Time complexity	421
52.1.2	Space complexity	421
 VI Index and such 423		
53	Index	425

Contents

PART I

INTRODUCTION

Chapter 1

Introduction

1.1 Programming and computational thinking

1.1.1 History

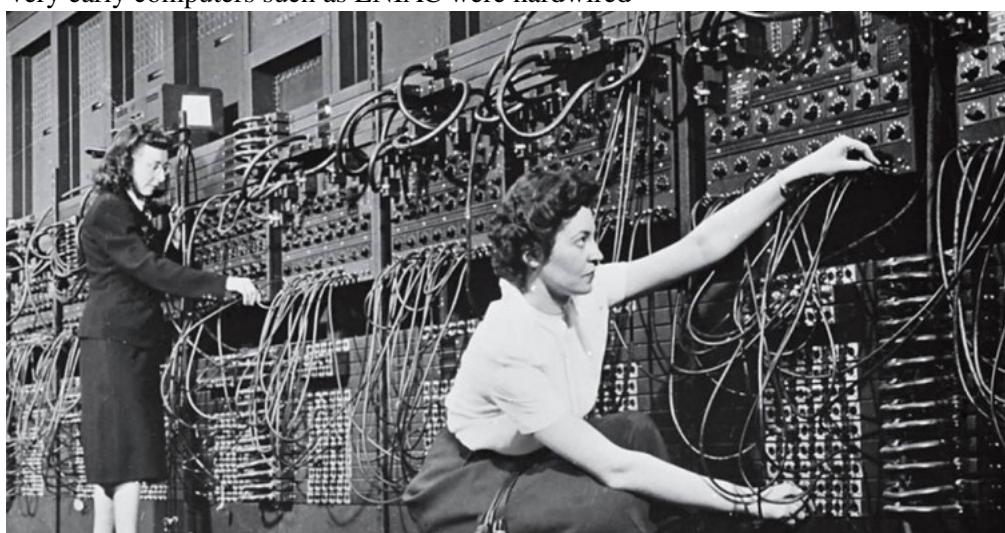
Earliest computers

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



Hands-on programming

Very early computers such as ENIAC were hardwired



1. Introduction

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

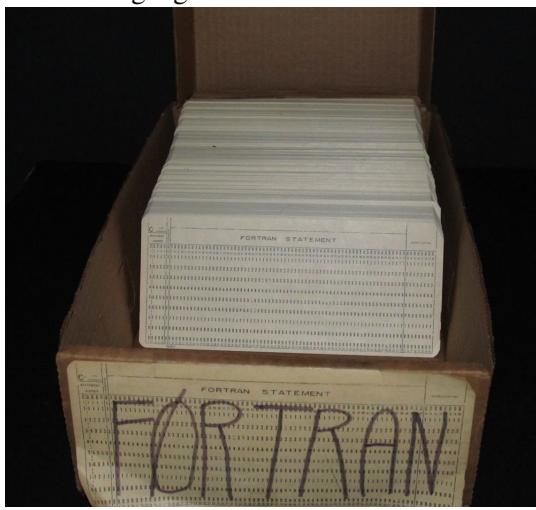
Program entry

Later programs were written on punchcards



The first programming language

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



Programming is everywhere

Programming is used in many different ways these days.

- You can make your own commands in Microsoft Word.

- You can make apps for your *smartphone*.
- You can solve the riddles of the universe using big computers.

This course is aimed at people in the last category.

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

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 PCs¹.

There are scientific aspects to programming:

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

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

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

1.1.3 Computational thinking

Programming is not simple

Programs can get pretty big:

It’s not just translating formulas anymore.

Translating ideas to computer code: computational thinking.

Computational thinking: elevator scheduling

Mathematical thinking:

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

Sufficient condition ≠ existence proof.

Computational thinking: actual design of a solution

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

Coming up with a strategy takes creativity!

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

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

Exercise 1.2. Algorithms are usually not uniquely determined. There can be cleverness involved.

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 two cars using different algorithms meet each other?

Computation and complexity

Looking up a name in the phone book

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

What is the average search time in the two cases?

Having a correct solution is not enough!

Programming languages are about ideas

A powerful programming language serves as a framework within which we organize our ideas. Every programming language has three mechanisms for accomplishing this:

- primitive expressions
- means of combination
- means of abstraction

Abelson and Sussman, The Structure and Interpretation of Computer Programs

Abstraction

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

A program has layers of abstractions.

Abstraction is good

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

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

Good programming style makes code intelligible and maintainable.

(Bad programming style may lead to lower grade.)

Trust him, he’s Dutch!

The competent programming 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

Data abstraction

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 flight software for the Apollo space 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

Do you have to know much about hardware?

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

<https://youtu.be/JEpsKnWZrJ8>

Advanced programmers know that hardware influences the speed of execution (see TACC's ISTC course).

1.1.5 Algorithms

What is an algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time

[A. Levitin, Introduction to The Design and Analysis of Algorithms, Addison-Wesley, 2003]

The instructions are in some language:

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

Program steps

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

Program data

- Input and output data: to/from file, user input, screen output, graphics.
- Data during the program run:
 - Simple variables: character, integer, floating point
 - Arrays: indexed set of characters and such

- Data structures: trees, queues
 - * Defined by the user, specific for the application
 - * Found in a library (big difference between C/C++!)

1.2 About the choice of language

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

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

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

Comparing two languages

Python vs C++ on bubblesort:

```

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.

The right language is not all

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.

1.3 Further reading

Tutorial, assignments: <http://www.cppforschool.com/>

Problems to practice: <http://www.spoj.com/problems/classical/>

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. Install XQuartz and 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.

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.

3. Teachers guide

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	homework	prime	geom	infect
1	Statements and ex- pressions	4.11		40.1		
2	Conditionals	5.3		40.2		
4	Looping	6.4		40.3, 40.4, 40.6		
5	continue					
6	Functions	40.5		40.5		
7	continue				41.1	
8	I/O					
9	Structs			40.7		
10	Objects			40.8, 40.10	41.3	42.1
11	continue					
12	has-a rela- tion				41.9, 41.10, 41.1, 41.12	42.2
13	inheritance				41.13, 41.14	
14	Arrays					42.2 and further
15	continue					
16	Strings					
Advanced						
Pointers and C-style addresses		Section				
		51.1.2				
Prototypes (and separate compi- lation) and namespaces						
Error handling and exceptions		22.1				
Lambdas		24.3				

Table 3.1: Two-month lesson plan for C++

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 40.

geom Geometry related concepts; this is mostly an exercise in object-oriented programming. Chapter 41.

infect The spreading of infectious diseases; these are exercises in object-oriented design. Students can explore various real-life scenarios. Chapter 42.

pagerank The Google Pagerank algorithm. Students program a simulated internet, and explore pageranking, including ‘search engine optimization’. This exercise uses lots of pointers. Chapter 43.

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

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

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.

PART II

C++

Chapter 4

Basic elements of C++

4.1 From the ground up: Compiling C++

In this chapter and the next you are going to learn the C++ language. But first we need some externalia: how do you deal with any program?

Two kinds of files

In programming you have two kinds of files:

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

Your source files are translated to binary by a *compiler*.

Let's say that

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

which you can verbalize as ‘invoke the `g++` (or `icpc`) compiler, with output `myprogram`, on `myprogram.cxx`’.

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
```

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

Run this program (it gives no output):

```
./zeroprogram
```

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

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*.

Exercise 4.3. 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.

Program statements

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

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

and arbitrary:

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

Exercise 4.4. 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.

Errors

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

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

Exercise 4.5. True or false?

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

Fixed elements

You see that certain parts of your program are inviolable:

- There are *keywords* such as `return` or `cout`; you can not change their definition.
- Curly braces and parentheses need to be matched.
- There has to be a `main` keyword.
- The `iostream` and `std` are usually needed.

Exercise 4.6. 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?

Return 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:`

`make[5]: *** No rule to make target 'run_r...`

For the source of this example, see section [4.8.1](#)

4.3 Variables

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

- a name, so that you can refer to them,

- a *datatype*, and
- a value.

Think of a variable as a labeled placed in memory.

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

Typical variable lifetime

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

4.3.1 Variable declarations

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

Variable names

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

Declaration

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

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

Where do declarations go?

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

Note: it is legal to define a variable before the main program but that's not a good idea. Please only declare *inside* `main` (or inside a function et cetera).

Exercise 4.7. Which of the following are legal variable names?

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

4.3.2 Assignments

Setting a variable

```
|| i = 5;
```

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

let $i = 5$.

Assignment

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

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

Variable of the left-hand side gets value of the right-hand side.

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

Assignments

A variable can be given a value more than once. The following sequence of statements is a legitimate part of a program:

```
|| int n;
|| n = 3;
|| n = 2*n + 5;
|| n = 3*n + 7;
```

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

Special forms

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 4.8. 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 4.9.

```
#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 Datatypes

Datatypes

Variables come in different types;

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

For complex numbers see section 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 stored in an `int` or `float` variable. This is addressed in section 23.5.

4.3.3.1 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 4.10. 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;
```

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.

4.3.3.2 Floating point variables

floating point numbers.

- On the other hand, not all real numbers have a floating point representation. For instance, since computer numbers are binary, $1/2$ is representable but $1/3$ is not.
- You can assign variables of one type to another, but this may lead to truncation (assigning a floating point number to an integer) or unexpected bits (assigning a single precision floating point number do a double precision).

Floating points numbers do not behave like mathematical numbers.

Warning: floating point arithmetic

Floating point arithmetic is full of pitfalls.

- Don't count on $3 * (1 / 3)$ 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 4.12. 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 a range of inputs. Are there ones that surprise you?

Complex numbers exist, see section 23.1.

4.3.3.3 Boolean values

Truth values

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

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

4.3.3.4 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:

Quick intro to 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 4.13. Write a program that asks for the user's first name, and prints something like
Hello, Susan! in response.

What happens if you enter first and last name?

4.3.4 Initialization

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

Initialization syntax

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. Initialization is often omitted for performance reasons.

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

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

Terminal output

You have already seen `cout`:

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

Terminal input

There is also a `cin`, which serves to take user input and put it in a numerical variable.

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

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

There is also `getline`, which is more general.

For more I/O, see chapter 13.

4.5 Expressions

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

4.5.1 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 library calls

Math function in `cmath`:

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

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

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.

Boolean expressions

- Testing: `==` `!=` `<` `>` `<=` `>=`
- Not, and, or: `!` `&&` `||`

- Shortcut operators:

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

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.

Bit operators

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 4.14.

- 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.

Other conversation are done through a *cast*; see section 24.4.

Exercise 4.15. Write a program that asks for two integer numbers n_1, n_2 .

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

$$n_1 \mod n_2$$

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

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

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?

Exercise 4.17. True or false?

1. Within a certain range, all integers are available as values of an integer variable.
2. Within a certain range, all real numbers are available as values of a float variable.
3. $5(7+2)$ is equivalent to 45.
4. $1--1$ is equivalent to zero.
5. `int i = 2./3.;` The variable `i` is 1.
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 $-2^{31} \dots 0 \dots 2^{31} - 1$.
- Floating point numbers are represented by a sign bit, an exponent, and a number of significant digits. For 4-byte numbers, the number of significant (decimal) digits is around 6; for 8-byte numbers it is around 15.

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

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

- *Inf*: infinity. This happens if you add or multiply enough large numbers together. There is of course also a value $-\text{Inf}$. 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 23.5 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

Integer constants

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++14.

Floating point constants

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

This prevents numerical accidents:

```
|| double x = 3.;
```

converts float to double, maybe introducing random bits.

4.7 Review questions

Exercise 4.18. What is the output of:

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

Exercise 4.19. Given

```
|| int n;
```

give an expression that uses elementary mathematical operators to compute 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?

4.8 Sources used in this chapter

4.8.1 Listing of code/basic/return

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** return.cxx : nonzero return result from main  
*****  
*****  
<#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
//codesnippet returnone  
int main() {  
    return 1;  
}  
//codesnippet end
```

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 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'?

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

5.2 Operators

You have already seen arithmetic expressions; now we need to look at logical expressions: just what can be tested in a conditional. Here is a fragment of language grammar that spells out what is legal. You see that most of the rules are recursive, but there is an important exception.

What are logical expressions?

```

logical_expression ::=
    comparison_expression
    | NOT comparison_expression
    | logical_expression CONJUNCTION comparison_expression
comparison_expression ::=
    numerical_expression COMPARE numerical_expression
numerical_expression ::=
    quantity
    | numerical_expression OPERATOR quantity
quantity ::= number | variable

```

Comparison and logical operators

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

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

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;
```

Can you rewrite your test so that the output lines are switched?

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

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.

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.

Switch statement example

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:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section [5.5.1](#)

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.
```

5.5 Sources used in this chapter

5.5.1 Listing of code/basic/switch

```
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** switch.cxx : illustrating switch statement
*****
*****
```

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

```
int main() {
    int n;
    cin >> n;
    //codesnippet switchstatement
    switch (n) {
        case 1 :
        case 2 :
            cout << "very small" << endl;
            break;
        case 3 :
            cout << "trinity" << endl;
            break;
        default :
            cout << "large" << endl;
    }
    //codesnippet end
    return 0;
}
```


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:

```
|| for (int var=low; var<upper; var++) {  
    // statements involving the loop variable:  
    cout << "The square of " << var << " is " << var*var << 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 1 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

6. Looping

```
|| 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.

Popular increments

- `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++)
```

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:

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section 6.5.1

(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;
    || }
```

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++)
    ||     ...
```

6. Looping

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?

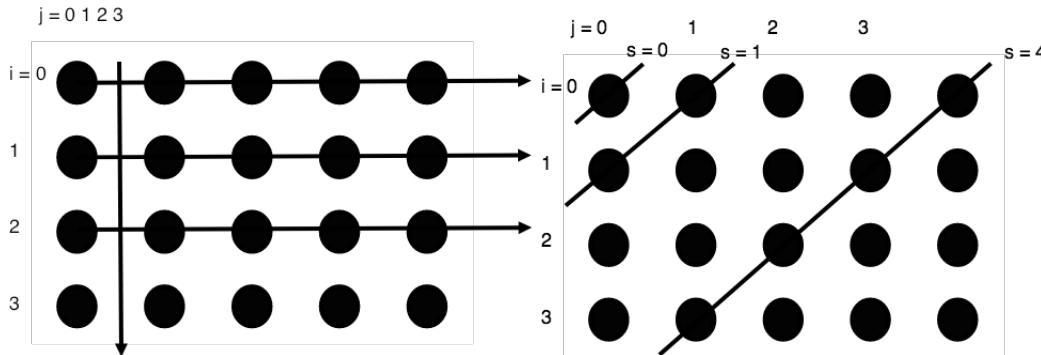


Figure 6.1: Lexicographic and diagonal ordering of an index set

Figure 6.1 illustrates that you can look at the i, j indices by row/column or by diagonal. Just like rows and columns being defined as $i = \text{constant}$ and $j = \text{constant}$ respectively, a diagonal is defined by $i + j = \text{constant}$.

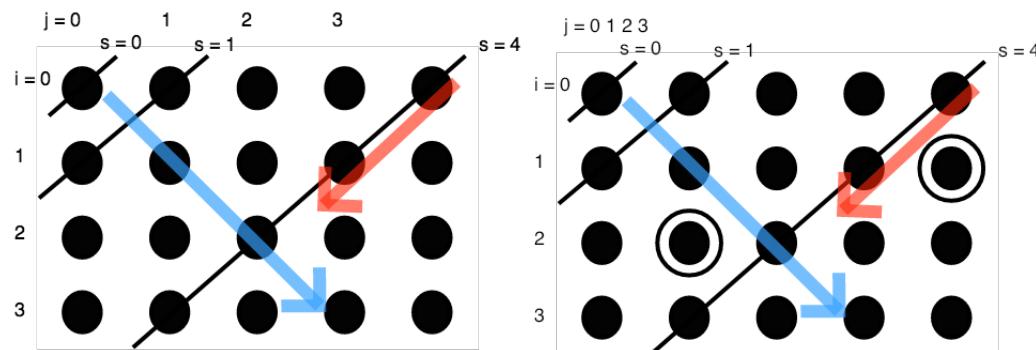


Figure 6.2: Illustration of the second part of exercise 6.4

6.2 Looping until

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

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

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

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

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

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 you read in. A good test case is $N = 40$.

Secondly, find pair with $i \cdot j > N$, but with the smallest value for $i + j$. 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, ..., then find i, j .

Your 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 */ }
```

?

Where did the break happen?

Suppose you want to know what the loop variable was when the break happened.

You need the loop variable to be global:

```
|| 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!

Test in the loop header

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

6. Looping

```
|| bool some_test{false};  
|| for (int var=low; !some_test ; var++) {  
| ... some code ...  
| some_test = ... some condition ...  
| }
```

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

Skip iteration

```
|| 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.

While loop

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

While syntax 1

```
|| cout << "Enter a positive number: " ;  
|| cin >> invar;  
|| while (invar>0) {  
| | cout << "Enter a positive number: " ;
```

```

    } cin >> invar;
}
cout << "Sorry, " << invar << " is negative" << endl;

```

Problem: code duplication.

While syntax 2

```

do {
    cout << "Enter a positive number: ";
    cin >> invar;
} while (invar>0);
cout << "Sorry, " << invar << " is negative" << endl;

```

The post-test syntax leads to more elegant code.

Exercise 6.6. 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`.

Exercise 6.7. A horse is tied to a pole with a 1 meter elastic band. A spider that was sitting on the pole 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.

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

$7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \dots$

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

Try all starting values $u_1 = 1, \dots, 1000$ 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

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

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>

6.5 Sources used in this chapter

6.5.1 Listing of code/basic/pretest

```
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** pretest.cxx : illustrating if test
*****
***** /*****
```

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

int main() {

    //codesnippet pretest
    cout << "before the loop" << endl;
    for (int i=5; i<4; i++)
        cout << "in iteration "
            << i << endl;
    cout << "after the loop" << endl;
    //codesnippet end
    return 0;
}
```

6. Looping

Chapter 7

Functions

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

- Find a block of code that has a clearly identifiable function.
- Turn this into a function: the function definition will contain that block, plus a header and (maybe) a return statement.
- The function definition is placed before the main program.
- The function is called by its name.

Introducing a function

program

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



float foo_compute(float x) {

```
// foo computation  
xtmp = ... x ...  
ytmp = ... x ... xtmp ....  
return .... xtmp .... ytmp ....
```

function

```
x = ...  
y = foo_compute(x);  
.... y ....
```

program

Using a function can also shorten your code, since it replaces two similar code blocks with one function definition and two calls.

Even if the number of lines doesn't go down, there are still reasons for using functions.

- By removing *code duplication* you have removed a likely source of future errors: if you later edit one occurrence of the code block, you're likely to forget the other.
- By introducing 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.

Program without functions

Everything in the main:

```

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

```

Function definition and call

Function for which we compute the zero:

```

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

```

Function for zero finding:

```

float f(float m) {
    return m*m*m - m*m-1;
};
float find_zero_between(float l,float r) {
    while (r-l>.1) {
        float mid = (l+r)/2.,
        fmid = f(mid);
        if (fmid<0) l = mid;
    else r = mid;
    }
    cout << "Zero happens at: " << mid << endl;
}

```

Code becomes more readable (though not necessarily shorter): introduce application terminology.

In this example, the function definition consists of:

- The keyword `void` indicates that the function does not give any results back to the main program.
- The name `report_evenness` 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.

The *function call* consists of

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

In the previous example, the function had an input, and performed some screen output. To have a function that takes part in the computation of your program, you would write something like:

```
int my_computation(int i) {
    return i+3;
}
...
// in the main:
int result;
result = my_computation(5);
```

Here is a program with a function that doubles its input:

Function definition and call

Code:

```
int double_this(int n) {
    int twice_the_input = 2*n;
    return twice_the_input;
}
/* ... */
int number = 3;
cout << "Twice three is: "
    << double_this(number) << endl;
```

Output

[func] twicein:

make[5]: *** No rule to make target 'run_t

For the source of this example, see section [7.10.1](#)

7.1.1 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, like 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.

7.2 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 occurrences of the function code do not have to be identical:

Code reuse

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);
```

Code reuse

Example: multiple norm calculations:

Repeated code:

becomes:

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

(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.

- Maintenance: 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.
- Functions have to be defined before you can use them.

7.3 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:

Function call

The function call

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

7.3.1 Examples

Functions with return result

```
|| #include <cmath>
double pi() {
    return 4*atan(1.0);
}
```

The *atan* is a *standard function*

Functions with input and output

```
float squared(float x) {
    return x*x;
}
```

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.

7.4 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**.

Functions without input, without return result

```
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 function with input

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

Review 7.2. True or false?

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

7.5 Parameter passing

C++ functions resemble mathematical functions: you have seen that a function can have an input and an output. In fact, they can have multiple inputs, separated by commas, but they have only one output¹.

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

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

7.5.1 Pass by value

The following style of programming is very much inspired by mathematical functions, and is known as *functional programming*².

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

$$\| y = f(x_1, x_2, x_3);$$
- The definition of the C++ parameter passing mechanism says that input arguments are copied to the function, meaning that they don't change in the calling program:

Code:

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

Output

[func] passvalue:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section 7.10.2

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.

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:

1. Unless you use more sophisticated mechanisms.
 2. 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:

```
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:

make[5]: *** No rule to make target 'run_1

For the source of this example, see section [7.10.3](#)

Exercise 7.1. If you are doing the prime project (chapter [40](#)) you can now do exercise [40.5](#).

Background Square roots through Newton

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 7.2.

- 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 $|f(x, y)| < 10^{-5}$. Print x .
- Second part: write a function `newton_root` that computes \sqrt{y} .

7.5.2 Pass by reference

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

We need the concept of a *reference*:

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:**

make[5]: *** No rule to make target `run_r

For the source of this example, see section 7.10.4

(You will not use references often this way.)

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

Parameter passing by reference

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

We also the following terminology for function parameters:

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

*Pass by reference example 1***Code:**

```

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

    int var = 0;
    f(var);
    cout << var << endl;
}

```

Output**[basic] setbyref:**

make[5]: *** No rule to make target `run_s

For the source of this example, see section 7.10.5

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.

Pass by reference example 2

```

bool can_read_value( int &value ) {
    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;
}

```

Exercise 7.3. 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 7.4. 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 7.5. If you are doing the geometry project, you should now do the exercises in section 41.1.

Const parameters

You can prevent local changes to the function parameter:

```

/* This does not compile:
void change_const_scalar(const int i) { i += 1; }
*/

```

This is mostly to protect you against yourself.

7.6 Recursive functions

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

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

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

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

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

```
|| 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 7.6. The sum of squares:

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

can be defined recursively as

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

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

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

Exercise 7.7. Write a recursive function for computing Fibonacci numbers:

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

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

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

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

Remark 4 If you let your Fibonacci program print out each time it computes a value, you'll see that most values are computed several times. (Math question: how many times?) This is wasteful in running time. This problem is addressed in section [51.3.1](#).

7.6.1 Stack overflow

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

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

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 correct³, 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*.

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

7.7 Advanced function topics

7.7.1 Default arguments

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.

7.7.2 Polymorphic functions

Polymorphic functions

You can have multiple functions with the same name:

```
|| double sum(double a,double b) {
    ||| return a+b;
}
|| double sum(double a,double b,double c) {
    ||| return a+b+c;
}
```

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

7.8 Library functions

7.8.1 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 23.6):

```
||#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.

7.9 Review questions

Exercise 7.8. What is the output of the following programs (assuming the usual headers):

<pre> int add1(int i) { return i+1; } int main() { int i=5; i = add1(i); cout << i << endl; } </pre>	<pre> void add1(int i) { i = i+1; } int main() { int i=5; add1(i); cout << i << endl; } </pre>
<pre> void add1(int &i) { i = i+1; } int main() { int i=5; add1(i); cout << i << endl; } </pre>	<pre> int add1(int &i) { return i+1; } int main() { int i=5; i = add1(i); cout << i << endl; } </pre>

Exercise 7.9. 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.

Exercise 7.10. 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.

Exercise 7.11. 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.

Exercise 7.12. 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.

7.10 Sources used in this chapter

7.10.1 Listing of code/func/twicein

```
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** twicein.cxx : simple function illustration
*****
*****
```

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

//examplesnippet twicein
int double_this(int n) {
    int twice_the_input = 2*n;
    return twice_the_input;
}
//examplesnippet end

int main() {
    //examplesnippet twicein
    int number = 3;
    cout << "Twice three is: "
        << double_this(number) << endl;
    //examplesnippet end
    return 0;
}
```

7.10.2 Listing of code/func/passvalue

```
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** passvalue.cxx : illustration pass-by-value
*****
*****
```

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

//examplesnippet passvalue
double squared( double x ) {
    x = x*x;
    return x;
}
//examplesnippet end

int main() {

    double number, other;
```

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

return 0;
}
```

7.10.3 Listing of code/func/localparm

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** localparm.cxx : simple parameter passing
*****
*****
```

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

#include <vector>
using std::vector;

//examplesnippet localparm
void change_scalar(int i) {
    i += 1;
}
//examplesnippet end

int main() {

    int number;

    //examplesnippet localparm
    number = 3;
    cout << "Number is 3: "
        << number << endl;
    change_scalar(number);
    cout << "is it still 3? Let's see: "
        << number << endl;
    //examplesnippet end

    return 0;
}
```

7.10.4 Listing of code/basic/ref

```
*****  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** ref.cxx : using references, not as parameter  
****  
*****  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
int main() {  
  
    //codesnippet refint  
    int i;  
    int &ri = i;  
    i = 5;  
    cout << i << "," << ri << endl;  
    i *= 2;  
    cout << i << "," << ri << endl;  
    ri -= 3;  
    cout << i << "," << ri << endl;  
    //codesnippet end  
  
    return 0;  
}
```

7.10.5 Listing of code/basic/setbyref

```
*****  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** setbyref.cxx : illustration of passing by ref  
****  
*****  
  
#include <iostream>  
using std::cout;  
using std::endl;  
  
//codesnippet setbyref  
void f( int &i ) {  
    i = 5;  
}  
int main() {  
  
    int var = 0;  
    f(var);  
    cout << var << endl;  
    //codesnippet end
```

```
    ||  return 0;  
    || }
```

Chapter 8

Scope

8.1 Scope rules

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

8.1.1 Lexical scope

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

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

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

make[5]: *** No rule to make target 'run_s

For the source of this example, see section ??

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() << endl;
    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:

make[5]: *** No rule to make target 'run_d

For the source of this example, see section [10.6.10](#)

8.4 Review questions

Exercise 8.4. 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?

Exercise 8.5. 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;
}
```

Exercise 8.6. 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;
}
```

Exercise 8.7. 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;
}
```

8.5 Sources used in this chapter

8.5.1 Listing of code/object/destructor

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** destructor.cxx : illustration of objects going out of scope
*****
*****
```

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

//examplesnippet destructor
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor"
        << endl;
    };
    ~SomeObject() {
        cout << "calling the destructor"
        << endl;
    };
};
//examplesnippet end

int main() {

//examplesnippet destructoruse
    cout << "Before the nested scope"
    << endl;
{
    SomeObject obj;
    cout << "Inside the nested scope"
    << endl;
```

```
    }
    cout << "After the nested scope"
    << endl;
//examplesnippet end

    return 0;
}
```

8. Scope

Chapter 9

Structures

9.1 Why structures?

You have seen the basic datatypes in section 4.3.3. 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.

```
// struct vector {
//   double x; double y; int label;
// };
```

The elements of a structure are also called *members*. You can give them an initial value:

```
// struct vector { double x=0.; double y=0.; } ;
```

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, for instance if you need to pass the structure to a function.

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

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

- 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.

Struct initialization

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

```
struct vector_a { double x; double y; } ;
struct vector_b { double x=0; double y=0; } ;

int main() {

    // initialization when you create the variable:
    struct vector_a x_a = {1.5,2.6};
    // initialization done in the structure definition:
    struct vector_b x_b;
    // ILLEGAL:
    // x_b = {3.7, 4.8};
    x_b.x = 3.7; x_b.y = 4.8;
```

Using structures

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 vector v1,v2;

    v1.x = 1.; v1.y = 2.; v1.label = 5;
    v2 = {3.,4.,5};

    v2 = v1;
    cout << "v2: "
        << v2.x << "," << v2.y
        << endl;
```

Output

[struct] point:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [9.3.1](#)

Period syntax: ‘apostrophe-s’.

Exercise 9.1. True or false?

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

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

creates an object with an integer and a double as members.

- With the above definition and `struct numbered xn;`,

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

is correct C++.

- With the same definitions, this is correct C++:

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

Note: if you use initializations in the `struct` definition, you can not use the brace-assignment.

Functions on structures

You can pass a structure to a function:

Code:

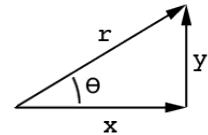
```

double distance
( struct vector v1,
  struct vector v2 )
{
    double
        d1 = v1.x-v2.x, d2 = v1.y-v2.y;
    return sqrt( d1*d1 + d2*d2 );
}
/* ... */
struct vector v1 = { 1.,1. };
cout << "Displacement x,y?";
double dx,dy; cin >> dx >> dy; cout << endl;
cout << "dx=" << dx << ", dy=" << dy << endl;
struct vector v2 = { v1.x+dx,v1.y+dy };
cout << "Distance: " << distance(v1,v2) << endl;

```

Output**[struct] pointfun:**

make[5]: *** No rule to make target 'run_p

For the source of this example, see section 9.3.3

Exercise 9.2. Write a function that, given a vector as defined above, returns the angle with the x -axis. (Hint: the `atan` function is in `cmath`)

Code:

```

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

```

Output**[struct] pointangle:**

make[5]: *** No rule to make target 'run_p

Exercise 9.3. Write a `void` function that has a **struct vector** parameter, and exchanges its coordinates:

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

Code:

```

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

```

Output**[struct] pointflip:**

make[5]: *** No rule to make target 'run_p

Exercise 9.4. Write a function $y = f(x, a)$ that takes a **struct vector** and **double** parameter as input, and returns a vector that is the input multiplied by the scalar.

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

9. Structures

Exercise 9.5. If you are doing the prime project (chapter 40) you can now do exercise 40.7.

Exercise 9.6. Write a function `inner_product` that takes two `vector` structures and computes the inner product.

Denotations

You can use initializer lists as struct *denotations*:

Code:

```
|| cout << "Distance: "
  ||<< distance( {1.,2.}, {6.,14.} )
  ||<< endl;
```

Output

[struct] pointdenote:

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section 9.3.3

Exercise 9.7. Take exercise 9.6 and rewrite it to use denotations.

Returning structures

You can return a structure from a function:

Code:

```
|| struct vector vector_add
  || ( struct vector v1,
    ||   struct vector v2 ) {
  || struct vector p_add =
  ||   {v1.x+v2.x, v1.y+v2.y};
  || return p_add;
  || };
  /* ... */
  v3 = vector_add(v1, v2);
  cout << "Added: " <<
  v3.x << "," << v3.y << endl;
```

Output

[struct] pointadd:

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section 9.3.4

(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, for instance using vectors to store the matrix rows, and then using the inner product method to multiply matrices?

Passing structures by reference

Prevent copying cost by passing by reference, use `const` to prevent changes:

```
|| double distance( const struct vector &v1, const struct vector &v2 ) {
  ||   double d1 = v1.x-v2.x, d2 = v1.y-v2.y;
  ||   return sqrt( d1*d1 + d2*d2 );
  || }
```

9.3 Sources used in this chapter

9.3.1 Listing of code/struct/point

```

/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2016-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** point.cxx : struct for cartesian vector
*****
***** /



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

//codesnippet structdef
struct vector {
    double x; double y; int label;
};
//codesnippet end

//codesnippet structuse
int main() {

    struct vector v1,v2;

    v1.x = 1.; v1.y = 2.; v1.label = 5;
    v2 = {3.,4.,5};

    v2 = v1;
    cout << "v2: "
        << v2.x << "," << v2.y
        << endl;
    //codesnippet end

    return 0;
}

```

9.3.2 Listing of code/struct/pointfun

```

/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** pointfun.cxx : function taking struct arguments
*****
***** /


#include <cmath>

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

```

```

struct vector { double x; double y; } ;

//codesnippet structpass
double distance
( struct vector v1,
struct vector v2 )
{
double
d1 = v1.x-v2.x, d2 = v1.y-v2.y;
return sqrt( d1*d1 + d2*d2 );
}
//codesnippet end

int main() {

    cout << "Struct Pass" << endl;
//codesnippet structpass
struct vector v1 = { 1.,1. };
cout << "Displacement x,y?";
double dx,dy; cin >> dx >> dy; cout << endl;
cout << "dx=" << dx << ", dy=" << dy << endl;
struct vector v2 = { v1.x+dx,v1.y+dy };
cout << "Distance: " << distance(v1,v2) << endl;
//codesnippet end
cout << ".. struct pass" << endl;

cout << "Struct Denote" << endl;
//codesnippet structdenote
cout << "Distance: "
<< distance( {1.,2.}, {6.,14.} )
<< endl;
//codesnippet end
cout << ".. struct denote" << endl;

return 0;
}

```

9.3.3 Listing of code/struct/pointfun

```

/*********************************************
 ****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
****
**** pointfun.cxx : function taking struct arguments
****

/*****************************************/
#include <cmath>

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

struct vector { double x; double y; } ;

```

```

//codesnippet structpass
double distance
( struct vector v1,
struct vector v2 )
{
double
d1 = v1.x-v2.x, d2 = v1.y-v2.y;
return sqrt( d1*d1 + d2*d2 );
}
//codesnippet end

int main() {
    cout << "Struct Pass" << endl;
//codesnippet structpass
struct vector v1 = { 1.,1. };
cout << "Displacement x,y?";
double dx,dy; cin >> dx >> dy; cout << endl;
cout << "dx=" << dx << ", dy=" << dy << endl;
struct vector v2 = { v1.x+dx,v1.y+dy };
cout << "Distance: " << distance(v1,v2) << endl;
//codesnippet end
cout << "... struct pass" << endl;

cout << "Struct Denote" << endl;
//codesnippet structdenote
cout << "Distance: "
<< distance( {1.,2.}, {6.,14.} )
<< endl;
//codesnippet end
cout << "... struct denote" << endl;

return 0;
}

```

9.3.4 Listing of code/struct/pointadd

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** pointadd.cxx : function returning struct  
*****  
*****  
#include <cmath>  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
struct vector { double x; double y; } ;
```

```
//codesnippet structreturn
struct vector vector_add
  ( struct vector v1,
    struct vector v2 ) {
  struct vector p_add =
    {v1.x+v2.x, v1.y+v2.y};
  return p_add;
}
//codesnippet end

int main() {

  struct vector v1, v2, v3;

  v1.x = 1.; v1.y = 1.;
  v2 = {4., 5.};

//codesnippet structreturn
  v3 = vector_add(v1, v2);
  cout << "Added: " <<
    v3.x << "," << v3.y << endl;
//codesnippet end

  return 0;
}
```

Chapter 10

Classes and objects

10.1 What is an object?

You learned about `structs` (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:

Definition of object

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 an object, first ask yourself: ‘what functionality should this support’.

Objects and classes

An object is a particular instance of a *class* of similar objects, so you need a *class definition*, after which you define objects of that class.

Object functionality

Small illustration: vector objects.

Code:

```
Vector v(1.,2.); // make vector (1,2)
cout << "vector has length "
     << v.length() << endl;
v.scaleby(2.);
cout << "vector has length "
     << v.length() << endl
     << "and angle " << v.angle()
     << endl;
```

Output

[object] functionality:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section 10.6.1

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?

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 Vector` shows the constructor and the data members:

```
|| class Vector {
|| private:
||     double x, y;
|| public:
||     Vector( double x, double y )
||         : x(x), y(y) {};
```

Constructor and initialization

- 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.

Initialization

```
|| Vector( float x, float y )
||     : x(x), y(y) {};
```

or

```
|| Vector( float xx, float yy ) {
||     x = xx; y = yy; };
```

10.1.2 Methods

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

Class methods

We used methods `length` and `scaleby`. These are defined inside the class:

```
|| void scaleby( double a ) {
||     x *= a; y *= a; };
|| double length() {
||     return sqrt(x*x + y*y); };
|| double angle() {
||     return atan(y/x);
|| };
|| };
```

- They 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:

```
|| Vector vec(5,12);
|| double s = vec.length();
```

10.1.3 Initialization

Member default values

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.

Member initializer lists

Other syntax for initialization:

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

Member initialization in the constructor

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

Advantage of initializer list

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:

make[5]: *** No rule to make target `run_p

For the source of this example, see section 10.6.2

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

Initializer lists

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 Methods

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

Code:

```
class Vector {  
private:  
    double vx,vy;  
public:  
    Vector( double x,double y ) {  
        vx = x; vy = y;  
    };  
    double length() {  
        return sqrt(vx*vx + vy*vy); };  
    double angle() {  
        return 0.; /* something trig */; };  
};  
  
int main() {  
    Vector p1(1.,2.);  
    cout << "p1 has length "  
        << p1.length() << endl;
```

Output

[geom] pointfunc:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section 10.6.3

Exercise 10.2. Add methods `print` and `angle` to the `Vector` class.

How many parameters do they need?

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.

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 Vector {
    /* ... */
    void scaleby( double a ) {
        vx *= a; vy *= a; }
    /* ... */
};

/* ... */
Vector p1(1.,2.);
cout << "p1 has length "
    << p1.length() << endl;
p1.scaleby(2.);
cout << "p1 has length "
    << p1.length() << endl;
```

Output**[geom] pointscaleby:**

make[5]: *** No rule to make target `run_p

For the source of this example, see section [10.6.4](#)

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 Vector {
    /* ... */
    Vector scale( double a ) {
        return Vector( vx*a, vy*a ); }
    /* ... */
};

/* ... */
cout << "p1 has length "
    << p1.length() << endl;
Vector p2 = p1.scale(2.);
cout << "p2 has length "
    << p2.length() << endl;
```

Output**[geom] pointscale:**

make[5]: *** No rule to make target `run_p

For the source of this example, see section [10.6.5](#)

10.1.5 Default constructor

One of the more powerful ideas in C++ is that there can be more than one constructor. You will often be faced with this whether you want or not. The following code looks plausible:

```
Vector v1(1.,2.), v2;
cout << "v1 has length " << v1.length() << endl;
v2 = v1.scale(2.);
cout << "v2 has length " << v2.length() << endl;
```

but it will give an error message during compilation. The reason is that

```
Vector v2;
```

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:

```
|| Vector() {};
|| Vector( double x, double y )
  : x(x), y(y) {};
```

10.1.6 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 your 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.

Decouple internals from externals

```
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.

Public versus private

- Implementation: data members, keep `private`,
- Interface: `public` functions to get/set data.
- Protect yourself against inadvertent changes of object data.
- Possible to change implementation without rewriting calling code.

You should not write access functions lightly: you should first think about what elements of your class should conceptually be inspectable or changeable by the outside world. Consider for example a class where a certain relation holds between members. In that case only changes are allowed that maintain that relation.

Private access gone wrong

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

```

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.

10.1.7 Examples

Classes for abstract objects

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:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section [10.6.6](#)

Exercise 10.3.

- 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 10.4. If you are doing the prime project (chapter [40](#)), now is a good time to do exercise [40.8](#).

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 Accessors and other methods

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.

Exercise 10.5. If you are doing the geometry project, this is a good time to do the exercises in section 41.3.

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).

```
||| 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) {};
|||};
```

10.3.1 Methods of base and derived classes

Overriding methods

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

```

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

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

```

Exercise 10.6. If you are doing the geometry project, you can now do the exercises in section 41.4.

10.3.2 Virtual methods

Base vs derived 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.

Abstract classes

Special syntax for *abstract method*:

```

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

class Deriv {
public:
    virtual void f() { ... };
};

```

Example: using virtual class

```

class VirtualVector {
private:
public:
    virtual void setlinear(float) = DenseVector v(5);
    virtual float operator[] (int) = v[3];
};

Suppose DenseVector derives from
VirtualVector:
DenseVector v(5);
v.setlinear(7.2);
cout << v[3] << endl;

```

Implementation

```

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);
    };
}

```

10.3.3 Advanced topics in inheritance

More

- Multiple inheritance: an X is-a A, but also is-a B.
This mechanism is somewhat dangerous.
- Virtual base class: you don't actually define a function in the base class, you only say 'any derived class has to define this function'.
- Friend classes:

```

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

```

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

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

Direct alteration of internals

Return a reference to a private member:

```

class Vector {
private:
    double vx, vy;
public:
    double &x() { return vx; };
};
int main() {
    Vector v;
    v.x() = 3.1;
}

```

Reference to internals

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?

Setting members through accessor

Use a single accessor for getting and setting:

Code:

```

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

int main() {
    SomeObject myobject(1.);
    cout << "Object member initially :"
        << myobject.xvalue() << endl;
    myobject.xvalue() = 3.;
    cout << "Object member updated :"
        << myobject.xvalue() << endl;
}

```

Output**[object] accessref:**

make[5]: *** No rule to make target `run_a

For the source of this example, see section [10.6.7](#)

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 10.7. This is a good time to do the exercises in section [41.2](#).

10.4.3 Accessibility

Access levels

Methods and data can be

- private, because they are only used internally;
- public, because they should be usable from outside a class object, for instance in the main program;
- protected, because they should be usable in derived classes (see section [10.3.1](#)).

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

Operator overloading

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

For instance:

```
|| class Point {
| private:
|   float x,y;
| public:
|   Point operator*(float factor) {
|     return Point(factor*x,factor*y);
|   };
| };
```

Can even redefine equals and parentheses.

Exercise 10.8. 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:

make[5]: *** No rule to make target 'run_f

For the source of this example, see section ??

Exercise 10.9. 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);
vector<int> ints{5,6,7,8};
for_each
  (ints.begin(),ints.end(),
  printx2);
```

Output

[object] functor2:

make[5]: *** No rule to make target 'run_f

(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:

make[5]: *** No rule to make target 'run_c

For the source of this example, see section [10.6.8](#)

Copying is recursive

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:

make[5]: *** No rule to make target 'run_c

For the source of this example, see section [10.6.9](#)

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:

```
make[5]: *** No rule to make target 'run_d
```

For the source of this example, see section [10.6.10](#)

Exercise 10.10. 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:

```
make[5]: *** No rule to make target 'run_d
```

Destructors and exceptions

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:**

make[5]: *** No rule to make target 'run_e

For the source of this example, see section [10.6.11](#)

10.4.9 ‘this’ pointer*‘this’ pointer to the current object*

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

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

‘this’ use

This is not often needed. Typical use case: 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);
}
```

10.4.10 Static members*Static class members*

A static member acts like shared between all objects.

```
class MyClass {
private:
    static int object_count;
public:
    MyClass() { object_count++; }
}
```

Initialization has to be done elsewhere:

```
|| MyClass::object_count = 0;
```

10.5 Review question

Exercise 10.11. 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

Exercise 10.12. Describe various ways to initialize the members of an object.

10.6 Sources used in this chapter

10.6.1 Listing of code/object/functionality

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** functionality.cxx : illustrating object functionality
*****
*****
```

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

#include <cmath>
using std::sqrt;

//codesnippet functionalityconstruct
class Vector {
private:
    double x,y;
public:
    Vector( double x,double y )
```

```

        : x(x),y(y) {};
//codesnippet end
//codesnippet functionalitymethod
void scaleby( double a ) {
    x *= a; y *= a; }
double length() {
    return sqrt(x*x + y*y); }
double angle() {
    return atan(y/x);
};
};

//codesnippet end

int main() {
    //codesnippet functionality
    Vector v(1.,2.); // make vector (1,2)
    cout << "vector has length "
        << v.length() << endl;
    v.scaleby(2.);
    cout << "vector has length "
        << v.length() << endl
        << "and angle " << v.angle()
        << endl;
    //codesnippet end

    //codesnippet functionalitymethoduse
    Vector vec(5,12);
    double s = vec.length();
    //codesnippet end

    return 0;
}
}

```

10.6.2 Listing of code/geom/pointinitxy

```

/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** pointinit.cxx : about object initialization
*****
*/
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//codesnippet classpointinitxy
class Vector {
private:
    double x,y;
public:
    Vector( double x,double y ) : x(x),y(y) {
    }
}

```

```
//codesnippet end
    double getx() { return x; }
    double gety() { return y; }
};

int main() {
//codesnippet classpointinitxy
    Vector p1(1.,2.);
    cout << "p1 = "
        << p1.getx() << "," << p1.gety()
        << endl;
//codesnippet end

    return 0;
}
```

10.6.3 Listing of code/geom/pointfunc

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** pointfun.cxx : class with method
*****
*****
```

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

#include <cmath>
using std::sqrt;

//codesnippet pointfunc
class Vector {
private:
    double vx, vy;
public:
    Vector( double x, double y ) {
        vx = x; vy = y;
    };
    double length() {
        return sqrt(vx*vx + vy*vy);
    }
    double angle() {
        return 0.; /* something trig */;
    };
};

int main() {
    Vector p1(1.,2.);
    cout << "p1 has length "
        << p1.length() << endl;
//codesnippet end

    return 0;
}
```

```
|| }
```

10.6.4 Listing of code/geom/pointscaleby

```
|||| ****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
**** pointscaleby.hxx : method that operates on members
**** ****
|||| ****
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <cmath>
using std::sqrt;

//codesnippet pointscaleby
class Vector {
//codesnippet end
private:
    double vx, vy;
public:
    Vector( double x, double y ) {
        vx = x; vy = y;
    };
//codesnippet pointscaleby
    void scaleby( double a ) {
        vx *= a; vy *= a; };
//codesnippet end
    double length() {
        return sqrt(vx*vx + vy*vy); };
//codesnippet pointscaleby
};
//codesnippet end

int main() {
//codesnippet pointscaleby
    Vector p1(1.,2.);
    cout << "p1 has length "
        << p1.length() << endl;
    p1.scaleby(2.);
    cout << "p1 has length "
        << p1.length() << endl;
//codesnippet end

    return 0;
}
```

10.6.5 Listing of code/geom/pointscale

```
|| / ****
****
```

10.6.6 Listing of code/object/stream

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** stream.hxx : simulate an integer stream
```

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
***** accessref.cxx : method returning reference
*****



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

//codesnippet integerstream
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;
    //codesnippet end
    return 0;
}
```

10.6.7 Listing of code/object/accessref

```
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
***** accessref.cxx : method returning reference
*****



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

//codesnippet objaccessref
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;
//codesnippet end

    return 0;
}

```

10.6.8 Listing of code/object/copyscalar

```

//****************************************************************************
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
****
**** copyscalar.cxx : copy constructor with a simple class
****

//****************************************************************************

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

//codesnippet classwithcopy
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; };
};

//codesnippet end

int main() {

    //codesnippet classwithcopyuse
    has_int an_int(5);
    has_int other_int(an_int);
    an_int.printme();
    other_int.printme();
//codesnippet end

    return 0;
}

```

10.6.9 Listing of code/object/copyvector

```
|| ****
```

```

***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** copyvector.cxx : copy constructor with a class containing vector
*****
***** ***** ***** ***** ***** ***** ***** ***** ***** ***** ***** ***** /


#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <vector>
using std::vector;

//codesnippet classwithcopyvector
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; }
};

//codesnippet end

int main() {

    //codesnippet classwithcopyvectoruse
    has_vector a_vector(5);
    has_vector other_vector(a_vector);
    a_vector.set(3);
    a_vector.printme();
    other_vector.printme();
    //codesnippet end

    return 0;
}

```

10.6.10 Listing of code/object/destructor

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** destructor.cxx : illustration of objects going out of scope
*****
*****
```

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

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

int main() {

//examplesnippet destructoruse
    cout << "Before the nested scope"
    << endl;
{
    SomeObject obj;
    cout << "Inside the nested scope"
    << endl;
}
    cout << "After the nested scope"
    << endl;
//examplesnippet end

    return 0;
}
```

10.6.11 Listing of code/object/exceptdestruct

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** exceptdestruct.cxx : an exception calls the destructor
*****
*****
```

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

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

```
        << endl; } ;  
};  
//examplesnippet end  
  
int main() {  
  
//examplesnippet exceptdestruct  
    try {  
        SomeObject obj;  
        cout << "Inside the nested scope" << endl;  
        throw(1);  
    } catch (...) {  
        cout << "Exception caught" << endl;  
    }  
//examplesnippet end  
  
    return 0;  
}
```


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, object. 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.

Vector definition

Definition, mostly without 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` is up to you, and
- `size` is the (initial size of the array). This is an integer, or more precisely, a `size_t` parameter.
- `init_value` will be used for all elements.

11.1.2 Element access

The simplest way to access vector elements is with the square bracket notation:

```
|| x[1] = 3.14;  
|| cout << x[2];
```

This gives very fast access, but there is no *checking* on whether the index is within the *array bounds*. Accessing an element outside the bounds may abort your code, typically with a *segmentation fault*, but your code may just as well proceed, using invalid data.

As you see in this example, if `a` is an array, and `i` an integer, then `a.at(i)` is the `i`'th element.

- An array element `a.at(i)` can be used to get the value of an array element, or it can occur in the left-hand side of an assignment to set the value.
- The *array index* (or *array subscript*) `i` starts numbering at zero.
- Therefore, if an array has n elements, its last element has index $n-1$.
- If you try to get an array elements outside the bounds of the array, the compiler will only detect this in simple cases. More likely, your program may give a runtime error, but that does not necessarily happen. You could just get some random value.

There is a safer way to access elements:

```
|| x.at(1) = 3.14;  
|| cout << x.at(2);
```

This is slightly slower, but it does perform bounds checking for every access.

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:

Vector constant initialization

There is a syntax for initializing a vector with a constant:

```
|| vector<float> x(25, 3.15);
```

which gives a vector 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*:

Vector initialization

You can initialize a vector as a whole:

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:**

make[5]: *** No rule to make target `run_d

For the source of this example, see section [11.10.1](#)

11.2 Ranging over an array

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

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

Range over elements

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

```

    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:**

make[5]: *** No rule to make target `run_d

For the source of this example, see section [11.10.2](#)

Ranging over a vector

```

    for ( auto e : my_vector)
        cout << e;

```

(You can spell out the type of the array 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:

Range over vector denotation

Code:

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

Output

[array] rangedenote:

make[5]: *** No rule to make target 'run_r...

For the source of this example, see section 11.10.3

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

Indexing the elements

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 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:

make[5]: *** No rule to make target 'run_v...

For the source of this example, see section ??

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 an array. Use:

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

Which mechanism do you use for traversing the array?

Hint:

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

Exercise 11.3. Find the location of the first negative element in an array.

Which mechanism do you use?

Exercise 11.4. Check whether an array is sorted.

Range over elements by reference

Range-based loop indexing makes a copy of the array element. If you want to alter the array, 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:**

make[5]: *** No rule to make target `run_v...

For the source of this example, see section 11.10.4

Exercise 11.5. If you do the prime numbers project, you can now do exercise 40.8.1.

*Indexing with pre/post increment*Array indexing in **while** loop and such:

```

    || x = a[i++]; /* is */ x = a.at(i); i++;
    y = b[++i]; /* is */ i++; y = b.at(i);

```

*Example of increment indexing***Code:**

```

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

```

Output**[loop] plusplus:**

make[5]: *** No rule to make target `run_p...

*For the source of this example, see section ??*Exercise: modify this so that after the while loop **index** is the number of leading odd elements.

11.3 Vector are a class

You wouldn't tell it from the above examples, but vectors actually form a **vector** class. The angle bracket notation means that we have a class that is parametrized with the type (see chapter 21 for the details), and you can have vectors of ints, vectors of chars, et cetera. We can now say that **vector**<**int**> is a type, pronounced 'vector-of-int', and you can make new variables of that type.

Vector copy

Vectors can be copied just like other datatypes:

Code:

```

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

```

Output**[array] vectorcopy:**

make[5]: *** No rule to make target `run_v...

For the source of this example, see section 11.10.5

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 11.6. 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 be 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:

Dynamic extension

Extend with `push_back`:

Code:

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

Output

[array] vectorend:

make[5]: *** No rule to make target 'run_v...

For the source of this example, see section [11.10.11](#)

also `pop_back`, `insert`, `erase`.

Flexibility comes with a price.

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

Dynamic size extending

```
|| 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

Vector as function argument

You can pass a vector to a function:

```
void print0( vector<double> v ) {
    cout << v.at(0) << endl;
};
```

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

Vector pass by value example

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:

make[5]: *** No rule to make target 'run_v

For the source of this example, see section [11.10.7](#)

Exercise 11.7. Revisit exercise [11.6](#) and introduce a function for computing the L_2 norm.

Vector pass by reference

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

Code:

```

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:

make[5]: *** No rule to make target 'run_v...

For the source of this example, see section 11.10.8

Exercise 11.8. 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.

The alternative would in-place sorting. Find arguments for/against that approach.

(See section 7.8.1 for the random function.)

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

Vector as function return

You can have a vector as return type of a function. 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:

make[5]: *** No rule to make target 'run_v...

For the source of this example, see section 11.10.9

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 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 an object that contains a vector, where the size of the vector is passed to the constructor. Since the class definition is an abstract definition of all the object, clearly you can not have the array size there.

```

class witharray {
private:
    vector<int> the_array(????? );
public:
    witharray( int n ) {
        the_array(????? n ????? );
    }
}

```

The solution is to specify a vector without size in the class definition, which creates a vector of size zero. When you create an object, you then create a vector of the right size, and use that as the vector member of the object.

Create and assign

The following mechanism works:

```

class witharray {
private:
    vector<int> the_array;
public:
    witharray( int n )
        : the_array(vector<int>(n)) {
    };
}

```

Better than

```

witharray( int n ) {
    the_array = vector<int>(n);
}

```

The two cases behave slightly differently.

- The expression `vector<int>(n)` creates an anonymous vector of size `n`;
 - In the first case, that anonymous vector is moved to become the value of the `the_array` variable;
 - in the second case, the object is first created with `the_array` being an empty vector, and the size-`n` is then written over it. This is less efficient and less elegant.
- Either way, now you have an object with a vector of size `n` internally.

11.5.1 Timing

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

Vector extension

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;
```

Vector extension

With subscript:

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

You can also use `new` to allocate (see section 16.6.2):

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

Timings are partly predictable, partly surprising:

Timing

```
||  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 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 vectors

Multi-dimensional is harder with vectors:

```

vector<float> row(20);
vector<vector<float>> rows(10, row);

```

Create a row vector, then store 10 copies of that:
vector of vectors.

This is not the best implementation of a matrix, for instance because the elements are not contiguous. However, let's continue with it for a moment.

Matrix class

```

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

Exercise 11.12. 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[i*n+j] = v;
    };
    double get(int i,int j) {
        return the_matrix[i*n+j];
    };
/* ... */
};
```

Exercise 11.13. Why are `m`, `n` here 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 11.14. 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> array(5, 2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
```

Output

[array] vectorend:

make[5]: *** No rule to make target `run_v

For the source of this example, see section 11.10.11

Code:

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

Output

[array] vectorenditerator:

make[5]: *** No rule to make target `run_v

For the source of this example, see section 11.10.11

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:

make[5]: *** No rule to make target `run_s

For the source of this example, see section [11.10.12](#)

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:

make[5]: *** No rule to make target `run_r

For the source of this example, see section [11.10.13](#)

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 11.15. 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);

```

C/C++ row major

(1,1)	(1,2)	→
(2,1)		→
(3,1)		

Physical:

(1,1) (1,2) ... (2,1) ... (3,1)

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

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 when control leaves the scope.
- Dynamically created objects, such as the target of a pointer, live on the heap because their lifetime is not subject to scope.

The existence of the second category is a source of *memory leaks* in languages such as C. This danger is greatly lessened in C++.

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

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

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 **GSL!** (*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.

11.9 Exercises

Exercise 11.16. 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 11.17. Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

Pascal's triangle

Pascal's triangle contains binomial coefficients:

where

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

The coefficients can easily be computed from the recurrence

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

Exercise 11.18.

- 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 pattern consists of a central asterisk surrounded by four rows of asterisks. The first row has three asterisks on each side of the center. The second row has five asterisks on each side. The third row has seven asterisks on each side. The fourth row has nine asterisks on each side. The pattern is symmetric about both horizontal and vertical axes.

- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.

Exercise 11.19. Extend the Pascal exercise:

Optimize your code to use precisely enough space for the coefficients.

Exercise 11.20. 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 11.21. Put eight queens on a chessboard so that none threatens any other.

Exercise 11.22. From the ‘Keeping it REAL’ book, exercise 3.6 about Markov chains.

11.10 Sources used in this chapter

11.10.1 Listing of code/array/dynamicinit

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2018/9 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** dynamicinit.cxx : initialization of vectors  
*****  
*****
```

```
#include <iostream>  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
int main() {  
  
    //codesnippet dynamicinit  
    {  
        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;  
    }  
    //codesnippet end  
  
    return 0;  
}
```

11.10.2 Listing of code/array/dynamicmax

```
*****  
***** This file belongs with the course
```

```
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*** 
**** dynamicmax.cxx : static array length examples
**** 
***** **** **** **** **** **** **** **** **** **** **** **** **** **** /


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

#include <vector>
using std::vector;

int main() {

    //examplesnippet dynamicmax
    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;
    //examplesnippet end

    return 0;
}
```

11.10.3 Listing of code/array/rangedenote

```
/***** **** **** **** **** **** **** **** **** **** **** **** **** **** /


**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*** 
**** rangedenote.cxx : range over denotation
**** 
***** **** **** **** **** **** **** **** **** **** **** **** **** /


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

#include <vector>
using std::vector;

int main() {

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

```
    }  
    return 0;  
}
```

11.10.4 Listing of code/array/vectorrangeref

```
/*********************************************  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** vectorrangeref.cxx : range-based indexing by reference  
****  
*****  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
int main() {  
  
    //codesnippet vectorrangeref  
    vector<float> myvector  
        = {1.1, 2.2, 3.3};  
    for ( auto &e : myvector )  
        e *= 2;  
    cout << myvector.at(2) << endl;  
    //codesnippet end  
  
    return 0;  
}
```

11.10.5 Listing of code/array/vectorcopy

```
/*********************************************  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** vectorcopy.cxx : example of vector copying  
****  
*****  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;
```

```
int main() {  
    //codesnippet vectorcopy  
    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;  
    //codesnippet end  
  
    return 0;  
}
```

11.10.6 Listing of code/array/vectorend

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** vectorend.cxx : example of vector end iterator  
*****  
*****  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
int main() {  
  
    cout << "End Bracket" << endl;  
    {  
        //codesnippet vectorpush  
        vector<int> array(5,2);  
        array.push_back(35);  
        cout << array.size() << endl;  
        cout << array[array.size()-1] << endl;  
        //codesnippet end  
    }  
    cout << "... bracket" << endl;  
  
    cout << "End Iterator" << endl;  
    {  
        //codesnippet vectorpushiterator  
        vector<int> array(5,2);  
        array.push_back(35);  
        cout << array.size() << endl;  
        cout << array[array.size()-1] << endl;  
        cout << *(--array.end()) << endl;  
        //codesnippet end  
    }  
}
```

```
    cout << "... iterator" << endl;
}
} }
```

11.10.7 Listing of code/array/vectorpassnot

```
/*
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorpassnotcxx : example of vector passed by value
 ****
 ****/
```

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

#include <vector>
using std::vector;

//codesnippet vectorpassval
void set0
( vector<float> v, float x )
{
    v.at(0) = x;
}
//codesnippet end

int main() {

    //codesnippet vectorpassval
    vector<float> v(1);
    v.at(0) = 3.5;
    set0(v, 4.6);
    cout << v.at(0) << endl;
    //codesnippet end

    return 0;
}
```

11.10.8 Listing of code/array/vectorpassref

```
/*
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorpassrefcxx : example of vector passed by reference
 ****
 ****/
```

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

#include <vector>
using std::vector;

//codesnippet vectorpassref
void set0
( vector<float> &v, float x )
{
    v.at(0) = x;
}
//codesnippet end

int main() {

    //codesnippet vectorpassref
    vector<float> v(1);
    v.at(0) = 3.5;
    set0(v, 4.6);
    cout << v.at(0) << endl;
    //codesnippet end

    return 0;
}
```

11.10.9 Listing of code/array/vectorreturn

```
/*
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-9 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorreturn.cxx : return vector from function
 ****
 ****
 */

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

#include <vector>
using std::vector;

//codesnippet vectorreturn
vector<int> make_vector(int n) {
    vector<int> x(n);
    x.at(0) = n;
    return x;
}
//codesnippet end

int main() {
```

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

return 0;
}
```

11.10.10 Listing of code/array/vectorend

```
/*
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorend.cxx : example of vector end iterator
 ****
 ****
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

    cout << "End Bracket" << endl;
{
    //codesnippet vectorpush
    vector<int> array(5,2);
    array.push_back(35);
    cout << array.size() << endl;
    cout << array[array.size()-1] << endl;
    //codesnippet end
}
    cout << "... bracket" << endl;

    cout << "End Iterator" << endl;
{
    //codesnippet vectorpushiterator
    vector<int> array(5,2);
    array.push_back(35);
    cout << array.size() << endl;
    cout << array[array.size()-1] << endl;
    cout << *( --array.end() ) << endl;
    //codesnippet end
}
    cout << "... iterator" << endl;

    return 0;
}
```

```
|| }
```

11.10.11 Listing of code/array/vectorend

```
*****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** vectorend.cxx : example of vector end iterator  
****  
*****  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
int main() {  
  
    cout << "End Bracket" << endl;  
    {  
        //codesnippet vectorpush  
        vector<int> array(5,2);  
        array.push_back(35);  
        cout << array.size() << endl;  
        cout << array[array.size()-1] << endl;  
        //codesnippet end  
    }  
    cout << "... bracket" << endl;  
  
    cout << "End Iterator" << endl;  
    {  
        //codesnippet vectorpushiterator  
        vector<int> array(5,2);  
        array.push_back(35);  
        cout << array.size() << endl;  
        cout << array[array.size()-1] << endl;  
        cout << *( --array.end() ) << endl;  
        //codesnippet end  
    }  
    cout << "... iterator" << endl;  
  
    return 0;  
}
```

11.10.12 Listing of code/array/staticinit

```
*****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
```

```
*****
**** staticinit.cxx : initialization of static arrays
*****
***** ***** ***** ***** ***** ***** ***** ***** *****/
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

    //codesnippet arrayinit
    {
        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;
    }
    //codesnippet end

    return 0;
}
```

11.10.13 Listing of code/array/rangemax

```
/***** ****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*** 
**** rangemax.cxx : static array length examples
**** 
***** ***** ***** ***** ***** ***** ***** ***** *****/
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

    //examplesnippet rangemax
    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;
//examplesnippet end

return 0;
}
```


Chapter 12

Strings

12.1 Characters

Characters and ints

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

Char / int equivalence

Equivalent to (short) integer:

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:

make[5]: *** No rule to make target 'run_i

For the source of this example, see section [12.5.1](#)

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

Exercise 12.1. Write a program that accepts an integer $1 \dots 26$ 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

String declaration

```
||| #include <string>
using std::string;
// .. and now you can use 'string'
```

(Do not use the C legacy mechanisms.)

String creation

A *string* variable contains a string of characters.

```
|| 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?

Quotes in strings

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**:

```
make[5]: *** No rule to make target 'run_q'
```

For the source of this example, see section 12.5.2

Concatenation

Strings can be concatenated:

```
|| txt = txt1+txt2;
|| txt += txt3;
```

String indexing

You can query the *size*:

```
|| int txtlen = txt.size();
```

or use subscripts:

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

Ranging over a string

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**:

```
make[5]: *** No rule to make target 'run_s'
```

For the source of this example, see section 12.5.5

New syntax: range-based for

Code:

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

Output

[string] stringrange:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section [12.5.5](#)

Range with reference

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:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section [12.5.5](#)

Iterating over a string

```
|| 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:

$$s \equiv 3: "acd" \Rightarrow "dfg".$$

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?

More vector methods

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

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 Conversion

to_string

12.4 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.

12.5 Sources used in this chapter

12.5.1 Listing of code/string/intchar

```
/*
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018/9 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** intchar.hxx : int/char equivalence
 ****
 ****
 #include <iostream>
 using std::cin;
 using std::cout;
 using std::endl;
 #include <iomanip>
 using std::setw;

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

```

#include <string>
using std::string;

int main() {

//codesnippet intchar
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;
//codesnippet end

return 0;
}

```

12.5.2 Listing of code/string/quote

```

/*********************************************
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** quote.cxx : quote handling
*****
*****************************************/
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <string>
using std::string;

int main() {

//codesnippet quotestring
string
    one("a b c"),
    two("a \"b\" c"),
    three( R"(a ""b """c)" );
cout << one << endl;
cout << two << endl;
cout << three << endl;
//codesnippet end

return 0;
}

```

12.5.3 Listing of code/string/stringrange

```

|| / ****

```

12.5.4 Listing of code/string/stringrange

```
/*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****
```

```
**** stringrange.cxx : range over string
*****
*****  
  
#include <iostream>
using std::cin;
using std::cout;
using std::endl;  
  
#include <string>
using std::string;  
  
int main() {  
  
    cout << "Index" << endl;
    //codesnippet stringindex
    string abc = "abc";
    cout << "By character: ";
    for (int ic=0; ic<abc.size(); ic++)
        cout << abc[ic] << " ";
    cout << endl;
    //codesnippet end
    cout << ".. index" << endl;  
  
    cout << "Range" << endl;
    //codesnippet stringrange
    cout << "By character: ";
    for ( char c : abc )
        cout << c << " ";
    cout << endl;
    //codesnippet end
    cout << ".. range" << endl;  
  
    cout << "Set" << endl;
    //codesnippet stringrangeset
    for ( char &c : abc )
        c += 1;
    cout << "Shifted: " << abc << endl;
    //codesnippet end
    cout << ".. set" << endl;  
  
    return 0;
}
```

12.5.5 Listing of code/string/stringrange

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** stringrange.cxx : range over string  
*****  
*****  
*****
```

```
using std::cin;
using std::cout;
using std::endl;

#include <string>
using std::string;

int main() {

    cout << "Index" << endl;
    //codesnippet stringindex
    string abc = "abc";
    cout << "By character: ";
    for (int ic=0; ic<abc.size(); ic++)
        cout << abc[ic] << " ";
    cout << endl;
    //codesnippet end
    cout << ".. index" << endl;

    cout << "Range" << endl;
    //codesnippet stringrange
    cout << "By character: ";
    for ( char c : abc )
        cout << c << " ";
    cout << endl;
    //codesnippet end
    cout << ".. range" << endl;

    cout << "Set" << endl;
    //codesnippet stringrangeset
    for ( char &c : abc )
        c += 1;
    cout << "Shifted: " << abc << endl;
    //codesnippet end
    cout << ".. set" << endl;

    return 0;
}
```

Chapter 13

Input/output

13.1 Formatted output

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:

```
make[5]: *** No rule to make target 'run_c...
```

For the source of this example, see section [13.6.1](#)

Reserve space

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

Code:

```
|| cout << "Width is 6:" << endl;
    for (int i=1; i<200000000; i*=10)
        cout << "Number: "
            << setw(6) << i << endl;
    cout << endl;
    cout << "Width is 6:" << endl;
    cout << "."
        << setw(6) << 1 << 2 << 3 << endl;
    cout << endl;
```

Output

[io] width:

```
make[5]: *** No rule to make target 'run_w...
```

For the source of this example, see section [13.6.2](#)

Padding character

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

13. Input/output

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:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section [13.6.3](#)

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

Left alignment

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:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section [13.6.4](#)

Number base

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:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section [13.6.5](#)

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
```

Exercise 13.2. Use integer output to print real numbers aligned on the decimal:

```
1.345
```

```
23.789
456.1234
```

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

Hexadecimal

Hex output is useful for pointers (chapter 15):

Code:

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

Output

[pointer] coutpoint:

```
make[5]: *** No rule to make target 'run_c
```

For the source of this example, see section 16.8.2

Back to decimal:

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

13.1.1 Floating point output

Floating point precision

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:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section 13.6.7

(Notice the rounding)

Fixed point precision

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:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section 13.6.8

Aligned fixed point output

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:

make[5]: *** No rule to make target 'run_a

For the source of this example, see section [13.6.9](#)

Scientific notation

```
cout << "Combine width and precision:" << endl;
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4) << x << endl;
    x *= 10;
}
```

Output

```
Combine width and precision:
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

Text output to file

Streams are general: work the same for console out and file out.

```
#include <fstream>
```

Use:

```
#include <fstream>
using std::ofstream;
/* ...
ofstream file_out;
file_out.open("fio_example.out");
/* ...
file_out << number << endl;
file_out.close();
```

Binary output

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

13.2.1 Binary output

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:

make[5]: *** No rule to make target `run_f

For the source of this example, see section [13.6.10](#)

Code:

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

Output

[io] fiobin:

make[5]: *** No rule to make target `run_f

For the source of this example, see section [13.6.11](#)

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.

Redefine less-less

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

Better terminal input

It is better to use `getline`. This returns a string, rather than a value, so you need to convert it with the following bit of magic:

```
#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

File input streams

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;
}
```

End of file test

There are several ways of testing for the end of a file

- For text files, the `getline` function returns `false` if no line can be read.
- The `eof` function can be used after you have done a read.
- `EOF` is a return code of some library functions; it is not true that a file ends with an EOT character. Likewise you can not assume a Control-D or Control-Z at the end of the file.

Exercise 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 think that `getline` always returns a `bool`, but that's not true. It actually returns an `ifstream`. However, a conversion operator

13. Input/output

```
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*

13.6 Sources used in this chapter

13.6.1 Listing of code/io/cunformat

```
/*********************************************  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** cunformat.cxx : default formatting  
****  
*****  
#include <iostream>  
using std::cout;  
using std::endl;  
  
int main() {  
  
    //codesnippet cunformat  
    for (int i=1; i<200000000; i*=10)  
        cout << "Number: " << i << endl;  
    cout << endl;  
    //codesnippet end  
  
    return 0;  
}
```

13.6.2 Listing of code/io/width

```
/*********************************************  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** width.cxx : setting output width  
****  
*****  
#include <iostream>  
using std::cout;  
using std::endl;  
#include <iomanip>  
using std::right;
```

```

using std::setbase;
using std::setfill;
using std::setw;

int main() {

    //codesnippet formatwidth6
    cout << "Width is 6:" << endl;
    for (int i=1; i<200000000; i*=10)
        cout << "Number: "
            << setw(6) << i << endl;
    cout << endl;
    cout << "Width is 6:" << endl;
    cout << "."
        << setw(6) << 1 << 2 << 3 << endl;
    cout << endl;
    //codesnippet end

    return 0;
}

```

13.6.3 Listing of code/io/formatpad

```

/*****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** formatpad.cxx : padded io
 ****
 ****
#include <iostream>
using std::cout;
using std::endl;
//codesnippet formatpad
#include <iomanip>
using std::setfill;
using std::setw;
//codesnippet end

int main() {

    //codesnippet formatpad
    for (int i=1; i<200000000; i*=10)
        cout << "Number: "
            << setfill('.') << setw(6) << i
            << endl;
    //codesnippet end

    return 0;
}

```

13.6.4 Listing of code/io/formatleft

```

|| /*****

```

13. Input/output

```
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** formatleft.cxx : left aligned io
*****
***** ***** ***** ***** ***** ***** ***** ***** ***** ***** ***** /



#include <iostream>
using std::cin;
using std::cout;
using std::endl;
//codesnippet formatleft
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
//codesnippet end

int main() {

    //codesnippet formatleft
    for (int i=1; i<200000000; i*=10)
        cout << "Number: "
            << left << setfill('.')
            << setw(6) << i << endl;
    //codesnippet end
    cout << endl;

    return 0;
}
```

13.6.5 Listing of code/io/format16

```
*****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** format16.cxx : base 16 formatted io  
****  
*****  
  
#include <iostream>  
using std::cout;  
using std::endl;  
//codesnippet format16  
#include <iomanip>  
using std::setbase;  
using std::setfill;  
//codesnippet end  
  
int main() {  
  
    //codesnippet format16
```

```

cout << setbase(16) << setfill(' ');
for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
        cout << i*16+j << " ";
    cout << endl;
}
//codesnippet end

return 0;
}

```

13.6.6 Listing of code(pointer/coutpoint

```

/*********************************************
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** coutpoint.cxx : print a pointer
*****
******************************************/

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

int main() {

//codesnippet coutpoint
int i;
cout << "address of i, decimal: "
    << (long)&i << endl;
cout << "address of i, hex      : "
    << std::hex << &i << endl;
//codesnippet end

return 0;
}

```

13.6.7 Listing of code/io/formatfloat

```

/*********************************************
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** formatfloat.cxx : floating point output
*****
******************************************/

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

```

```
//codesnippet formatfloat
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
//codesnippet end

int main()  {

    float x;
    //codesnippet formatfloat
    x = 1.234567;
    for (int i=0; i<10; i++)  {
        cout << setprecision(4) << x << endl;
        x *= 10;
    }
    //codesnippet end

    return 0;
}
```

13.6.8 Listing of code/io/fix

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** fix.cxx : fixed precision io  
*****  
*****  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <iomanip>  
using std::fixed;  
using std::setprecision;  
using std::setw;  
  
int main() {  
  
    double x;  
    //codesnippet fixfrac  
    x = 1.234567;  
    cout << fixed;  
    for (int i=0; i<10; i++) {  
        cout << setprecision(4) << x << endl;  
        x *= 10;  
    }  
    //codesnippet end  
  
    return 0;  
}
```

```
|| }
```

13.6.9 Listing of code/io/align

```
*****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** fix.cxx : fixed precision io  
****  
*****  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <iomanip>  
using std::fixed;  
using std::setprecision;  
using std::setw;  
  
int main() {  
  
    double x;  
    //codesnippet align  
    x = 1.234567;  
    cout << fixed;  
    for (int i=0; i<10; i++) {  
        cout << setw(10) << setprecision(4) << x  
        << endl;  
        x *= 10;  
    }  
    //codesnippet end  
  
    return 0;  
}
```

13.6.10 Listing of code/io/fio

```
*****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2016/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** fio.cxx : file io  
****  
*****  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
#include <iomanip>
```

```
using std::right;
using std::setbase;
using std::setfill;
using std::setw;

//codesnippet fio
#include <fstream>
using std::ofstream;
//codesnippet end

int main() {

    //codesnippet fio
    ofstream file_out;
    file_out.open("fio_example.out");
    //codesnippet end

    int number;
    cout << "A number please: ";
    cin >> number;
    cout << endl;
    //codesnippet fio
    file_out << number << endl;
    file_out.close();
    //codesnippet end
    cout << "Written." << endl;

    return 0;
}
```

13.6.11 Listing of code/io/fiobin

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** fiobin.cxx : binary file io
*****
*****
```

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <iomanip>
using std::right;
using std::setbase;
using std::setfill;
using std::setw;

#include <fstream>
using std::ofstream;
using std::ios;

int main() {
```

```
//codesnippet fiobin
ofstream file_out;
file_out.open
    ("fio_binary.out",ios::binary);
//codesnippet end

int number;
cout << "A number please: ";
cin >> number;
// file_out << number ;
//codesnippet fiobin
file_out.write( (char*)(&number),4);
//codesnippet end
file_out.close();
cout << "Written." << endl;

return 0;
}
```


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 7.5.2. Make sure you study that material first.

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

```
|| void change_scalar(int i) {
    i += 1;
}
/* ... */
number = 3;
cout << "Number is 3: "
    << number << 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.

Reference: change argument

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
```

Reference: save on copying

<pre>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</pre>	<p>Instead write:</p> <pre> void f(BigDude &thing) { }; Prevent changes: void f(const BigDude &thing) { };</pre>
---	--

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:**

make[5]: *** No rule to make target `run_c

For the source of this example, see section [17.5.1](#)

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:

make[5]: *** No rule to make target 'run_r...

For the source of this example, see section [14.6.2](#)

(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 access to arrays and/or vectors that are private to the object.

Suppose you have an object that contains an `int` array. You can return an element by defining the subscript (square bracket) operator for the class:

```
class vector10 {
private:
    int array[10];
public:
    /* ... */
    int operator()(int i) {
        return array[i];
    };
    int operator[](int i) {
        return array[i];
    };
};
```

```

    };
/* ...
vector10 v;
cout << v(3) << 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.

14.6 Sources used in this chapter

14.6.1 Listing of code/const/constref

```

*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** constref.cxx : returning a const by ref
*****
***** */

```

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

//codesnippet constref
class has_int {
private:
    int mine{1};
public:
    const int& int_to_get() { return mine; }
    int& int_to_set() { return mine; }
    void inc() { mine++; }
};

//codesnippet end

int main() {

//codesnippet constref
    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;
//codesnippet end

    return 0;
}
```

14.6.2 Listing of code/func/rhsref

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** rhsref.cxx : result of an expression can not be reference
*****
*****
```

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

//codesnippet rhsrefclass
class myclass {
private:
    int stored{0};
public:
    myclass(int i) : stored(i) {};
    int &data() { return stored; };
};

//codesnippet end
```

```
int main() {  
  
    //codesnippet rhsref  
    myclass obj(5);  
    cout << "object data: "  
        << obj.data() << endl;  
    int dc当地 = obj.data();  
    dc当地++;  
    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;  
    //codesnippet end  
  
    return 0;  
}
```


Chapter 15

Pointers

The term pointer is used to denote a reference to a quantity. This chapter will explain pointers, and give some uses for them.

We remark that there is less need for pointers in C++ than there was in C.

- To pass an argument *by reference*, use a *reference*. Section 7.5.
- Strings are done through `std::string`, not character arrays; see 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 51.1.2.
- Objects on the heap.
- Use `nullptr` as a signal.

15.1 The ‘arrow’ notation

Members from pointer

- 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?

Creating a shared pointer

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;
```

(It is also possible to cast a `new` object to shared pointer, but that method is not recommended.)

Using shared pointers requires at the top of your file:

```
|| #include <memory>
|| using std::shared_ptr;
|| using std::make_shared;
```

Simple example

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:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [15.6.1](#)

Pointers to arrays

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

Linked lists

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; }
    void set_tail( shared_ptr<Node> tail ) {
        tail_ptr = tail; }
};

void print() {
    cout << datavalue;
    if (has_next()) {
        cout << ", " ; tail_ptr->print();
    }
};


```

List usage

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:**

make[5]: *** No rule to make target 'run_s

Linked lists and recursion

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:**

make[5]: *** No rule to make target 'run_a

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:**

make[5]: *** No rule to make target 'run_i

Exercise 15.3. For a more sophisticated approach to linked lists, do the exercises in section 51.1.2.

Exercise 15.4. If you are doing the prime numbers project (chapter 40) you can now do exercise 40.8.2.

15.2.2 Smart pointers versus address pointers

Pointers don't go with addresses

The oldstyle `&y` address pointer can not be made smart:

```
|| auto
  ||| p1 = shared_ptr<HasY>( &y ),
  ||| p2 = shared_ptr<HasY>( &y );
  ||| p1->y = 3;
  ||| cout << "Pointer 2's y: "
  |||     << p2->y << endl;
```

gives:

```
address(56325,0x7fff977cc380) malloc: *** error for object
0x7ffeeb9caf08: pointer being freed was not allocated
```

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.

Reference counting illustrated

We need a class with constructor and destructor tracing:

```
|| class thing {
  || public:
  ||| thing() { cout << "calling constructor\n"; };
  ||| ~thing() { cout << "calling destructor\n"; };
  ||};
```

Pointer overwrite

Let's create a pointer and overwrite it:

Code:

```

cout << "set pointer1"
      << endl;
auto thing_ptr1 =
    shared_ptr<thing>
    ( new thing );
cout << "overwrite pointer"
      << endl;
thing_ptr1 = nullptr;

```

Output**[pointer] ptr1:**

make[5]: *** No rule to make target 'run_p

For the source of this example, see section 15.6.2

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.

*Pointer copy***Code:**

```

cout << "set pointer2" << endl;
auto thing_ptr2 =
    shared_ptr<thing>
    ( new 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:**

make[5]: *** No rule to make target 'run_p

For the source of this example, see section 15.6.3

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.

Getting the underlying 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:

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section [15.6.4](#)

15.4.2 Example: linked lists

The standard example of pointer manipulation is ‘linked lists’. This is discussed in some detail in section [51.1.2](#).

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>
```

15.5.4 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.

15.5.5 Void pointer

The need for *void pointers* 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* know what it points to, use `std::any`, which is usually smart enough to call destructors when needed.

15.5.6 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:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [15.6.6](#)

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:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [15.6.6](#)

15.6 Sources used in this chapter

15.6.1 Listing of code(pointer/pointx)

```
/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** pointx.cxx : access through arrow
*****
*/
#include <iostream>
using std::cout;
using std::endl;

#include <memory>
using std::make_shared;

//codesnippet pointx
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;
//codesnippet end
}
```

15.6.2 Listing of code/pointer/ptr1

```
/*********************************************
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** ptr1.cxx : shared pointers
*****
*****
```

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

#include <memory>
using std::shared_ptr;
using std::make_shared;

//codesnippet thingcall
class thing {
public:
    thing() { cout << ".. calling constructor\n"; };
    ~thing() { cout << ".. calling destructor\n"; };
};

//codesnippet end

int main() {

    //codesnippet shareptr1
    cout << "set pointer1"
        << endl;
    auto thing_ptr1 =
        make_shared<thing>();
    cout << "overwrite pointer"
        << endl;
    thing_ptr1 = nullptr;
    //codesnippet end

#if 0
    // alternatively
    auto thing_ptr1 = shared_ptr<thing>( new thing );
#endif

    return 0;
}
```

15.6.3 Listing of code/pointer/ptr2

```
/*********************************************
*****
**** This file belongs with the course
```

15.6.4 Listing of code/pointer/pointy

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** pointx.cxx : access through arrow
```

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
***** ptrdouble.cxx : shared pointers to scalar
*****



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

#include <memory>
using std::make_shared;

class HasY {
public:
    double y;
    HasY( double y ) : y(y) {};
};

int main() {
    //codesnippet pointy
    auto Y = make_shared<HasY>(5);
    cout << Y->y << endl;
    Y.get()->y = 6;
    cout << ( *Y.get() ).y << endl;
    //codesnippet end
}
```

15.6.5 Listing of code/pointer/ptrdouble

```
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
***** ptrdouble.cxx : shared pointers to scalar
*****



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

#include <memory>
using std::shared_ptr;
using std::make_shared;

int main() {

    cout << "Double" << endl;
    {
        //codesnippet ptrdouble
        // shared pointer to allocated double
        auto array = shared_ptr<double>( new double );
        double *ptr = array.get();
        array.get()[0] = 2.;
        cout << ptr[0] << endl;
        //codesnippet end
    }
}
```

```
    cout << "double" << endl;
    cout << "Init" << endl;
{
    //codesnippet ptrdoubleinit
    // shared pointer to initialized double
    auto array = make_shared<double>(50);
    double *ptr = array.get();
    cout << ptr[0] << endl;
    //codesnippet end
}
cout << "init" << endl;

return 0;
}
```

15.6.6 Listing of code/pointer/ptrdouble

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** ptrdouble.hxx : shared pointers to scalar
*****
***** /*****
```

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

#include <memory>
using std::shared_ptr;
using std::make_shared;

int main() {

    cout << "Double" << endl;
{
    //codesnippet ptrdouble
    // shared pointer to allocated double
    auto array = shared_ptr<double>( new double );
    double *ptr = array.get();
    array.get()[0] = 2.;
    cout << ptr[0] << endl;
    //codesnippet end
}
cout << "double" << endl;
cout << "Init" << endl;
{
    //codesnippet ptrdoubleinit
    // shared pointer to initialized double
    auto array = make_shared<double>(50);
    double *ptr = array.get();
    cout << ptr[0] << endl;
    //codesnippet end
}
```

```
    }
    cout << "init" << endl;
}

return 0;
}
```


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. When possible, references should be used.

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.

Memory addresses

If you have an

```
|| int i;
```

then `&i` is the address of `i`.

An address is a (long) integer, denoting a memory address. Usually it is rendered in *hexadecimal* notation. C style:

Code:

```
|| int i;
   printf("address of i: %ld\n",
          (long)(&i));
   printf(" same in hex: %lx\n",
          (long)(&i));
```

Output

[pointer] printfpoint:

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section [16.8.1](#)

and C++:

Code:

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

Output

[pointer] coutpoint:

make[5]: *** No rule to make target 'run_c...

For the source of this example, see section 16.8.2

You could just print out the address of a variable, which is sometimes useful for debugging. If you want to store the address, you need to create a variable of the appropriate type. This is done by taking a type and affixing a star to it.

Address types

The type of '& i' is `int*`, pronounced ‘int-star’, or more formally: ‘pointer-to-int’.

You can create variables of this type:

```
|| 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.

Dereferencing

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;
```

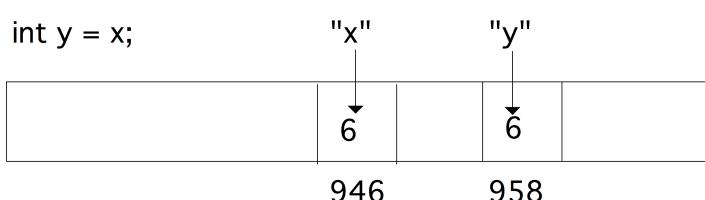
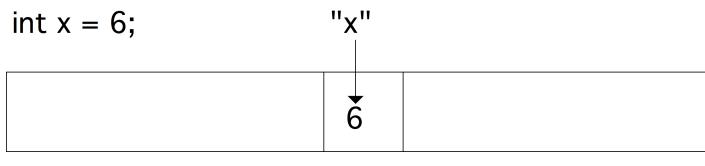
Output

[pointer] cintpointer:

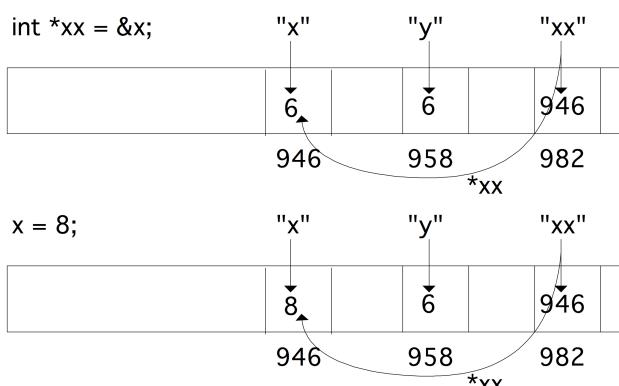
make[5]: *** No rule to make target 'run_c...

For the source of this example, see section 16.8.3

illustration



illustration



- `addr` is the address of `i`.
- You set `i` to 5; nothing changes about `addr`. This has the effect of writing 5 in the memory location of `i`.
- The first `cout` line dereferences `addr`, that is, looks up what is in that memory location.
- Next you change `i` to 6, that is, you write 6 in its memory location.
- The second `cout` looks in the same memory location as before, and now finds 6.

The syntax for declaring a pointer-to-sometype allows for a small variation, which indicates the two way you can interpret such a declaration.

Star stuff

Equivalent:

- `int* addr`: `addr` is an int-star, or
- `int *addr`: `*addr` is an int.

The notion `int* addr` is equivalent to `int *addr`, and semantically they are also the same: you could say that `addr` is an int-star, or you could say that `*addr` is an int.

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 all 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 pointer equivalence

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:

make[5]: *** No rule to make target ‘run_a

For the source of this example, see section [16.8.4](#)

Dynamic allocation

`new` gives a something-star:

```
|| double *x;
|| x = new double[27];
```

(Actually, C uses `malloc`, but that looks similar.)

16.4 Pointer arithmetic

Pointer arithmetic

pointer arithmetic uses the size of the objects it points at:

```
|| double *addr_of_element = array;
|| cout << *addr_of_element;
|| addr_of_element = addr_of_element+1;
|| cout << *addr_of_element;
```

Increment add size of the array element, 4 or 8 bytes, not one!

Exercise 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

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++ pass by reference

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;
  }
```

C-style pass by reference

In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```
|| 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.

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:

Problem with static arrays

```

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:

Declaration and allocation

```

double *array;
if (something) {
    array = new double[25];
} else {
    array = new double[26];
}

```

(Size in doubles, not in bytes as in C)

Memory leak1

```

void func() {
    double *array = new double[large_number];
    // code that uses array
}
int main() {
    func();
}

```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- ⇒ *memory leak*.

Memory leaks

```

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!

De-allocation

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.

Stop using C!

No need for `malloc` or `new`

- Use `std::string` for character arrays, and
- `std::vector` for everything else.

No performance hit if you don't dynamically alter the size.

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:

Allocation in C

```
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.

Allocation in a 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 released, 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.

16.8 Sources used in this chapter

16.8.1 Listing of code/pointer/printfpoint

```
/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** printfpoint.cxx : print a pointer
*****
*****
```

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

int main() {

    //codesnippet printfpoint
    int i;
    printf("address of i: %ld\n",
        (long)(&i));
    printf(" same in hex: %lx\n",
        (long)(&i));
    //codesnippet end

    return 0;
}
```

16.8.2 Listing of code/pointer/coutpoint

```
/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** coutpoint.cxx : print a pointer
*****
*****
```

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

int main() {

    //codesnippet coutpoint
    int i;
    cout << "address of i, decimal: "
        << (long)&i << endl;
    cout << "address of i, hex      : "
        << std::hex << &i << endl;
    //codesnippet end
}
```

```
    }  
    return 0;  
}
```

16.8.3 Listing of code/pointer/cintpointer

```
/*********************************************  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** cintpointer.cxx : oldstyle C pointers  
****  
*****  
/******************************************/  
  
#include <iostream>  
using std::cout;  
using std::endl;  
  
int main() {  
  
    //codesnippet cintpointer  
    int i;  
    int* addr = &i;  
    i = 5;  
    cout << *addr << endl;  
    i = 6;  
    cout << *addr << endl;  
    //codesnippet end  
  
    return 0;  
}
```

16.8.4 Listing of code/pointer/arrayaddr

```
/*********************************************  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** arrayaddr.cxx : pointer to C style array  
****  
*****  
/******************************************/  
  
#include <iostream>  
using std::cout;  
using std::endl;  
  
int main() {  
  
    //codesnippet arrayaddr  
    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;
//codesnippet end
    return 0;
}
```


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

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++;  
|| }
```

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.

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 7.5 to return changed values to the calling environment. The `const` keyword negates that possibility of changing the parameter.

But there is a second reason for using references. Parameters are passed by value, which means that they are copied, and that includes big objects such as `std::vector`. Using a reference to pass a vector is much less costly in both time and space, but then there is the possibility of changes to the vector propagating back to the calling environment.

Consider a class that has methods that return an internal member by reference, once as `const reference` and once not:

Code:

```

class has_int {
private:
    int mine{1};
public:
    const int& int_to_get() { return mine; };
    int& int_to_set() { return mine; };
    void inc() { mine++; };
};

/* ... */
has_int an_int;
an_int.inc(); an_int.inc(); an_int.inc();
cout << "Contained int is now: "
    << an_int.int_to_get() << 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:**

make[5]: *** No rule to make target 'run_c

For the source of this example, see section [17.5.1](#)

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:**

make[5]: *** No rule to make target 'run_c

For the source of this example, see section [17.5.2](#)

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`:

```
class Things {
private:
    int i;
public:
    int get() const { return i; }
    int inc() { return i++; }
}
```

While this is in no way required, it can be helpful in two ways:

- It 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.

17.4 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:**

make[5]: *** No rule to make target 'run_c

For the source of this example, see section [17.5.4](#)

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:**

make[5]: *** No rule to make target 'run_c

For the source of this example, see section [17.5.4](#)

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 Sources used in this chapter

17.5.1 Listing of code/const/constref

```
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
**** constref.cxx : returning a const by ref
 ****
*****
```

```

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

//codesnippet constref
class has_int {
private:
    int mine{1};
public:
    const int& int_to_get() { return mine; }
    int& int_to_set() { return mine; }
    void inc() { mine++; }
};

//codesnippet end

int main() {

//codesnippet constref
    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;
//codesnippet end

    return 0;
}

```

17.5.2 Listing of code/const/constcopy

```

*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** constcopy.cxx : demonstate yes/no copying
*****
***** */

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

class has_int {
private:
    int mine{1};
public:
    has_int(int v) { mine = v; }
    has_int(has_int &other) { cout <<
        "(calling copy constructor)" << endl;
}

```

```

        mine = other.mine;
    };
};

//codesnippet constcopy
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; };
//codesnippet end

int main() {

    has_int an_int(5);

    //codesnippet constcopy
    cout << "Calling f with copy..." << endl;
    f_with_copy(an_int);

    cout << "Calling f with ref..." << endl;
    f_with_ref(an_int);
//codesnippet end

    cout << "... done" << endl;

    return 0;
}

```

17.5.3 Listing of code/const/constpoint

```

/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** constpoint.cxx : pointers and constness
*****
*/
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <memory>
using std::make_shared;
using std::shared_ptr;

//codesnippet constpoint1
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; };
    //codesnippet end
    //codesnippet constpoint2
    auto get_thing_ptr() const {
        return thing_ptr; };
    //codesnippet end
    //codesnippet constpoint3
    // Error: does not compile
    // auto &get_thing_ptr() const {
    auto &access_thing_ptr() {
        return thing_ptr; };
    //codesnippet end
    //codesnippet constpoint4
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; };
    //codesnippet end
};

int main() {

    cout << "Point2" << endl;
    {
        //codesnippet constpoint2
        has_thing container(5);
        container.print();
        container.get_thing_ptr() =
            make_shared<thing>(6);
        container.print();
        //codesnippet end
    }
    cout << "point2" << endl;

    cout << "Point3" << endl;
    {
        //codesnippet constpoint3
        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();  
//codesnippet end  
}  
cout << "point3" << endl;  
  
#if 0  
{  
    //codesnippet constpoint4  
    has_thing constainer(1,2);  
    // Error: does not compile  
    constainer.get_const_ptr()->set_value(9);  
    //codesnippet end  
}  
#endif  
  
return 0;  
}
```

17.5.4 Listing of code/const/constpoint

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** constpoint.cxx : pointers and constness  
*****  
*****  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <memory>  
using std::make_shared;  
using std::shared_ptr;  
  
//codesnippet constpoint1  
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; }
//codesnippet end
//codesnippet constpoint2
auto get_thing_ptr() const {
    return thing_ptr; }
//codesnippet end
//codesnippet constpoint3
// Error: does not compile
// auto &get_thing_ptr() const {
auto &access_thing_ptr() {
    return thing_ptr; }
//codesnippet end
//codesnippet constpoint4
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; }
//codesnippet end
};

int main() {

    cout << "Point2" << endl;
{
    //codesnippet constpoint2
    has_thing container(5);
    container.print();
    container.get_thing_ptr() =
        make_shared<thing>(6);
    container.print();
    //codesnippet end
}
    cout << "point2" << endl;

    cout << "Point3" << endl;
{
    //codesnippet constpoint3
    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();
    //codesnippet end
}
}

```

```
    }
    cout << "point3" << endl;

#ifndef 0
{
    //codesnippet constpoint4
    has_thing constainer(1,2);
    // Error: does not compile
    constainer.get_const_ptr()->set_value(9);
    //codesnippet end
}
#endif

    return 0;
}
```

Chapter 18

Prototypes

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);
```

Prototypes and forward declarations, 1

A first use of prototypes is *forward declaration*.

Some people like defining functions after the main:

<pre> int f(int); int main() { f(5); }; int f(int i) { return i; }</pre>	versus before: <pre> int f(int i) { return i; } int main() { f(5); }</pre>
--	--

Prototypes and forward declarations, 2

You also need forward declaration for mutually recursive functions:

<pre> int f(int); int g(int i) { return f(i); } int f(int i) { return g(i); }</pre>

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.

<pre>// file: def.cxx int tester(float x) { }</pre>	<pre>// file : main.cxx int main() { int t = tester(...); return 0;</pre>
---	--

```
|| }
```

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

Compiling and linking

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile

```
icpc -c yourfile.cc
```

which gives you a file `yourfile.o`, a so-called *object file*; and

2. Then you use the compiler as *linker* to give you the *executable file*:

```
icpc -o yourprogram yourfile.o
```

In this particular example you may wonder what the big deal is. That will become clear if you have multiple source files: now you invoke the compile line for each source, and you link them only once.

Dealing with multiple files

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

At this point, you should learn about the *Make* tool for managing your programming project.

18.1.2 Header files

Even better than writing the prototype every time you need the function is to have a *header file*:

Prototypes and header files

```
// 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 5 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 6 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

Class prototypes

Header file:

```
|| class something {
    public:
        double somedo(vector);
};
```

Implementation file:

```
|| double something::somedo(vector v) {
    .... something with v ....
};
```

Data members in proto

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 impossible: in order to compile the templated definitions the compiler needs to know with what types they will be used.

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
#endif
EXTERN int processnumber
#endif

//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.

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 a disambiguating prefix for classes, functions, variables.

You have already seen namespaces in action when you wrote `std::vector`: the 'std' is the name of the namespace.

Defining a namespace

You can make your own namespace by writing

```
|| namespace a_namespace {
    // definitions
    class an_object {
    };
}
```

so that you can write

Namespace usage

```
|| 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:

Why not ‘using namespace std’?

This compiles, but should not:

```
|| #include <iostream>
|| using namespace std;

|| 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;

|| 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 literally 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 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.

20.2 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) ] = ...
```

20.3 Conditionals

There are a couple of *preprocessor conditions*.

20.3.1 Check on a value

The `#if` macro tests on nonzero. A common application is to temporarily remove code from compilation:

```
#if 0
    bunch of code that needs to
    be disabled
#endif
```

20.3.2 Check for macros

The `#ifdef` test tests for a macro being defined. Conversely, `#ifndef` tests for a macro not being defined. For instance,

```
|| #ifndef N  
|| #define N 100  
|| #endif
```

Why would a macro already be defined? Well you can do that on the compile line:

```
|| icpc -c file.cc -DN=500
```

Another application for this test is in preventing recursive inclusion of header files; see section 18.5.

20.3.3 Including a file only once

It is easy to wind up including a file such as `iostream` more than once, if it is included in multiple other header files. This adds to your compilation time, or may lead to subtle problems. A header file may even circularly include itself. To prevent this, header files often have a structure

```
// this is foo.h  
#ifndef FOO_H  
#define FOO_H  
  
// the things that you want to include  
  
#endif
```

Now the file will effectively be included only once: the second time it is included its content is skipped.

Many compilers support the pragma `#once` (which, however, is not a language standard) that has the same effect:

```
// this is foo.h  
#pragma once  
  
// the things you want to include only once
```

20.4 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.

Templated type name

If you have multiple routines that do ‘the same’ for multiple types, you want the type name to be a variable. Syntax:

```
|| template <typename yourtypevariable>
|| // ... stuff with yourtypevariable ...
```

Historically `typename` was `class` but that’s confusing.

21.1 Templatized functions

Example: function

Definition:

```
|| template<typename T>
|| void function(T var) { cout << var << endl; }
```

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:

make[5]: *** No rule to make target 'run_e

21.2 Templatized classes

The most common use of templates is probably to define templated classes. You have in fact seen this mechanism in action:

Templated vector

the STL contains in effect

```

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
}

```

21.3 Specific implementation

```

template <typename T>
void f(T);
template <>
void f(char c) { /* code with c */ };
template <>
void f(double d) { /* code with d */ };

```

21.4 Templating over non-types

THESE EXAMPLES ARE NOT GOOD.

See: <https://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templat>

Templating a value

Templating over integral types, not double.

The templated quantity is a value:

```
|| template<int s>
|| std::vector<int> svector(s);
|| /* ... */
|| svector(3) threevector;
|| cout << threevector.size();
```

Exercise 21.2. Write a class that contains an array. The length of the array should be templated.

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

Use assertions during development

```
|| #include <cassert>
|| ...
|| assert( bool expression )
```

Assertions are disabled by

```
|| #define NDEBUG
```

before the include.

You can pass this as compiler flag:

```
icpc -DNDEBUG yourprog.cxx
```

Function return values

22.2.2 Exception handling

Exception throwing

Throwing an exception is one way of signalling an error or unexpected behaviour:

```
|| void do_something() {
||     if ( oops )
||         throw(5);
|| }
```

Catching an exception

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;
|| }
```

You can throw integers to indicate an error code, a string with an actual error message. You could even make an error class:

Exception classes

```
|| 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!

Multiple catches

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 any 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 $i = 0 \dots 20$.

- 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.

Exceptions in constructors

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
}

```

More about exceptions

- 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!** (**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 `set jmp` / `long jmp` functions should not be used. These functions for instance do not have 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 the 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 actually implemented in terms of already existing language mechanisms. The STL is enormous, so we just highlight a couple of parts.

You have already seen

- arrays (chapter 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> age;
```

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;
age.insert(make_pair("Alice",29));
age["Bob"] = 32;
```

You can range over a map:

```
for ( auto person : age )
    cout << person.first << " has age " << person.second << endl;
```

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.

Iterators outside a loop

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:

make[5]: *** No rule to make target 'run_i

For the source of this example, see section 23.8.2

(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:

Iterators in vector methods

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`:

make[5]: *** No rule to make target `run_e`

For the source of this example, see section [23.8.2](#)

Note: end is exclusive.

23.2.2.2 Algorithms using iterators

Numerical reductions can be applied using iterators:

Reduction operation

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`:

make[5]: *** No rule to make target `run_a`

For the source of this example, see section [23.8.4](#)

Reduction with supplied operator

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:

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section [23.8.4](#)

23.2.2.3 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);
cout << "no first and last: ";
for ( auto i : sub ) cout << i << ", ";
cout << endl;
```

Output

[iter] subvector:

```
make[5]: *** No rule to make target 'run_s
```

For the source of this example, see section [23.8.5](#)

23.3 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.

23.3.1 Tuples

Remember how in section [7.5.2](#) we said that if you wanted to return more than one value, you could not do that through a return value, and had to use an *output parameter*? Well, using the STL there is a different solution.

You can make a *tuple*: an entity that comprises several components, possibly of different type, and which unlike a `struct` you do not need to define beforehand.

C++11 style tuples

```
|| std::tuple<int,double> id = std::make_tuple(3,5.12);
|| id = std::make_tuple(3,5.12);
|| 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(false,-1);
||   else
||     return make_tuple(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)};
|| }
```

Catching a returned tuple

The calling code is particularly elegant:

Code:

```
|| auto [succeed,y] = maybe_root1(x);
|| if (succeed)
||   cout << "Root of " << x << " is " << y << endl;
|| else
||   cout << "Sorry, " << x << " is negative" << endl;
|| //codesnippet tupleauto
|| /* ... */
|| }

|| if (false) {
||   auto [succeed,y] = maybe_root2(x);
||   if (succeed)
||     cout << "Root of " << x << " is " << y << endl;
||   else
||     cout << "Sorry, " << x << " is negative" << endl;
|| }
|| return 0;
|| }
```

Output

[stl] tuple:

make[5]: *** No rule to make target 'run_t

For the source of this example, see section [23.8.6](#)

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;

```

23.3.2 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. This can be done through `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 {}:

```

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();

```

23.4 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`
- 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. An examples applied to a `std::vector`:

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:**

```
make[5]: *** No rule to make target 'run_p
```

For the source of this example, see section [24.9.5](#)

See section [23.2.2](#) for iterators, and section [24.3](#) for the use of lambda expressions.

23.5 Limits

There used to be a header file `limits.h` that contained macros such as `MAX_INT`. While this is still available, the STL offers a better solution in the `numeric_limits` header.

Templated functions for limits

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;

cout << numeric_limits<long>::max();
```

Exercise 23.1. Write a program to discover what the maximal n is so that $n!$, that is, n -factorial, can be represented in an `int`, `long`, or `long long`. Can you write this as a templated function?

Operations such as dividing by zero lead to floating point numbers that do not have a valid value. For efficiency of computation, the processor will compute with these as if they are any other floating point number.

There are tests for detecting whether a number is `Inf` or `Nan`. However, using these may slow a computation down.

Detection of Inf and NaN

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
```

23.5.1 Storage

- An `int` used to be 16 bits in the olden days, but these days 32 bits is more common.
- A `long int` has at least the range of an `int`, and at least 32 bits.
- A `long long int` has at least the range of a `long`, and at least 64 bits.

23.6 Random numbers

The STL has a *random number generator* that is more general and more flexible than the C version (section 7.8.1).

- There are several generators that give uniformly distributed numbers;
- then there are distributions that translate this to non-uniform or discrete distributions.

Random number example

```
// 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
```

23.7 Time

23.8 Sources used in this chapter

23.8.1 Listing of code/stl/iter

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** itera.hxx : use of iterators  
*****  
*****  
<#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
int main() {
```

```

    {
        cout << "Iter" << endl;
        //codesnippet veciterator
        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;
        //codesnippet end
        cout << "iter" << endl;
    }

    {
        cout << "Erase.." << endl;
        //codesnippet vecerase
        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;
        //codesnippet end
        cout << "...erase" << endl;
    }

    return 0;
}

```

23.8.2 Listing of code/stl/iter

```

/*
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** iteracxx : use of iterators
*****
*/

```

```

#include <iostream>

```

```
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

{
    cout << "Iter" << endl;
    //codesnippet veciterator
    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;
    //codesnippet end
    cout << "iter" << endl;
}

{
    cout << "Erase.." << endl;
    //codesnippet vecerase
    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;
    //codesnippet end
    cout << "...erase" << endl;
}

return 0;
}
```

23.8.3 Listing of code/stl/reduce

```
*****
**** This file belongs with the course
```

```

***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** reduce.cxx : use of reductions
*****
***** ****
****

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

#include <vector>
using std::vector;

#include <numeric>
using std::accumulate;
using std::multiplies;

int main() {

{
    cout << "Accumulate .." << endl;
    //codesnippet vecaccumulate
    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;
    //codesnippet end
    cout << "... accumulate" << endl;
}

{
    cout << "Product .." << endl;
    //codesnippet vecproduct
    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;
    //codesnippet end
    cout << "... product" << endl;
}

return 0;
}

```

23.8.4 Listing of code/stl/reduce

```

***** ****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
***** ****
****
```

23.8.5 Listing of code/iter/iter

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu
```

```
****  
**** iter.cxx : tinkering with iterators  
****  
*****  
  
#include <iostream>  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
#include <memory>  
using std::shared_ptr;  
  
int main() {  
  
{  
    cout << "Subvectorcopy" << endl;  
    //codesnippet subvectorcopy  
    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;  
    //codesnippet end  
    cout << "... subvector" << endl;  
}  
  
{  
    cout << "Subvectorassign" << endl;  
    //codesnippet subvectorcopy  
    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;  
    //codesnippet end  
    cout << "... subvector" << endl;  
}  
  
{  
    cout << "Subpointer" << endl;
```

```

//codesnippet subpointer
auto vec = shared_ptr<vector<int>>(new vector<int>{11,22,33,44,55,66});
auto second = vec->begin(); second++;
auto before = vec->end(); before--;
auto sub = shared_ptr<vector<int>>(new vector<int>(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.get()) ) cout << i << ", ";
cout << endl;
vec->at(1) = 222;
cout << "did we get a change in the sub vector? " << sub->at(0) << endl;
//codesnippet end
cout << "... subpointer" << endl;
}

{
//codesnippet iterderef
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;
//codesnippet end
}

return 0;
}

```

23.8.6 Listing of code/stl/tuple

```

*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** tuple.cxx : std::tuple
****

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

#include <cmath>

#include <tuple>
using std::make_tuple;
using std::tuple;

//codesnippet tuplemake
auto maybe_root1(float x) {

```

```

    if (x<0)
        return make_tuple<bool,float>(false,-1);
    else
        return make_tuple<bool,float>(true,sqrt(x));
};

//codesnippet end

//codesnippet tupledenote
tuple<bool,float> maybe_root2(float x) {
    if (x<0)
        return {false,-1};
    else
        return {true,sqrt(x)};
};
//codesnippet end

int main() {
    float x;
    cin >> x;
{
    //codesnippet tupleauto
    auto [succeed,y] = maybe_root1(x);
    if (succeed)
        cout << "Root of " << x << " is " << y << endl;
    else
        cout << "Sorry, " << x << " is negative" << endl;
    //codesnippet tupleauto
}

if (false) {
    auto [succeed,y] = maybe_root2(x);
    if (succeed)
        cout << "Root of " << x << " is " << y << endl;
    else
        cout << "Sorry, " << x << " is negative" << endl;
}
return 0;
}

```

23.8.7 Listing of code/stl/printeach

```

*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** printeach.cxx : use of for_each
*****
***** ****

```

```

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

```

```
//codesnippet printeach
#include <algorithm>
using std::for_each;
//codesnippet end

#include <vector>
using std::vector;

int main() {

    cout << "Print" << endl;
    //codesnippet printeach
    vector<int> ints{3,4,5,6,7};
    for_each
        ( ints.begin(), ints.end(),
        [] (int x) -> void {
            if (x%2==0)
                cout << x << endl;
        } );
    //codesnippet end
    cout << "print" << endl;

    cout << "Count" << endl;
    //codesnippet counteach
    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;
    //codesnippet end
    cout << "count" << endl;

    return 0;
}
```

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) );
```

Type deduction in functions

Return type can be deduced in C++17:

```
|| auto equal(int i,int j) {
  return i==j;
};
```

Type deduction in functions

Return type 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 and references, 1

`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:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [24.9.1](#)

Auto and references, 2

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:

make[5]: *** No rule to make target 'run_r

For the source of this example, see section [24.9.2](#)

Auto and references, 3

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[5]: *** No rule to make target 'error_constrefget

24.1.2 Iterating

Auto iterators

```
vector<int> myvector(20) is actually short for:
for ( auto copy_of_int : myvector )
    s += copy_of_int;
for ( auto &ref_to_int : myvector )
```

for (std::iterator it=myvector.begin() ;
 it!=myvector.end() ; ++it)
 s += *it; // note the deref

Range iterators Can be used with anything that is iterable
(vector, map, your own classes!)

Other iterator uses

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);

```

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 iterable like this.)

This is not magic: it is possible to iterate over any class: a *class* is *iteratable* that has a number of conditions satisfied.

The class needs to have:

- a method `begin` with prototype

```
|| iteratableClass iteratableClass::begin()
```

That gives an object in the initial state, which we will call the ‘iterator object’; likewise

- a method `end`

```
|| iteratableClass iteratableClass::end()
```

that gives an object in the final state; furthermore you need

- an increment operator

```
|| void iteratableClass::operator++()
```

that advances the iterator object to the next state;

- a test

```
|| bool iteratableClass::operator!=(const iteratableClass&)
```

to determine whether the iteration can continue; finally

- a dereference operator

```
|| iteratableClass::operator*()
```

that takes the iterator object and returns its state.

Simple illustration

Let’s make a class, called a `bag`, that models a set of integers, and we want to enumerate them. For simplicity sake we will make a set of contiguous integers:

```

|| class bag {
||   // basic data
|| private:
||   int first, last;
|| public:
||   bag(int first,int last) : first(first),last(last) {};

```

Internal state

When you create an iterator object it will be copy of the object you are iterating over, except that it remembers how far it has searched:

```

|| private:
||   int seek{0};

```

Initial/final state

The `begin` method gives a `bag` with the `seek` parameter initialized:

```

|| public:
||   bag &begin() {
||     seek = first; return *this;
||   };
||   bag end() {
||     return *this;
||   };

```

These routines are public because they are (implicitly) called by the client code.

Termination test

The termination test method is called on the iterator, comparing it to the `end` object:

```

|| bool operator!=( const bag &test ) const {
||   return seek<=test.last;
|| };

```

Dereference

Finally, we need the increment method and the dereference. Both access the `seek` member:

```

|| void operator++() { seek++; };
|| int operator*() { return seek; };

```

Use case

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:**

```
make[5]: *** No rule to make target 'run_b
```

For the source of this example, see section ??

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 [40.15](#), implementing a prime number generator with this mechanism.

If you think you understand `const`, consider that the `has` method is conceptually `const`. 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* makes dynamic definition of functions possible.

Lambda expressions

```
[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:

```
|| 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.

24.3.1 Lambda members of classes

Storing a lambda in a class is hard because it has unique type. Solution: use *std::function*.

Lambda in object

```
#include <functional>
using std::function;
/* ... */
class SelectedInts {
private:
    vector<int> bag;
    function< bool(int) > selector;
public:
    SelectedInts( function< bool(int) > f ) {
        selector = f; }
    void add(int i) {
        if (selector(i))
            bag.push_back(i);
    }
    int size() { return bag.size(); }
    std::string string() { std::string s;
        for ( int i : bag )
            s += to_string(i)+" ";
        return s;
    }
};
```

Illustration

Code:

```
|| SelectedInts multiples
||   ( [divisor] (int i) -> bool {
||     return i%divisor==0; } );
||   for (int i=1; i<50; i++)
||     multiples.add(i);
```

Output

[func] lambdafun:

```
make[5]: *** No rule to make target 'run_l...
```

For the source of this example, see section 24.9.4

Exercise 24.3. Refer to 7.17 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 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.

24.3.3 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:`

make[5]: *** No rule to make target 'run_c

For the source of this example, see section 24.9.5

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 7 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 bignum = static_cast<size_t>(hundredk) * hundredk;
|| cout << "bignum: " << bignum << endl;
```

However, if the conversion is possible, the result may still not be ‘correct’.

Code:

```

long int hundreddg = 1000000000000;
cout << "long number:      "
     << hundreddg << endl;
int overflow;
overflow = static_cast<int>(hundreddg);
cout << "assigned to int: "
     << overflow << endl;

```

Output**[cast] intlong:**

make[5]: *** No rule to make target `run_i

For the source of this example, see section [24.9.6](#)

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: 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] derivright:

make[5]: *** No rule to make target 'run_d

For the source of this example, see section ??

On the other hand, a **static_cast** would not do the job:

Code:

```
void g( Base *obj ) {
    Derived *der =
        static_cast<Derived*>(obj);
    der->print();
}
Base *object = new Derived();
g(object);
Base *nobject = new Erived();
g(nobject);
```

Output

[cast] derivewrong:

make[5]: *** No rule to make target 'run_d

For the source of this example, see section ??

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 A word about void pointers

A traditional use for casts in C was the treatment of *void pointers*. The need for this is not as severe in C++ as it was before.

A typical use of void pointers appears in the PETSc [?, ?] 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 *Ivalue* 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. That said, here are some of the highlights of recent standards.

24.8.1 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;  
}
```

24.8.2 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

```
|| 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  
}
```

24.8.3 C++17**24.8.4 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
- *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.

24.9 Sources used in this chapter**24.9.1 Listing of code/auto/plainget**

```

/*
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** plainget.cxx : combining auto and references
*****
*/
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//codesnippet autoclass
class A {
private: float data;
public:
    A(float i) : data(i) {};
    auto &access() {
        return data; };
    void print() {
        cout << "data: " << data << endl; };
};

//codesnippet end

int main() {

    //codesnippet autoplain
    A my_a(5.7);
    auto get_data = my_a.access();
    get_data += 1;
    my_a.print();
    //codesnippet end
}

```

```
    }  
    return 0;  
}
```

24.9.2 Listing of code/auto/refget

```
/******  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** refget.cxx : combining auto and references  
****  
*****/  
  
#include <iostream>  
using std::cin;  
using std::cout;  
using std::endl;  
  
class A {  
    private: float data;  
    public:  
        A(float i) : data(i) {};  
        auto &access() {  
            return data; };  
        void print() {  
            cout << "data: " << data << endl; };  
};  
  
int main() {  
  
    //codesnippet autoref  
    A my_a(5.7);  
    auto &get_data = my_a.access();  
    get_data += 1;  
    my_a.print();  
    //codesnippet end  
  
    return 0;  
}
```

24.9.3 Listing of code/auto/constrefget

```
/******  
****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu  
****  
**** constrefget.cxx : combining auto and references  
**** This Does Not Compile  
****  
*****/  
  
#include <iostream>
```

```
using std::cin;
using std::cout;
using std::endl;

class A {
private: float data;
public:
    A(float i) : data(i) {};
    auto &access() {
        return data; }
    void print() {
        cout << "data: " << data << endl; }
};

int main() {
    //codesnippet constrefget
    A my_a(5.7);
    const auto &get_data = my_a.access();
    get_data += 1;
    my_a.print();
    //codesnippet end

    return 0;
}
```

24.9.4 Listing of code/func/lambdafun

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** lambdafun.cxx : storing a lambda
*****
***** //codesnippet lambdaaclass
#include <functional>
using std::function;
//codesnippet end

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

#include <vector>
using std::vector;

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

//codesnippet lambdaaclass
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;
    }
}
//codesnippet end

int main() {

    SelectedInts greaterthan5
        ( [] (int i) -> bool { return i>5; } );
    int upperbound = 20;
    for (int i=0; i<upperbound; i++)
        greaterthan5.add(i);
    // cout << "Ints under " << upperbound <<
    //     " greater than 5: " << greaterthan5.size() << endl;

    int divisor;
    cout << "Give a divisor: "; cin >> divisor; cout << endl;
    cout << "... using " << divisor << endl;
    //codesnippet lambdaclassed
    SelectedInts multiples
        ( [divisor] (int i) -> bool {
            return i%divisor==0; } );
    for (int i=1; i<50; i++)
        multiples.add(i);
    //codesnippet end
    cout << "Multiples of " << divisor << ":" "
        << "\n"
        << multiples.string() << endl;

    return 0;
}

```

24.9.5 Listing of code/stl/printeach

```

/*
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** printeach.cxx : use of for_each
*****
*/

```

```

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

//codesnippet printeach
#include <algorithm>
using std::for_each;
//codesnippet end

#include <vector>
using std::vector;

int main() {

    cout << "Print" << endl;
    //codesnippet printeach
    vector<int> ints{3,4,5,6,7};
    for_each
        ( ints.begin(), ints.end(),
        [] (int x) -> void {
            if (x%2==0)
                cout << x << endl;
        } );
    //codesnippet end
    cout << "print" << endl;

    cout << "Count" << endl;
    //codesnippet counteach
    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;
    //codesnippet end
    cout << "count" << endl;

    return 0;
}

```

24.9.6 Listing of code/cast/intlong

```

*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** intlong.cxx : cast int
*****
***** /*****

```

```
#include <iostream>
using namespace std; //::static_cast;

int main() {
    //codesnippet longcast
    long int hundredg = 1000000000000;
    cout << "long number: "
        << hundredg << endl;
    int overflow;
    overflow = static_cast<int>(hundredg);
    cout << "assigned to int: "
        << overflow << endl;
    //codesnippet end

    return 0;
}
```

Chapter 25

C++ for C programmers

25.1 I/O

There is little employ for `printf` and `scanf`. Use `cout` (and `cerr`) and `cin` instead. There is also the `fmlib` library.

Chapter 13.

25.2 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.

25.2.1 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:

```
make[5]: *** No rule to make target 'run_c...
```

For the source of this example, see section 25.10.1

25.3 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.

25.4 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.

25.5 Pointers

Many of the uses for C *pointers*, which are really addresses, have gone away.

- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see above.
- Traversing arrays and vectors can be done with ranges; section 11.2.
- To pass an argument by *reference*, use a *reference*. Section 7.5.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

There are some legitimate needs for pointers, such as Objects on the heap. In that case, use `shared_ptr` or `unique_ptr`; section 15.2. The C pointers are now called *bare pointers*, and they can still be used for ‘non-owning’ occurrences of pointers.

25.5.1 Parameter passing

No longer by address: now true references! Section 7.5.

25.6 Objects

Objects are structures with functions attached to them. Chapter 10.

25.7 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter 19.

25.8 Templates

If you find yourself writing the same function for a number of types, you'll love templates. Chapter 21.

25.9 Obscure stuff

25.9.1 Lambda

Function expressions. Section 24.3.

25.9.2 Const

Functions and arguments can be declared const. This helps the compiler. Section 17.1.

25.9.3 Lvalue and rvalue

Section 24.5.

25.10 Sources used in this chapter

25.10.1 Listing of code/array/cvector

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2019 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** cvector.hxx : vector from C array  
*****  
*****  
#include <iostream>  
using std::cerr;  
using std::cin;  
using std::cout;  
using std::endl;  
  
#include <vector>  
using std::vector;  
  
int main() {  
  
    {  
        int length{53};
```

```
//codesnippet cvector1
float *x;
x = (float*)malloc(length*sizeof(float));
//codesnippet end
if (!x) {
    cerr << "Could not allocate example 1\n";
    throw(1);
}
//codesnippet cvector1
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;
//codesnippet end
}

return 0;
}
```

PART III

FORTRAN

Chapter 26

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

26.1 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.

26.2 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 26.1. 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:

```
make[5]: *** No rule to make target 'run_s
```

For the source of this example, see section [26.8.1](#)

26.3 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.

26.3.1 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](#).)

26.3.2 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
```

26.3.3 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
```

26.4 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 26.4.1.
- *attributes* can be `dimension`, `allocatable`, `intent`, `parameters` et cetera.

- *name* is something you come up with. This has to start with a letter.

Data types

- Numeric: Integer, Real, Complex. Further specifications for numerical precision are discussed in section 26.4.2.
- Logical: Logical.
- Character: Character. Strings are realized as arrays of characters; chapter 31.

26.4.1 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
```

26.4.2 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.

Single/double precision constants

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:**

make[5]: *** No rule to make target `run_e

For the source of this example, see section [26.8.2](#)

You can query how many bytes a data type takes with `kind`.

Numerical precision

Number of bytes determines numerical precision:

- Computations in 4-byte have relative error $\approx 10^{-6}$
- Computations in 8-byte have relative error $\approx 10^{-15}$

Also different exponent range: max $10^{\pm 50}$ and $10^{\pm 300}$ respectively.

Storage size

F08: `storage_size` reports number of bits.

F95: `bit_size` works on integers only.

`c_sizeof` reports number of bytes, requires `iso_c_binding` module.

Force a constant to be `real(8)`:

Double precision constants

```
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

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:**

make[5]: *** No rule to make target `run_c

For the source of this example, see section [26.8.3](#)

26.4.3 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.

26.5 Input/Output, or I/O as we say

Simple I/O

- 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.

26.6 Expressions

Arithmetic expressions

- Pretty much as in C++
- Exception: `r**a` for power r^a .
- Modulus is a function: `MOD(7,3)`.

Boolean expressions

- Long form:

```
.and. .not. .or.
.lt. .le. .eq. .ne. .ge. .gt.
.true. .false.
```

- Short form:

```
< <= == /= > >=
```

Conversion and casting

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

Complex numbers exist

Strings

Strings are delimited by single or double quotes.

For more, see chapter 31.

26.7 Review questions

Exercise 26.2. What is the output for this fragment, assuming i, j are integers?

```
|| integer :: idiv
|| /* ... */
|| i = 3 ; j = 2 ; idiv = i/j
|| print *, idiv
```

Exercise 26.3. What is the output for this fragment, assuming i, j are integers?

```
|| real :: fdiv
|| /* ... */
|| i = 3 ; j = 2 ; fdiv = i/j
|| print *, fdiv
```

Exercise 26.4. 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 26.5. 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 26.6. 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

```

26.8 Sources used in this chapter

26.8.1 Listing of code/basicf/stop

```

!*****
! ***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
! ***
!*** emptyprog.F90 : an empty program
! ***
!*****
!!codesnippet stopf
Program SomeProgram
  stop 'the code stops here'
End Program SomeProgram
!!codesnippet end

```

26.8.2 Listing of code/basicf/e0

```

!*****
! ***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
! ***
!*** e0.F90 : constants
! ***
!*****
Program D0
  implicit none

  !!codesnippet f0const
  real(8) :: x, y, z
  x = 1.
  y = .1
  z = x+y
  print *, z
  x = 1.d0
  y = .1d0
  z = x+y
  print *, z
  !!codesnippet end

End Program D0

```

26.8.3 Listing of code/basicf/complex

```
!*****  
!***  
!*** This file belongs with the course  
!*** Introduction to Scientific Programming in C++/Fortran2003  
!*** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu  
!***  
!*** complex.F90 : basic complex stuff  
!***  
!*****  
  
Program Complex  
    implicit none  
  
    !!codesnippet fcomplex  
Complex :: &  
        fourtyfivedegrees = (1.,1.), &  
        other  
print *,fourtyfivedegrees  
other = 2*fourtyfivedegrees  
print *,other  
    !!codesnippet end  
  
End Program Complex
```


Chapter 27

Conditionals

27.1 Forms of the conditional statement

The Fortran conditional statement uses the `if` keyword:

Conditionals

Single line conditional:

```
|| if ( test ) statement
```

The full if-statement is:

```
|| if ( something ) then
    do something
else
    do otherwise
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
```

27.2 Operators

Comparison and logical operators

Operator	old style	meaning	example
<code>==</code>	<code>.eq.</code>	equals	<code>x==y-1</code>
<code>/=</code>	<code>.ne.</code>	not equals	<code>x*x*!=5</code>
<code>></code>	<code>.gt.</code>	greater	<code>y>x-1</code>
<code>>=</code>	<code>.ge.</code>	greater or equal	<code>sqrt(y)>=7</code>
<code><</code>	<code>.lt.</code>	less than	
<code><=</code>	<code>.le.</code> <code>.and. .or.</code> <code>.not.</code> <code>.eqv.</code> <code>.neqv.</code>	less equal and, or not equiv not equiv	<code>x<1 .and. x>0</code> <code>.not. (x>1 .and. x<2)</code> $(x \wedge y) \vee (\neg x \wedge \neg y)$ $(x \wedge \neg y) \vee (\neg x \wedge y)$

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

Exercise 27.1. Read in three grades: Algebra, Biology, Chemistry, each on a scale $1 \dots 10$.

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.

27.3 Select statement

The Fortran equivalent of the C++ `case` statement is `select`. It takes single values or ranges; works for integers and characters.

Select statement

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!

27.4 Boolean variables

The Fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`

Exercise 27.2. Print a boolean variable. What does the output look like in the true and false case?

27.5 Review questions

Exercise 27.3. What is a conceptual difference between the C++ `switch` and the Fortran `Select statement`?

Chapter 28

Loop constructs

28.1 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.

Indexed Do loops

```
|| 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
```

While loop

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.

F77 note: Do not use label-terminated loops. Do not use non-integer loop variables.

28.2 Interruptions of the control flow

For indeterminate looping, you can use the `while` test, or leave out the loop parameter altogether. In that case you need the `exit` statement to stop the iteration.

Exit and cycle

Loop without counter or while test:

```
|| do
  call random_number(x)
  if (x>.9) exit
  print *, "Nine out of ten exes agree"
end do
```

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
```

28.3 Implied do-loops

There are do loops that you can write in a single line by an expression and a loop header. In effect, such an *implied do loop* becomes the sum of the indexed expressions. This is useful for I/O. For instance, iterate a simple expression:

Implied do loops

```
|| 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 28.1. 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.

28.4 Review questions

Exercise 28.2. 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 29

Scope

29.1 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.

29.1.1 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
```

29.1.1.1 Variables in a module

Variables in a module (section 33.2) have a lifetime that is independent of the calling hierarchy of program units: they are *static variables*.

29.1.1.2 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.

29.1.2 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 30

Subprograms and modules

30.1 Procedures

Programs can have subprograms: parts of code that for some reason you want to separate from the main program. If you structure your code in a single file, this is the recommended structure:

Subprograms in contains clause

```
|| Program foo
||   < declarations>
||   < executable statements >
|| Contains
||   < subprogram definitions >
|| End Program foo
```

That is, subprograms are placed after the main program statements, separated by a `Contains` clause.

In general, these are the placements of subprograms:

- Internal: after the `Contains` clause of a program
- In a `Module`; see section 33.2.
- Externally: the subprogram is not internal to a `Program` or `Module`. In this case it's safest to declare it through an `Interface` specification; section 30.2.

30.1.1 Subroutines and functions

Fortran has two types of subprograms:

- 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 30.1. Rewrite the above subroutine **foo** without a **return** statement.

A subroutine is invoked with a **call** statement:

```
|| call foo()
```

Subroutine with argument

Code:

```
|| program printing
||   implicit none
||   call printint(5)
||   contains
||     subroutine printint(invalue)
||       implicit none
||       integer :: invalue
||       print *, invalue
||     end subroutine printint
||   end program printing
```

Output

[funcf] **printone:**

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [30.3.1](#)

Subroutine can change argument

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:**

make[5]: *** No rule to make target 'run_a

For the source of this example, see section [30.3.2](#)

Recursion

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:

make[5]: *** No rule to make target 'run_f

For the source of this example, see section [30.3.3](#)

Note the `result` clause. This prevents ambiguity.

30.1.2 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 subprogram that return a result to its calling program, much like a non-void function in C++

Function definition and usage

- **subroutine vs function:**
compare `void` functions vs non-void in C++.
- Return type, keyword `function`, name, parameters
- Function body has statements
- Result is returned by assigning to the function name
- Use: `y = f(x)`

Function example

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:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section 30.3.4

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 33.2 below), it becomes known through a `use` statement.

F77 note: Without modules and `contains` sections, you need to declare the function type explicitly in the calling program. The safe way is through using an `interface` specification.

Exercise 30.2. 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:

make[5]: *** No rule to make target 'run_r

For the source of this example, see section 30.3.5

Why a ‘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

Why a ‘contains’ clause, take 2

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:

make[5]: *** No rule to make target ‘run_n

For the source of this example, see section ??

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.

30.1.3 Arguments

Subprogram arguments

Arguments are declared in subprogram 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.

Parameter passing

- Everything is passed by reference.
Don’t worry about large objects being copied.
- Optional intent declarations:
Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.

Fortran nomenclature

The term *dummy argument* is what Fortran calls the parameters in the subprogram definition. The arguments in the subprogram call are the *actual arguments*.

Intent checking

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

```

Why intent checking?

Self-protection: if you state the intended behaviour of a routine, the compiler can detect programming mistakes.

Allow compiler optimizations:

<pre> x = f() call ArgOut(x) print *,x </pre>	<pre> do i=1,1000 x = ! something y1 = x call ArgIn(x) y2 = ! same expression as y1 </pre>
Call to <code>f</code> removed	y ₂ is same as y ₁ because x not changed

(May need further specifications, so this is not the prime justification.)

Exercise 30.3. Write a subroutine `trig` that takes a number α as input and passes $\sin \alpha$ and $\cos \alpha$ back to the calling environment.

30.1.4 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.

30.1.5 More about arguments

Keyword 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:**

make[5]: *** No rule to make target `run_k`

For the source of this example, see section [30.3.6](#)

Optional arguments

- Extra specifier: `Optional`
- Presence of argument can be tested with `Present`

30.2 Interfaces

If you want to use a subprogram in your main program, the compiler needs to know the signature of the subprogram: 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 subprogram resides in the same file as the main program.

If the subprogram is in a separate file, the compiler does not see definition and usage in one go. To allow 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 subprogram.

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)

```

Subprogram:

```

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.

30.2.1 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.

30.3 Sources used in this chapter

30.3.1 Listing of code/funcf/printone

```
!*****
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** printone.F90 : trivial subroutine
!***
!*****
```

```
!!codesnippet fprintone
program printing
    implicit none
    call printint(5)
contains
    subroutine printint(invalue)
        implicit none
        integer :: invalue
        print *,invalue
    end subroutine printint
end program printing
!!codesnippet end
```

30.3.2 Listing of code/funcf/addone

```
!*****
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** addone.F90 : subroutine with output argument
!***
!*****
```

```
!!codesnippet faddone
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
!!codesnippet end
```

30.3.3 Listing of code/funcf/fact

```
!*****
!***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** fib.F90 : recursive Fibonacci function
!***
!*****
```

```
program Factorial
    implicit none
    integer :: i,f
    read *,i
    f = fact(i)
    print *,i,"factorial is",f
contains
    !!codesnippet frecursf
    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
    !!codesnippet end
end program Factorial
```

30.3.4 Listing of code/funcf/plusone

```
!*****
!***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017-9 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** plusone.F90 : function with return type
!***
!*****
```

```
!!codesnippet fplusone
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
```

```
|| !codesnippet end
```

30.3.5 Listing of code/funcf/readpos

```
!*****
!***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** readpos.F90 : exercise for function with intent out
!***
!*****
```

```
!!codesnippet readpos
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
        !!codesnippet end
        real(4) :: maybe
        do
            read *, maybe
            if (maybe<=0) then
                print *, "No, not", maybe
            else
                read_positive = maybe
                exit
            end if
        end do
        !!codesnippet readpos
    end function read_positive
end program readpos
!!codesnippet end
```

30.3.6 Listing of code/funcf/keyword

```
!*****
!***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** keyword.F90 : function call with keywords
!***
!*****
```

```
program keyword
    implicit none
    integer :: i
    !!codesnippet sayxykw
    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
  !!codesnippet end
end program keyword
```


Chapter 31

String handling

31.1 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"
```

31.2 Characters

31.3 Strings

The `len` function gives the length of the string as 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:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section ??

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:

make[5]: *** No rule to make target 'run_c

For the source of this example, see section ??

31.4 Strings versus character arrays

31.5 Sources used in this chapter

Chapter 32

Structures, eh, types

Fortran's way of bundling up data, and naming that bundle, is a `type`.

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 definition

Type name / End Type block.

Variable declarations inside the block

```
|| type mytype
||   integer :: number
||   character :: name
||   real(4) :: value
|| end type mytype
```

Creating a type structure

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
```

Member access

Access structure members with %

```
|| Type(mytype) :: typed_object
|| typed_object%member = ....
```

Example

```
|| type point
||   real :: x,y
|| end type point
|| type(point) :: p1,p2
|| p1 = point(2.5, 3.7)
|| p2 = p1
|| print *,p2%x,p2%y
```

Type definitions can go in the main program

You can have arrays of types:

```
|| type(my_struct) :: data
|| type(my_struct),dimension(1) :: data_array
```

Types as subprogram argument

```
|| real(4) function length(p)           || print *, "Length:",length(p2)
||   implicit none
||   type(point),intent(in) :: p
||   length = sqrt( p%x**2 + p%y )
|| end function length
```

Exercise 32.1. Define a type Point that contains real numbers x, y.

Define a type Rectangle that contains two Points.

Write a function area that has one argument, a Rectangle.

Chapter 33

Modules

Fortran has a clean mechanism for importing data, functions, types that are defined in another file.

Module definition

Modules look like a program, but without executable code:

```
Module definitions
  type point
    real :: x,y
  end type point
contains
  real(4) function length(p)
    implicit none
    type(point),intent(in) :: p
    length = sqrt( p%x**2 + p%y )
  end function length
end Module definitions
```

Module use

Module imported through `use` statement;
comes before `implicit none`

```
Program size
  use definitions
  implicit none

  type(point) :: p1,p2
  p1 = point(2.5, 3.7)

  p2 = p1
  print *,p2%x,p2%y

end Program size
```

Exercise 33.1. Take exercise 32.1 and put all type definitions and all functions in a module.

33.1 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.

Separate compilation of modules

Suppose program is split over theprogram.F90 and themodule.F90.

- icpc -c themodule.F90; this gives
- an *object file* that will be linked later, and
- a .mod file (with the name of the module, not of the file);
- icpc -c theprogram.F90 will read the .mod file; and finally
- icpc -o myprogram theprogram.o themodule.o uses the compiler as *linker* to form the executable.

The module needs to be compiled before the program.

33.2 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**.

Module use

```
|| Program ModProgram
||   use FunctionsAndValues
||   implicit none
|
||   print *, "Pi is:", pi
||   call SayHi()
|
|| End Program ModProgram
```

Also possible:

```
|| Use mymodule, Only: func1, func2
|| Use mymodule, func1 => new_name1
```

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**, **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 33.2. 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
```

33.2.1 Polymorphism

```
|| module somemodule
|
| INTERFACE swap
| MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
| END INTERFACE
|
| contains
| subroutine swapreal
| ...
| end subroutine swapreal
| subroutine swapint
| ...
| end subroutine swapint
```

33.2.2 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 34

Classes and objects

34.1 Classes

Classes and objects

Fortran classes are based on `type` objects, a little like the analogy between C++ `struct` and `class` constructs.

New syntax for specifying methods.

Object is type with methods

You define a type as before, with its data members, but now the type has a `contains` for the methods:

<pre>Module multmod type Scalar real(4) :: value contains procedure,public :: print procedure,public :: scaled end type Scalar contains ! methods end Module multmod</pre>	<pre>Program Multiply use multmod implicit none type(Scalar) :: x real(4) :: y x = Scalar(-3.14) call x%print() y = x%scaled(2.) print '(f7.3)',y end Program Multiply</pre>
--	---

Method definition

<pre>subroutine print(me) implicit none class(Scalar) :: me print'("The value is",f7.3)',me%value end subroutine print function scaled(me,factor) implicit none class(Scalar) :: me real(4) :: scaled,factor scaled = me%value * factor end function scaled</pre>

Methods have object as argument

You define functions that accept the type as first argument, but instead of declaring the argument as `type`, you define it as `class`.

The members of the class object have to be accessed through the `%` operator.

```

subroutine set(p, xu, yu)
  implicit none
  class(point) :: p
  real(8), intent(in) :: xu, yu
  p%x = xu; p%y = yu
end subroutine set

```

Class organization

- You're pretty much forced to use Module
- A class is a Type with a `contains` clause followed by `procedure` declaration
- Actual methods go in the `contains` part of the module
- First argument of method is the object itself.

Point program

<pre> Module PointClass Type, public :: Point real(8) :: x, y contains procedure, public :: distance End type Point contains ! End Module PointClass </pre>	<pre> 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 </pre>
--	---

Exercise 34.1. Take the point example program and add a distance function:

```

Type(Point) :: p1, p2

! initialize
dist = p1%distance(p2)

```

Exercise 34.2. Write a method add for the Point type:

```

Type(Point) :: p1, p2, sum
! initialize
sum = p1%add(p2)

```

What is the return type of the function `add`?

More OOP

Inheritance:

```

|| type, extends(baseclas) :: derived_class

```

Pure virtual:

```

|| type, abstract

```

Further reading

[http://fortranwiki.org/fortran/show/Object-oriented+
programming](http://fortranwiki.org/fortran/show/Object-oriented+programming)

Chapter 35

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.

35.1 Static arrays

The preferred way for specifying an array size is:

Fortran dimension

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 35.4 for dynamic arrays.)

Array indexing in Fortran is 1-based:

1-based Indexing

```
|| integer,parameter :: N=8
  || real(4),dimension(N) :: x
  || do i=1,N
    ||   ... x(i) ...
```

Unlike C++, Fortran can specify the lower bound explicitly:

Lower bound

```
|| real,dimension(-1:7) :: x
  || do i=-1,7
    ||   ... x(i) ...
```

Such arrays, as in C++, obey the scope: they disappear at the end of the program or subprogram.

35.1.1 Initialization

There are various syntaxes for *array initialization*, including the use of *implicit do-loops*:

Array initialization

```
|| 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 /)
```

35.1.2 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.

Array sections

- : to get all indices,
- :n to get indices up to n,
- n: to get indices n and up.
- m:n indices in range m, . . . , n.

Use of sections

Code:

```
|| real(8), dimension(5) :: x = &
   [.1d0, .2d0, .3d0, .4d0, .5d0]
  x(2:5) = x(1:4)
  print '(f5.3)', x
```

Output

[arrayf] sectionassign:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section ??

Exercise 35.1. 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?

You can even use a stride:

Strided sections

Code:

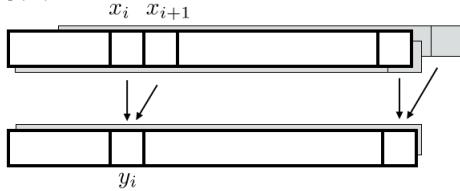
```
|| integer, dimension(5) :: &
   y = [0,0,0,0,0]
  integer, dimension(3) :: &
   z = [3,3,3]
  y(1:5:2) = z(:)
  print '(i3)', y
```

Output

[arrayf] sectionmng:

make[5]: *** No rule to make target 'run_s

For the source of this example, see section ??

Exercise 35.2.

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.

35.1.3 Integer arrays as indices*Index arrays*

```
|| integer, dimension(4) :: i = [2, 4, 6, 8]
|| real(4), dimension(10) :: x
|| print *, x(i)
```

35.2 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:

Multi-dimension arrays

```
|| 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 35.1.

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 35.3. Can you describe in words how memory elements are accessed if you would write

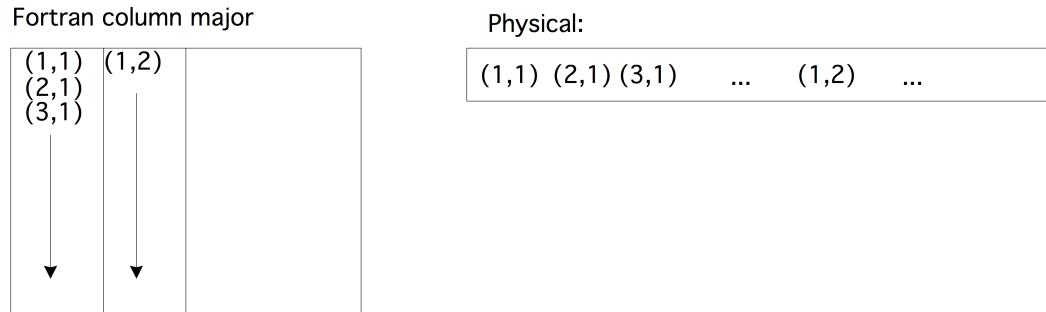


Figure 35.1: Column-major storage in Fortran

```

    do row=1, size(A, 1)
      do col=1, size(A, 2)
        .... A(row, col) ....
      end do
    end do
?
```

You can make sections in multi-dimensional arrays: you need to indicate a range in all dimensions.

Array sections in multi-D

```

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.

Array printing

Fill array by rows:

$$\begin{array}{cccc}
 1 & 2 & \dots & N \\
 N+1 & \dots & & \\
 & \dots & & \\
 & & & MN
 \end{array}$$

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:

make[5]: *** No rule to make target 'run_p

For the source of this example, see section [35.9.1](#)

35.2.1 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

[array] fsection2:

```
make[5]: *** No rule to make target 'run_f
```

For the source of this example, see section ??

- 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)
```

35.2.2 Reshaping

RESHAPE

```
|| array = RESHAPE( list, shape )
```

Example:

```
|| square = reshape( (/ (i, i=1, 16) /), (/4, 4/) )
```

SPREAD

```
|| array = SPREAD( old, dim, copies )
```

35.3 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:

Pass array: calling site

Passing array as one symbol:

```
|| Program ArrayComputations1D
||   use ArrayFunction
||   implicit none
|
| real(8),dimension(:) :: x(N)
| /* ... */
| print *, "Sum of one-based array:", arraysum(x)
```

Pass array: subprogram

Note declaration as dimension(:)
actual size is queried

```
|| real(8) function arraysum(x)
||   implicit none
||   real(8),intent(in),dimension(:) :: x
|
|   real(8) :: tmp = 0.
|   integer i
|
|   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*.

35.4 Allocatable arrays

Static arrays are fine at small sizes. However, there are two main arguments against using them at large sizes.

- Since the size is explicitly stated, it makes your program inflexible, requiring recompilation to run it with a different problem size.
- Since they are allocated on the so-called *stack*, making them too large can lead to *stack overflow*.

A better strategy is to indicate the shape of the array, and use `allocate` to specify the size later, presumably in terms of run-time program parameters.

Array allocation

```
|| real(8), dimension(:), allocatable :: x,y
|
| n = 100
| allocate(x(n), y(n))
```

You can `deallocate` the array when you don't need the space anymore.

If you are in danger of running out of memory, it can be a good idea to add a `stat=ierror` clause to the `allocate` statement:

```
|| integer :: ierr
|| allocate( x(n), stat=ierr )
|| if ( ierr/=0 ) ! report error
```

Has an array been allocated:

```
|| Allocated( x ) ! returns logical
```

Allocatable arrays are automatically deallocated when they go out of scope. This prevents the *memory leak* problems of C++.

Explicit deallocate:

```
|| deallocate( x )
```

35.5 Array output

Use implicit do-loops; section 28.3.

35.6 Operating on an array

35.6.1 Arithmetic operations

Between arrays of the same shape:

```
|| A = B+C  
|| D = D*E
```

(where the multiplication is by element).

35.6.2 Intrinsic functions

The following intrinsic functions are available for arrays:

Array intrinsics

- **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.

```
|| i = MAXLOC( array [, mask ] )
```

- **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.

Exercise 35.4. The 1-norm of a matrix is defined as the maximum sum 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:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Implement these functions using array intrinsics.

Exercise 35.5. 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?

35.6.3 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.

Operate where

```
|| where ( A<0 ) B = 0
```

Full form:

```
|| WHERE ( logical argument )
      sequence of array statements
ELSEWHERE
      sequence of array statements
END WHERE
```

35.6.4 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 35.6. Use array statements (that is, no loops) to fill a two-dimensional array A with random numbers between zero and one. Then fill two arrays A_{big} and A_{small} with the elements of A that are great than 0.5, or less than 0.5 respectively:

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i, j) = \begin{cases} 0 & \text{if } A(i, j) \geq 0.5 \\ A(i, j) & \text{otherwise} \end{cases}$$

Using more array statements, add `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) ) >
```

35.7 Array operations

35.7.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

35.7.1.1 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)
```

35.7.1.2 ‘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.

This mechanism is prone to misunderstanding and therefore now deprecated. It is not a parallel loop! For that, the following mechanism is preferred.

35.7.1.3 Do concurrent

Do concurrent

The `do concurrent` is a true do-loop. With the `concurrent` keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
|| do concurrent (i=1:n)
||   a(i) = b(i)
||   c(i) = d(i+1)
|| end do
```

(Do not use `for all`)

35.7.2 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 35.7. Create arrays A, C of length $2N$, and B of length N . Now implement

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \dots, N$$

and

$$C_i = A_{i/2}, \quad i = 1, \dots, 2N$$

using all four mechanisms. Make sure you get the same result every time.

35.7.3 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)
```

<i>Original</i>	1	2	3	4	5	6	7	8	9	10
<i>Section</i>	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

```

<i>Original</i>	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

```

<i>Original</i>	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.

35.8 Review questions

Exercise 35.8. 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?

35.9 Sources used in this chapter

35.9.1 Listing of code/arrayf/printarray

```
!*****
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** shape.F90 : array reshaping
!***
!*****
```

```
Program ArrayPrint
  implicit none
  integer,parameter :: M=4,N=5
  integer :: i,j,count=1

  real(4),dimension(M,N) :: rect

  !!codesnippet printarray
  do i=1,M
    do j=1,N
      rect(i,j) = count
      count = count+1
    end do
  end do
  print *,rect
  !!codesnippet end

End Program ArrayPrint
```

Chapter 36

Pointers

Pointers in C/C++ are based on memory addresses; Fortran pointers on the other hand, are more abstract.

36.1 Basic pointer operations

Pointers are aliases

- 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.

C++ vs Fortran pointers

Fortran pointers are 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.

Setting the pointer

- 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.

Pointer example

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:

make[5]: *** No rule to make target 'run_r...

For the source of this example, see section 36.4.1

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.

Assign pointer from other pointer

```
|| 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 36.1. 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, 2, 3, 4, 5, 9, 8, 7, 6, 0]
|| real,pointer :: biggest_element
|| integer :: index
|| logical :: correct

|| print '(10f5.2,1x)',array
|| call SetPointer(array,biggest_element)
|| print *, "Biggest element is",biggest_element
|| biggest_element = 0
|| print '(10f5.2,1x)',array
```

Output

[pointerf] arpointf:

make[5]: *** No rule to make target 'run_a

Pointer status

- Nullify: zero a pointer
- Associated: test whether assigned

Dynamic allocation

Associate unnamed memory:

```
|| Integer,Pointer,Dimension(:) :: array_point
|| Allocate( array_point(100) )
```

This is automatically deallocated when control leaves the scope.

36.2 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.

36.3 Example: linked lists

For pictures of linked lists, see section 51.1.2.

Linked list

- 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 36.2. 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.

Linked list datatypes

- 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)
```

List initialization

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

```

Attaching a 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

```

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

```

Inserting 1

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

```

Inserting 2

The actual insertion requires rerouting some pointers:

```

|| allocate(new_node)
|| new_node%value = value
|| new_node%next => current
|| previous%next => new_node

```

36.4 Sources used in this chapter

36.4.1 Listing of code/pointerf/realp

```
!*****
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** real.F90 : pointer to real
!***
!*****
```



```
Program PointAtReal

implicit none

!!codesnippet pointatreal
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
!!codesnippet end

end Program PointAtReal
```

Chapter 37

Input/output

37.1 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.

I/O commands

- 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 and unformatted I/O

- 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.

37.2 Print to terminal

The simplest command for outputting data is `print`.

```
|| print *, "The result is", result
```

In its easiest form, you use the star to indicate that you don't care about formatting; after the comma you can then have any number of comma-separated strings and variables.

37.2.1 Print on one line

The statement

```
|| print *, item1, item2, item3
```

will print the items on one line, as far as the line length allows.

Implicit do loops

Parametrized printing with an *implicit do loop*:

```
|| print *, ( i*i, i=1, n)
```

All values will be printed on the same line.

37.2.2 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)
```

37.2.3 Formats

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
```

Output

[iof] i4:

make[5]: *** No rule to make target 'run_i

For the source of this example, see section ??

(In the last line, the format specifier was not wide enough for the data, so an asterisk was used as output.)

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:

make[5]: *** No rule to make target 'run_i

For the source of this example, see section ??

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(i2,".")), i,j,k
```

Output**[iof] ij:**

make[5]: *** No rule to make target 'run_i

For the source of this example, see section ??

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.
- ‘*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:**

make[5]: *** No rule to make target 'run_a

For the source of this example, see section 37.5.1

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>

Format repetitions

```
|| print '( 3i4 )', i1,i2,i3
|| print '( 3(i2,":",f7.4) )', i1,r1,i2,r2,i3,r2
```

Repeats and line breaks

- If abc is a format string, then 10 (abc) gives 10 repetitions. There is no line break.
- If there is more data than specified in the format, the format is reused in a new print statement. This causes line breaks.
- The / (slash) specifier causes a line break.
- There may be a 80 character limit on output lines.

Exercise 37.1. Use formatted I/O to print the number 0 ··· 99 as follows:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

37.3 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.

37.3.1 Units

```
|| Open(11)
```

will result in a file with a name typically `fort.11`.

```
|| 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.

37.3.2 Other write options

```
|| write(unit,fmt,ADVANCE="no") data
```

will not issue a newline.

open Close

37.4 Unformatted output

So far we have looked at ascii output, which is nice to look at for a human , but is not the right medium to communicate data to another program.

- Ascii output requires time-consuming conversion.
- Ascii rendering leads to loss of precision.

Therefore, if you want to output data that is later to be read by a program, it is best to use *binary output* or *unformatted output*, sometimes also called *raw output*.

Unformatted output

Indicated by lack of format specification:

```
|| write (*) data
```

Note: may not be portable between machines.

37.5 Sources used in this chapter

37.5.1 Listing of code/fio/asterisk

```
!*****
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
!***
!*** asterisk.cxx : Fortran I/O
!***
!*****
```

```
Program Asterisk
  implicit none

  integer :: ipower, number=1

  !!codesnippet fasterisk
  do ipower=1, 5
    print '(i3)', number
    number = 10*number
  end do
  !!codesnippet end

end Program Asterisk
```


Chapter 38

Leftover topics

38.1 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)
```

38.2 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)
```

PART IV

EXERCISES AND PROJECTS

Chapter 39

Exercises

39.1 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;
```

39.2 Scope

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?

2. What is the output of:

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

3. What is the output of:

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

4. What is the output of:

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

39.3 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.

39.4 Subprograms

Exercise 39.1. Write the missing function `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 looppo
implicit none
real(4) :: userinput
do while (pos_input(userinput))
print &
' ("Positive input:",f7.3)',&
userinput
end do
print &
' ("Negative input:",f7.3)',&
userinput
/* ... */
end program looppo

```

Output**[funcf] looppo:**

```
make[5]: *** No rule to make target 'run_1
```

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?

39.5 Object oriented exercises

Exercise 39.2. Why is it a good idea to use an accessor function for the data members of a class, rather than declaring data members public and accessing them directly?

Exercise 39.3. 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 );
```

Exercise 39.4. Let a `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.

39.6 List access

Exercise 39.5. Explore the efficiency of using an array versus a linked list.

1. Compare re-allocating the array versus adding elements to the linked list. Start with a simple case: add elements only at the end, and keep a pointer to the final element in the list.
2. Investigate access times: retrieve many (as in: thousands if not more) elements from the array. Do this as follows: allocate an array of indexes, and repeatedly retrieve those list/array elements, for instance adding them together. Does the access time for the array go up if the number of elements gets large?
3. Optimize allocation for the list: create an array of list nodes and use those. Does this make a difference in access times?

Chapter 40

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.

40.1 Arithmetic

Before doing this section, make sure you study section 4.5.

Exercise 40.1. 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.

40.2 Conditionals

Before doing this section, make sure you study section 5.1.

Exercise 40.2. Read two numbers and print a message like

3 is a divisor of 9

if the first is an exact divisor of the second, and another message

4 is not a divisor of 9

if it is not.

40.3 Looping

Before doing this section, make sure you study section 6.1.

Exercise 40.3. 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.

Exercise 40.4. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

40.4 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 40.5. 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?

40.5 While loops

Before doing this section, make sure you study section 6.2.

Exercise 40.6. 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.)

40.6 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 40.7. Rewrite the exercise that found a predetermined number of primes, putting the `number_of_primes_found` and `last_number_tested` variables in a structure. Your main program should now look like:

```
|| 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?

40.7 Classes and objects

Before doing this section, make sure you study section 10.1.

In exercise 40.7 you made a structure that contains the data for a prime sequence, and you have separate functions that operate on that structure or on its members.

Exercise 40.8. 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:

```

    ||>> 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 40.9. The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes $p+q$. Write a program to test this for the even numbers up to a bound that you read in.

This is a great exercise for a top-down approach! Make an outer loop over the even numbers e . In each iteration, make a `primegenerator` object to generate p values. For each p test whether $e - p$ 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 40.10. 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:

$$n = \frac{p + q}{2} \quad \text{or} \quad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r \text{ prime}} \exists_{p,q \text{ prime}} : r = (p + q)/2 \text{ is prime.}$$

Write a program that tests this. You need at least one loop that tests all primes r ; for each r you then need to find the primes p, q that are equidistant to it. Do you use two generators for this, or is one enough?

For each r value, when the program finds the p, q values, print the q, p, r triple and move on to the next r .

Allocate an array where you record all the $p - q$ distances that you found. Print some elementary statistics, for instance: what is the average, do the distances increase or decrease with p ?

40.8 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, ...
2. The first number, 2, is a prime: record it and remove all multiples, giving

3, 5, 7, 9, 11, 13, 15, 17, ...

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, ...

4. The first remaining number, 5, is a prime: record it and remove all multiples, giving

7, 11, 13, 17, 19, 23, 29, ...

40.8.1 Arrays implementation

The sieve is easily implemented with an array that stores all integers.

Exercise 40.11. 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.

40.8.2 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 40.12. 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:

make[5]: *** No rule to make target 'run_i

Next, we need a stream that takes another stream as input, and filters out values from it.

Exercise 40.13. 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:

```
make[5]: *** No rule to make target 'run_o
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

Exercise 40.14. 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.

40.9 Range implementation

Before doing this section, make sure you study section 24.2.

Exercise 40.15. 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:

```
make[5]: *** No rule to make target 'run_r
```


Chapter 41

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.

41.1 Basic functions

Exercise 41.1. 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 41.2. 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:

```
make[5]: *** No rule to make target 'run_r
```

41.2 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 41.3. 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 41.4. 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 41.5. Write a method `halfway_point` that, given two `Point` objects `p, q`, construct the `Point` halfway, that is, $(p + q)/2$.

You can write this function directly, or you could write functions `Add` and `Scale` and combine these.

Exercise 41.6. 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:

```
make[5]: *** No rule to make target 'run_1'
```

Hint: see section 4.6.2.

Exercise 41.7. Revisit exercise 41.2 using the `Point` class. Your code should now look like:

```
|| newpoint = point.rotate(alpha);
```

Exercise 41.8. 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?

41.3 Using one class in another

Before doing this section, make sure you study section 10.2.

Exercise 41.9. Make a class `LinearFunction` with a constructor:

```
|| LinearFunction( Point input_p1, Point input_p2 );
```

and a 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 41.10. 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 41.11. Revisit exercises 41.2 and 41.7, 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.

Axi-parallel rectangle class

Intended API:

```
|| float Rectangle::area();
```

It would be convenient to store width and height; for

```
|| bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

Exercise 41.12.

- 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();
```

- Add a second constructor

```
|| Rectangle(Point bl,Point tr);
```

Can you figure out how to use member initializer lists for the constructors?

- Write another version of your class so that it stores two `Point` objects.

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.

41.4 Is-a relationship

Before doing this section, make sure you study section 10.3.

Exercise 41.13. 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 41.14. 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.

41.5 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 41.15. 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 42

Infectuous disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

42.1 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.

42.1.1 Other ways of modeling

Instead of capturing every single person in code, a ‘contact network’ model, it is possible to use an Ordinary Differential Equation (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 [?].

<http://mathworld.wolfram.com/Kermack-McKendrickModel.html>

This is known as a ‘compartamental model’. 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.

42.2 Coding up the basics

Before doing this section, make sure you study section 10.

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 42.1. 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.

42.3 Population

Before doing this section, make sure you study section 11.

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 42.2. 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.

42.4 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 42.3. 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 42.4. Incorporate inoculation: 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?

42.5 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 42.5. Code the random interactions. Now run a number of simulations varying

- The percentage of people inoculated, 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 inoculation that is needed for this as a function of the contagiousness of the disease.

42.6 Project writeup and submission

42.6.1 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; section 18.1.2.

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*.

42.6.2 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.

Chapter 43

PageRank

43.1 Basic ideas

We are going to simulate the Internet. In particular, we are going to simulate the *Pagerank* algorithm by which *Google* determines the importance of web pages.

Let's start with some basic classes:

- A `Page` contains some information such as its title and a global numbering in Google's data-center. It also contains a collection of links.
- We represent a link with a pointer to a `Page`. Conceivably we could have a `Link` class, containing further information such as probability of being clicked, or number of times clicked, but for now a pointer will do.
- Ultimately we want to have a class `Web` which contains a number of pages and their links. The `web` object will ultimately also contain information such as relative importance of the pages.

This application is a natural one for using pointers. When you click on a link on a web page you go from looking at one page in your browser to looking at another. You could implement this by having a pointer to a page, and clicking updates the value of this pointer.

Exercise 43.1. 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 43.2. 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?

43.2 Clicking around

Exercise 43.3. 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 43.4. 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 43.5. 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.

43.3 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 43.6. 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.

43.4 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:

make[5]: *** No rule to make target `run_p

For the source of this example, see section ??

Exercise 43.7. 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 43.8. 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 43.9. 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 43.10. Add a page that you will artificially make 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?

43.5 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 43.11. 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.

43.6 Sources used in this chapter

Chapter 44

Redistricting

In this project you can explore ‘gerrymandering’, the strategic drawing of districts to give a minority population a majority of districts¹.

44.1 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.

44.2 Basic functions

44.2.1 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 44.1. 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:

```
make[5]: *** No rule to make target 'run_v
```

44.2.2 Populations

Exercise 44.2. 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:

```
make[5]: *** No rule to make target 'run_d
```

Exercise 44.3. 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:

make[5]: *** No rule to make target 'run_p

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.

44.2.3 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 44.4. 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:

make[5]: *** No rule to make target 'run_g

Exercise 44.5. 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;

```

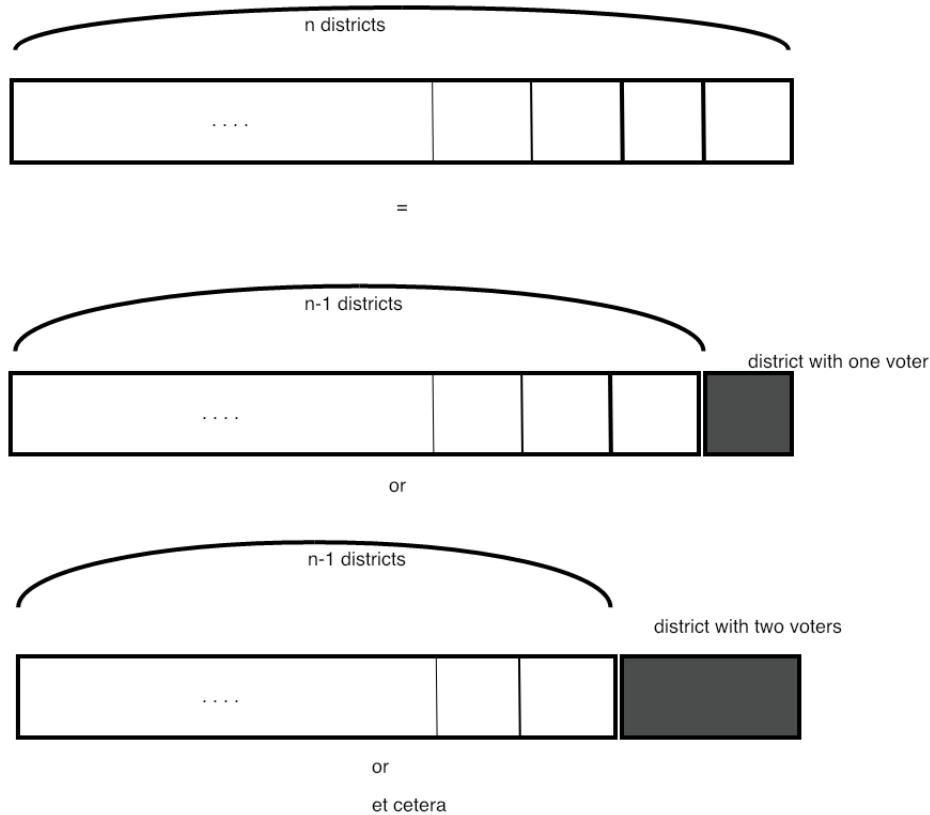


Figure 44.1: Multiple ways of splitting a population

44.3 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 44.1.

- For all $p = 0, \dots, n - 1$ considering splitting the state into $0, \dots, p - 1$ and $p, \dots, n - 1$.
- Use the best districting of the first group, and make the last group into a single district.
- Keep the districting that gives the strongest minority rule, over all values of p .

You can now realize the above simple example:

AAABB => AAA | B | B

Exercise 44.6. 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:

make[5]: *** No rule to make target 'run_d

Note: the range for p given above is not quite correct: for instance, the initial part of the population needs to be big enough to accommodate $n - 1$ voters.

Exercise 44.7. Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

44.4 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 44.8. 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 44.9. Code a dynamic programming solution to the redistricting problem.

44.5 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 44.10. 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 44.11. Look up the definition of efficiency gap (and ‘wasted votes’), and implement it in your code.

Chapter 45

Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

45.1 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 **TSP!** (**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.

45.2 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.

45.2.1 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 45.1. Code a class `Address` with the above functionality, and test it.

Next we need a class `AddressList` that contains a list of addresses.

Exercise 45.2. 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.

45.2.2 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.

45.2.3 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 45.3. 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:

make[5]: *** No rule to make target 'run_s

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each address a boolean field `done`, but you could also make a copy of the input list, and remove the elements that have been processed. For this, study the `erase` method for `vector` objects.

45.3 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.

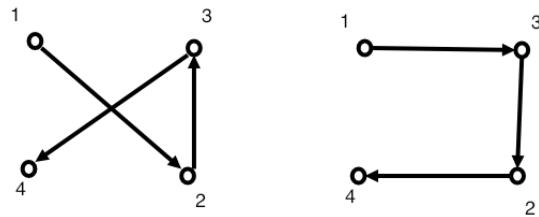


Figure 45.1: Illustration of the ‘opt2’ idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [?], 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 45.1 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 45.4. 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 45.5. What is the runtime complexity of this heuristic solution?

Exercise 45.6. 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.

45.4 Multiple trucks

If we introduce multiple delivery trucks, we get the ‘Multiple Traveling Salesman Problem’ [?]. 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.

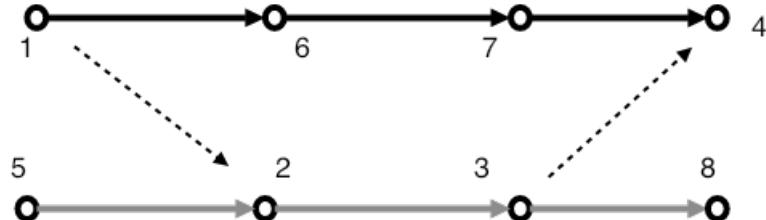
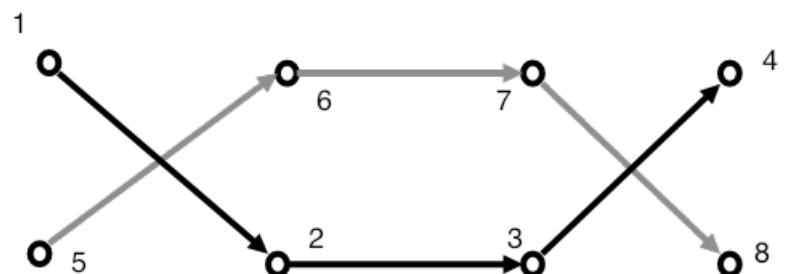


Figure 45.2: Extending the ‘opt2’ idea to multiple paths

Can we extend the opt2 heuristic to the case of multiple paths? For inspiration take a look at figure 45.2: 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 45.7. Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 45.3.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

45.5 Amazon prime

In section ?? you made the assumption that it doesn’t matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

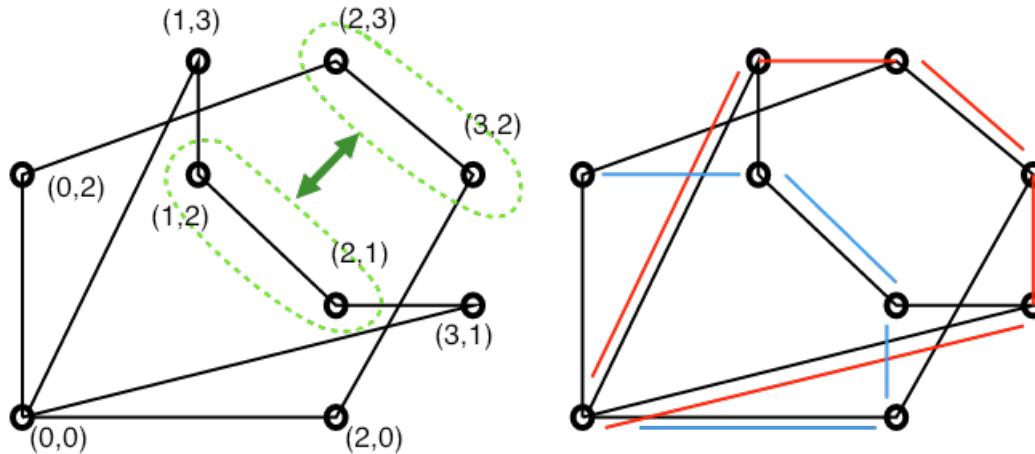


Figure 45.3: Multiple paths test case

Exercise 45.8. 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.

45.6 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 45.9. Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

Chapter 46

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 47

DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

47.1 Basic functions

Refer to section [23.3.1](#).

First we set up some basic mechanisms.

Exercise 47.1. There are four bases, A, C, G, T, and each has a complement: A \leftrightarrow T, C \leftrightarrow G.

Implement this through a map, and write a function

```
char BaseComplement(char);
```

Exercise 47.2. 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.

A ‘read’ is a short fragment of DNA, that we want to match against a genome. 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 47.3. 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?

Chapter 48

Cryptography

48.1 Basic ideas

https://simple.wikipedia.org/wiki/RSA_algorithm

https://simple.wikipedia.org/wiki/Exponentiation_by_squaring

Chapter 49

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 [?].)

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 35): 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.

49.1 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 49.1. 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.

49.2 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 year¹.

This is something we can test.

Exercise 49.2. Make an array `previous_record` of the same shape as `monthly_deviation`.

This array records (for each month, which, remember, we treat like independent measurements) whether that year was a record, or, if not, when the previous record occurred:

$$\text{PrevRec}(m, y) = \begin{cases} y & \text{if } \text{MonDev}(m, y) = \max_{m'} (\text{MonDev}(m', y)) \\ y' & \text{if } \text{MonDev}(m, y) < \text{MonDev}(m, y') \\ & \text{and } \text{MonDev}(m, y') = \max_{m'' < m'} (\text{MonDev}(m'', y)) \end{cases}$$

Again, use array notation. This is also a great place to use the `Where` clause.

Exercise 49.3. 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 49.4. Copy the code from <http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf> and adapt for our purposes: find the best fit for the slope and intercept for a linear function describing the gaps between records.

You'll find that the gaps are decidedly not linearly increasing. So is this negative result the end of the story, or can we do more?

Exercise 49.5. Can you turn this exercise into a test of global warming? Can you interpret the deviations as the sum of a yearly increase in temperature plus a stationary distribution, rather than a stationary distribution by itself?

1. Technically, we are dealing with a uniform distribution of temperatures, which makes the maxima and minima have a beta-distribution.

PART V

ADVANCED TOPICS

Chapter 50

Programming strategies

50.1 A philosophy of programming

Code for the reader, not the writer

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

High level and low level

- 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

Abstraction

- 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.

50.2 Programming: top-down versus bottom up

The exercises in chapter 40 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 40.8.

A problem with top-down programming is that you can not start testing until you have made your way down to the basic bits of code. With bottom-up it's easier to start testing. Which brings us to...

50.2.1 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;
```

50.3 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.

50.4 Documentation

Take a look at Doxygen.

50.5 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.

50.6 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 51

Tiniest of introductions to algorithms and data structures

51.1 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.

51.1.1 Stack

A *stack* is a data structure that is a bit like an array, except that you can only see the last element:

- You can inspect the last element;
- You can remove the last element; and
- You can add a new element that then becomes the last element; the previous last element becomes invisible: it becomes visible again as the last element if the new last element is removed.

The actions of adding and removing the last element are known as *push* and *pop* respectively.

Exercise 51.1. Write a class that implements a stack of integers. It should have methods

```
void push(int value);  
int pop();
```

51.1.2 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
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 `NUL` which is no more than a synonym for the value zero.

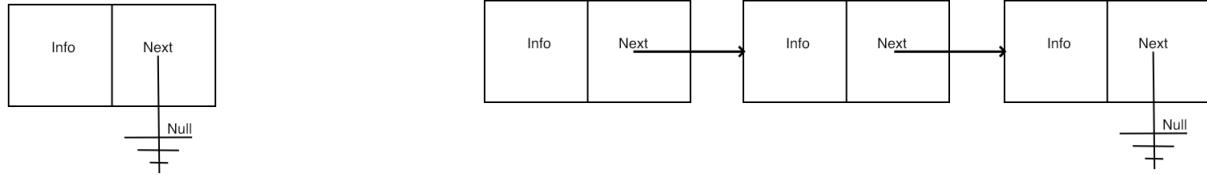


Figure 51.1: Node data structure and linked list of nodes

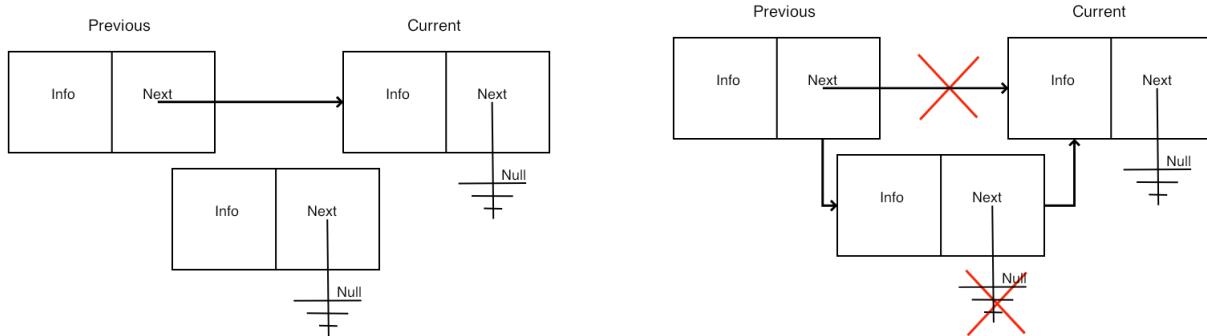


Figure 51.2: Insertion in a linked list

We illustrate this in figure 51.1.

Our main concern will be to implement operations that report some statistic of the list, such as its length, that test for the presence of information in the list, or that alter the list, for instance by inserting a new node. See figure 51.2.

51.1.2.1 Data definitions

In C++ you have a choice of pointer types. Conceptually we can say that the list object owns the first node, and each node owns the next. Therefore we use the `unique_ptr`; however, you can also use `shared_ptr` throughout, at slight overhead cost.

We declare the basic classes.

Definition of List class

A linked list has as its only member a pointer to a node:

```
class List {
private:
    unique_ptr<Node> head{nullptr};
public:
    List() {};
```

Initially null for empty list.

Definition of Node class

A node has information fields, and a link to another node:

```
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; };
    int value() {
        return datavalue;
    }
    int count() {
        return datacount;
    }
    //codesnippet nodelengthrecursive

```

A Null pointer indicates the tail of the list.

51.1.2.2 Simple functions

For many algorithms we have the choice between an iterative and a recursive version. The recursive version is easier to formulate, but the iterative solution is probably more efficient.

Recursive computation of the list length

```

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.

Iterative computation of the list length

Use a *bare pointer*, which is appropriate here because it doesn't own the node.

```

int length() {
    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.

Exercise 51.2. Write a function

```
bool List::contains_value(int v);
```

to test whether a value is present in the list.

Try both recursive and iterative.

51.1.2.3 Modification functions

The interesting methods are of course those that alter the list. Inserting a new value in the list has basically two cases:

1. If the list is empty, create a new node, and set the head of the list to that node.
2. If the list is not empty, we have several more cases, depending on whether the value goes at the head of the list, the tail, somewhere in the middle. And we need to check whether the value is already in the list.

Our choice of using unique pointers dictates a certain design.

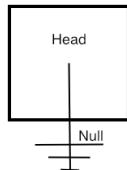
Insert routine design

We will write functions

```
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.

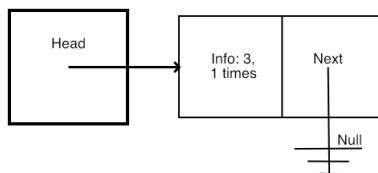
There are a lot of cases here. You can try this by an approach called Test-Driven Development (TDD): first you decide on a test, then you write the code that covers that case.



Step 1: dealing with an empty list

Exercise 51.3. Write a `List::length` method, so that this code gives the right output:

```
|| List mylist;
|| cout << "Empty list has length: "
||     << mylist.length() << endl;
|| cout << endl;
```



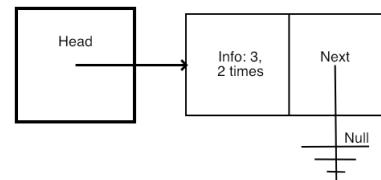
Step 2: insert the first element

Exercise 51.4. 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.length() << 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;

```



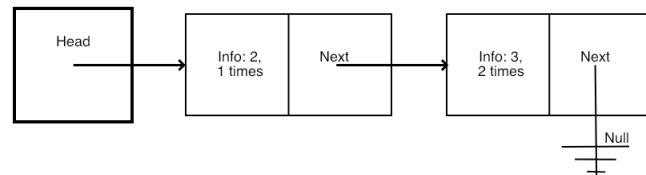
Step 3: inserting an element that already exists

Exercise 51.5. 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.length() << 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 51.6. One of the remaining cases is inserting an element that goes at the head. Update your insert method to get this to work:

```

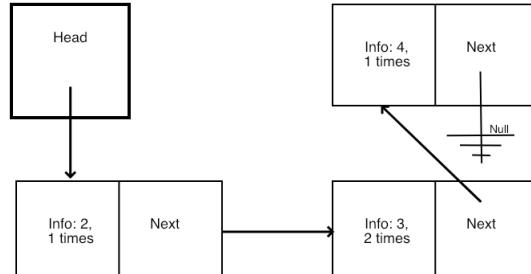
mylist.insert(2);
cout << "Inserting 2 goes at the head; now the length is: "
    << mylist.length() << endl;
if (mylist.contains_value(2))
    cout << "Indeed: contains 2" << endl;
else
    cout << "Hm. Should contain 2" << endl;
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << endl;

```

```

    || else
    ||| cout << "Hm. Should contain 3" << endl;
    ||| cout << endl;

```



Step 5: inserting an element at the end

Exercise 51.7. 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.length() << 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;

```

51.1.3 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
};
```

51.2 Algorithms

This *really really* goes beyond this book.

- Simple ones: numerical
- Connected to a data structure: search

51.2.1 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.

51.2.1.1 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 52.1.1) 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.

51.2.1.2 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 51.8. 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?

51.3 Programming techniques

51.3.1 Memoization

In section 7.6 you saw some examples of recursion. The factorial example could be written in a loop, and there are both arguments for and against doing so.

The Fibonacci example is more subtle: it can not immediately be converted to an iterative formulation, but there is a clear need for eliminating some waste that comes with the simple recursive formulation. The technique we can use for this is known as *memoization*: store intermediate results to prevent them from being recomputed.

Here is an outline.

```
int fibonacci(int n) {
    vector<int> fibo_values(n);
    for (int i=0; i<n; i++)
        fibo_values[i] = 0;
    fibonacci_memoized(fibo_values, n-1);
    return fibo_values[n-1];
}
int fibonacci_memoized( vector<int> &values, int top ) {
    int minus1 = top-1, minus2 = top-2;
    if (top<2)
        return 1;
    if (values[minus1]==0)
        values[minus1] = fibonacci_memoized(values, minus1);
    if (values[minus2]==0)
        values[minus2] = fibonacci_memoized(values, minus2);
    values[top] = values[minus1]+values[minus2];
    //cout << "set f(" << top << ") to " << values[top] << endl;
    return values[top];
}
```


Chapter 52

Complexity

52.1 Order of complexity

52.1.1 Time complexity

Exercise 52.1. 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 $1 \dots n$. You can have a solution that keeps a running sum, and a solution with an inner loop.

Exercise 52.2. How many operations, as a function of n , are performed in these two solutions?

52.1.2 Space complexity

Exercise 52.3. 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 52.4. How much space do the two solutions require?

PART VI

INDEX AND SUCH

Chapter 53

Index

#ifndef, 235
#ifdef, 235
#ifndef, 235
#once, 235
#pack, 235
__FILE__, 244
__LINE__, 244

abs, 46
abstraction, 65
algorithm, 46, 112, 252, 269
all (Fortran keyword), 340
all_of, 252
allocate, 338
Amazon
 delivery truck, 391
 prime, 391, 394
any (Fortran keyword), 341
any_of, 252, 269
Apple, 27
argument
 actual, 313
 default, 77
 dummy, 313
array, 127
 associative, 248
 assumed-shape, 338
 automatic, 333, 338
 bounds
 checking, 128
 index, 128
 initialization, 128, 334
 rank, 335
 static, 333
 subscript, 128

array, 142
assignment, 40
Associated (Fortran keyword), 347
asterisk
 in Fortran formatted I/O, 352
auto, 129, 276
auto_ptr, 276

bad_alloc, 244
bad_exception, 244
basic_ios, 170
begin, 265
bit_size, 293
bottom-up, 407
break, 59
bubble sort, 143, 418
bug, 20

C
 parameter passing, 205–208
 pointer, 201–208, 284
 preprocessor, 289
 string, 158, 284
C preprocessor, see preprocessor
C++, 275
 20, 244
 C++11, 276
 C++14, 47, 276
 C++17, 37, 251, 277
 C++20, 142, 277
 Core Guidelines, 410
 standard, 142
c_sizeof, 293
Caesar cypher, 157
calendars, 277

call, 310
call-back, 273
calling environment, 179
capture, 268
case sensitive, 39
cast, 46, 270
cerr, 168, 283
char, 155
Character (Fortran keyword), 292
cin, 283
clang++, 35
class, 99
 abstract, 108
 base, 107
 definition, 99
 derived, 107
 iteratable, 265
class, 330
Close (Fortran keyword), 351, 354
closure, 85
cmath, 44, 46, 268
code
 duplication, 68
 maintainance, 409
code duplication, 65
code reuse, 68
Collatz conjecture, 62
column-major, 335, 352
Common (Fortran keyword), 308
compilation
 separate, 224, 325, 378
compiler, 28, 35
 and preprocessor, 233
 one pass, 67
compiling, 28
Complex (Fortran keyword), 292
complex numbers, 41, 247
concepts, 277
conditional, 49
connected components, see graph, connected
const
 reference, 213
const, 213, 215
const_cast, 272
constexpr, 276
constructor, 100, 187, 284
copy, 112, 214
range, 283
container, 247
Contains (Fortran keyword), 308, 309
contains
 for class functions, 329
 in modules, 326
contains, 315
continuation character, 291
continue, 60
coroutines, 277
cout, 43, 283
Cshift, 339
datatype, 39
deallocate, 338
debugger, 245
definition vs use, 83
delete, 276
dereference, 202
derefencing, 139
destructor, 87, 114
 at end of scope, 86
dimension (Fortran keyword), 333
do
 concurrent, 61
do (Fortran keyword), 303
do concurrent, 341
do loop
 implicit, 352
 and array initialization, 334
 implied, 304
Dot_Product, 339
dynamic
 programming, 389
dynamic_cast, 270
efficiency gap, 390
Eigen, 137
emacs, 27, 27, 37, 290
end, 265, 290
endl, 168
EOF, 169
eof, 169
erase, 249
errno, 244
error

compile-time, 38
run-time, 38
syntax, 38
exception
 catch, 242
 throwing, 242
exception, 244
executable, 28, 36
exit, 304
expression, 40
extent
 of array dimension, 337
external, 315

F90, 289
false, 42, 44
Fasta, 399
Fastq, 400
file
 binary, 35, 38
 executable, 224
 handle, 114
 object, 224, 326
 source, 35
floating point, 42
flush, 168
fmtlib, 283
for
 indexed, 130
 range-based, 129
for_each, 252
forall, 341
Format (Fortran keyword), 351
Fortran
 90, 289
 case ignores, 290
 comments, 291
forward declaration, 223
 of classes, 85
 of functions, 85
friend, 109
function, 65, 311, 311
 argument, 69
 arguments, 69
 body, 69
 call, 66, 67
 defines scope, 70
 definition, 66
 header, 67
 parameter, 69
 parameters, 69
 prototype, 67
 result type, 69
 standard, 69
 function, 268
 function try block, 243
functional programming, 71, 179
functor, 112

g++, 35
gdb, 245
gerrymandering, 385
get, 191
getline, 169
gfortran, 290
GNU, 35
Goldbach conjecture, 367
Google, 381
graph
 connected, 382
 diameter, 383
greedy search, see search, greedy

has-a relation, 106
has_value, 252
hdf5, 351
header, 378
header file, 225, 233
 and global variables, 227
 treatment by preprocessor, 227
 vs modules, 277
heap, 142
hexadecimal, 201
Holmes
 Sherlock, 157
homebrew, 27
host association, 308

I/O
 formatted, 351
 list-directed, 351
 unformatted, 351
icpc, 35

icpc, 35
if (Fortran keyword), 299
ifdef, see #pragma ifdef
ifndef, see #pragma ifndef
ifort, 290
index
 section, 334
Inf, 47, 253
inheritance, 107
initialization
 variable, 39
initializer list, 102, 128, 129
inline, 314
insert, 249
int, 42
Integer (Fortran keyword), 292
Interface (Fortran keyword), 309
interface, 312, 315, 315
is-a relation, 107
is_eof, 170
is_open, 170
isinf, 253
isnan, 243, 253
iso_c_binding, 293
iteration
 of a loop, see loop, iteration
iterator, 139, 248, 276
iterator, 248

keywords, 38
kind, 292, 293

label, 353
lambda, see closure, 273
 expression, 267
 make mutable, 269
lamda
 const by default, 269
Lbound, 337
len (Fortran keyword), 321
lexicographic ordering, 58
limits, 253
limits.h, 253
linear regression, 404
linker, 224, 326
Linux, 27
list
linked, 411–416
 in Fortran, 347–349
logging, 168
Logical (Fortran keyword), 292, 300
long, 254
long long, 254
longjmp, 244
loop, 55
 body, 55
 counter, 55
 for, 55
 header, 55
 inner, 58
 iteration, 55
 nest, 58
 outer, 58
 variable, 56
 while, 55
lvalue, 273

macports, 27
Make, 224
makefile, 225, 378
malloc, 142, 187, 204, 207, 208, 283, 284
Manhattan distance, 391
map, 248
MatMul, 339
max, 46
MAX_INT, 253
MaxLoc (Fortran keyword), 339
MaxVal, 339
member
 initializer list, 100
 of struct, 91
members, see object, members
memoization, 389, 419
memory
 leak, 114, 142, 206, 206, 339
memory leak, 190
memory leaking, 208
method, 102
 abstract, 108
 overriding, 107
methods, see object, methods
Microsoft
 Windows, 27

Word, 18
MinLoc, 339
MinVal, 339
Module (Fortran keyword), 308, 309
modules, 277
move semantics, 274
mutable, 269

namespace, 229
NaN, 47, 253
new, 136, 188, 192, 206, 208, 276
Newton's method, 72
noexcept, 244
NP-hard, 393
NULL, 193
NULL, 158, 411
null terminator, 284
Nullify (Fortran keyword), 347
nullptr, 187, 193, 270, 411
nullptr_t, 193
numeric_limits, 253

object
 members, 99
 methods, 99
object file, *see* file, object
once, *see* #pragma once
Open (Fortran keyword), 351
open, 169
open (Fortran keyword), 354
OpenFrameworks, 275
operator
 overloading, 111
 and copies, 275
 of parentheses, 112
 shortcut, 45
 spaceship, 277
opt2, 393
Optional (Fortran keyword), 315
optional, 252
output
 binary, 355
 raw, 355
 unformatted, 355
override, 108

pack, *see* #pragma pack

package manager, 27
Pagerank, 381
parameter, *see also* function, parameter
 formal, 314
 input, 73
 output, 73, 250
 pass by value, 71
 passing, 71
 by reference, 179, 187, 284
 by value, 179
 passing by reference, 71, 73
 in C, 73
 passing by value, 71
 throughput, 73
Pascal's triangle, 143
pass by reference, *see* parameter, passing by reference
pass by value, *see* parameter, passing by value
PETSc, 244
pointer, 115
 arithmetic, 139, 204
 bare, 192, 284, 413
 dereference, 249
 derefencing, 345
 null, 193, 411
 smart, 142
 unique, 192
 void, 272
 in C++, 193
pointer, 345
polymorphism, 111
pop, 411
precedence, 51
preprocessor
 and header files, 227
 conditionals, 234–235
macro
 parametrized, 234
macros, 233–234
Present (Fortran keyword), 315
Print (Fortran keyword), 351
print, 351
printf, 283
private, 100, 102, 104, 326
procedure
 internal, 308

procedure, 330
procedures
 internal, 314
 module, 314
Product, 339
program
 statements, 36
programming
 dynamic, 388
 parallel, 61
protected, 107, 326
prototype, 223
public, 100, 104, 326
punch cards, 289
push, 411
push_back, 132
putty, 27
python, 24

quicksort, 418

RAII, 284
rand, 77
RAND_MAX, 77
random
 seed, 357
random number
 generator, 77, 254
 Fortran, 357
 seed, 77
random_number (Fortran keyword), 357
random_seed (Fortran keyword), 357
range-based for loop, see for, range-based
ranges, 277
rbegin, 264
Read (Fortran keyword), 351
Real (Fortran keyword), 292
recurrence, 342
recursion, see function, recursive
 depth, 76
 mutual, 76
Recursive (Fortran keyword), 311
reduction, 249
reference, 72, 179
 argument, 187, 284
 const, 134, 179, 183
 to class member, 180

 to class member, 180
reference count, 192
reinterpret_cast, 270
rend, 264
reserved words, 39
RESHAPE (Fortran keyword), 337
result (Fortran keyword), 311
return, 69
 code, 38
 makes copy, 183
 statement, 38
return, 310
root finding, 62
roundoff error analysis, 47
runtime error, 241
rvalue, 273
 reference, 275

Save (Fortran keyword), 308
scanf, 283
scope, 69, 83
 dynamic, 86
 in conditional branches, 52
 lexical, 83, 86
 of function body, 70
search
 greedy, 392, 393
segmentation fault, 128
select (Fortran keyword), 300
selected_int_kind, 292
selected_real_kind, 292
setjmp, 244
shape
 of array, 337
shared_ptr, 284, 412
shell
 inspect return code, 38
Single Source Shortest Path, 383
SIR model, 378
size, 337
size_t, 127, 270
sizeof, 207
smartphone, 19
source
 format
 fixed, 289

free, 289
source code, 28
source_location, 244
span, 142, 277
SPREAD (Fortran keyword), 337
srand, 77
stack, 76, 87, 142, 338, 411
 overflow, 76, 142, 338
Standard Template Library, 247
statement functions, 314
static_cast, 270, 272
std::any, 193
stop (Fortran keyword), 290
storage_size, 293
string, 156
 concatenation, 156
 null-terminated, 158
 size, 156
string, 284
struct
 denotation, 94
struct, 91, 251
subprogram, see function
Sum, 339
swap, 275
switch, 51
syntax
 error, 241
system_clock (Fortran keyword), 357
target, 345
templates
 and separate compilation, 226
test-driven development, 410
testing, 410
text
 formatting, 277
this, 115, 193
time zones, 277
timer
 resolution, 357
top-down, 407
Transpose, 339
trim (Fortran keyword), 321
true, 42, 44
tuple, 251
 denotation, 251
type (Fortran keyword), 323
Ubound, 337
unique_ptr, 192, 284, 412
unit, 354
unit testing, 410
Unix, 27
use, 326
use (Fortran keyword), 325
valgrind, 245
value, 252
values
 boolean, 42
variable, 38
 assignment, 39
 declaration, 39, 39
 global, 227
 in header file, 227
 initialization, 43
 lifetime, 84
 numerical, 41
 shadowing, 84
 static, 86, 307
vector, 229
 methods, 132
 subvector, 250
vector, 127, 131, 142, 250, 283, 411
vi, 27
Virtualbox, 27
Visual Studio, 27
VMware, 27
void, 66, 69, 70
where (Fortran keyword), 340
while, 60
Write (Fortran keyword), 351
write (Fortran keyword), 354
Xcode, 27
XQuartz, 27