

# Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

2018



# Contents

I	Introduction	13
1	<b>Introduction</b>	15
1.1	<i>Programming and computational thinking</i>	15
1.1.1	History	15
1.1.2	Is programming science, art, or craft?	17
1.1.3	Computational thinking	18
1.1.4	Hardware	20
1.1.5	Algorithms	20
1.2	<i>About the choice of language</i>	20
1.3	<i>Further reading</i>	21
2	<b>Warming up</b>	23
2.1	<i>Programming environment</i>	23
2.1.1	Language support in your editor	23
2.2	<i>Compiling</i>	24
3	<b>Teachers guide</b>	25
3.1	<i>Justification</i>	25
3.2	<i>Timeline for a C++/F03 course</i>	25
3.2.1	Project-based teaching	26
3.2.2	Choice: Fortran or advanced topics	27
II	<b>C++</b>	29
4	<b>Basic elements of C++</b>	31
4.1	<i>From the ground up: Compiling C++</i>	31
4.1.1	A quick word about unix commands	32
4.2	<i>Statements</i>	33
4.3	<i>Variables</i>	34
4.3.1	Variable declarations	34
4.3.2	Datatypes	35
4.3.3	Assignments	35
4.3.4	Floating point variables	37
4.3.5	Number values and undefined values	38
4.3.6	Boolean values	39
4.3.7	Strings	39
4.4	<i>Input/Output, or I/O as we say</i>	39

4.5	<i>Expressions</i>	40
4.5.1	Truth values	40
4.5.2	Type conversions	41
4.6	<i>Library functions</i>	42
4.7	<i>Review questions</i>	43
5	<b>Conditionals</b>	45
5.1	<i>Conditionals</i>	45
5.2	<i>Operators</i>	46
5.3	<i>Switch statement</i>	48
5.4	<i>Scopes</i>	48
5.5	<i>Sources used in this chapter</i>	48
6	<b>Looping</b>	51
6.1	<i>The ‘for’ loop</i>	51
6.2	<i>Looping until</i>	55
6.3	<i>Exercises</i>	57
6.4	<i>Sources used in this chapter</i>	58
7	<b>Functions</b>	61
7.1	<i>Function definition and call</i>	62
7.2	<i>Why use functions?</i>	63
7.3	<i>Anatomy of a function definition and call</i>	64
7.4	<i>Parameter passing</i>	65
7.4.1	Pass by value	66
7.4.2	Pass by reference	67
7.5	<i>Recursive functions</i>	69
7.5.1	Stack overflow	71
7.6	<i>More function topics</i>	72
7.6.1	Default arguments	72
7.6.2	Polymorphic functions	72
7.7	<i>Library functions</i>	72
7.7.1	Random function	72
7.8	<i>Review questions</i>	73
7.9	<i>Sources used in this chapter</i>	74
8	<b>Scope</b>	79
8.1	<i>Scope rules</i>	79
8.1.1	Lexical scope	79
8.1.2	Shadowing	79
8.1.3	Lifetime versus reachability	80
8.1.4	Scope subtleties	81
8.2	<i>Static variables</i>	81
8.3	<i>Scope and memory</i>	82
8.4	<i>Review questions</i>	83
8.5	<i>Sources used in this chapter</i>	84
9	<b>Structures</b>	87
9.1	<i>Why structures?</i>	87
9.2	<i>The basics of structures</i>	87
9.3	<i>Sources used in this chapter</i>	91

10	<b>Classes and objects</b>	95
10.1	<i>What is an object?</i>	95
10.1.1	Constructor	96
10.1.2	Public and private	96
10.1.3	Initialization	97
10.1.4	Methods	98
10.1.5	Default constructor	100
10.1.6	Accessors	101
10.1.7	Examples	101
10.1.8	Advanced topics	101
10.2	<i>Inclusion relations between classes</i>	106
10.2.1	Accessors and other methods	106
10.3	<i>Inheritance</i>	107
10.3.1	Methods of base and derived classes	108
10.3.2	Virtual methods	109
10.3.3	Advanced topics in inheritance	110
10.4	<i>Review question</i>	111
10.5	<i>Sources used in this chapter</i>	111
11	<b>Arrays</b>	121
11.1	<i>Introduction</i>	121
11.1.1	Initialization	121
11.1.2	Ranging over an array	122
11.1.3	Vector are a class	123
11.1.4	Vector methods	126
11.2	<i>Vectors are dynamic</i>	126
11.3	<i>Vectors and functions</i>	127
11.3.1	Pass vector to function	127
11.3.2	Vector as function return	128
11.4	<i>Vectors in classes</i>	129
11.4.1	Dynamic size of vector	130
11.4.2	Timing	130
11.5	<i>Wrapping a vector in an object</i>	131
11.6	<i>Multi-dimensional cases</i>	132
11.6.1	Matrix as vector of vectors	132
11.6.2	A better matrix class	133
11.7	<i>Static arrays</i>	133
11.8	<i>Advanced topics</i>	134
11.8.1	Iterators	134
11.8.2	Old-style arrays	135
11.9	<i>Exercises</i>	137
11.10	<i>Sources used in this chapter</i>	138
12	<b>Strings</b>	149
12.1	<i>Characters</i>	149
12.2	<i>Basic string stuff</i>	149
12.3	<i>Conversion</i>	152
12.4	<i>C strings</i>	152

12.5	<i>Sources used in this chapter</i>	152
13	<b>Input/output</b>	157
13.1	<i>Formatted output</i>	157
13.2	<i>Floating point output</i>	159
13.3	<i>Saving and restoring settings</i>	161
13.4	<i>File output</i>	161
13.4.1	Output your own classes	161
13.5	<i>Binary output</i>	162
13.6	<i>Input</i>	162
13.6.1	<i>File input</i>	163
13.6.2	<i>Input streams</i>	164
13.7	<i>Sources used in this chapter</i>	164
14	<b>References</b>	173
14.1	<i>Reference</i>	173
14.2	<i>Pass by reference</i>	173
14.3	<i>Reference to class members</i>	174
14.4	<i>Reference to array members</i>	176
14.5	<i>rvalue references</i>	177
14.6	<i>Sources used in this chapter</i>	177
15	<b>Pointers</b>	181
15.1	<i>The ‘arrow’ notation</i>	181
15.2	<i>Making a shared pointer</i>	182
15.2.1	<i>Pointers and arrays</i>	182
15.2.2	<i>Smart pointers versus address pointers</i>	183
15.3	<i>Garbage collection</i>	184
15.4	<i>More about pointers</i>	185
15.4.1	<i>Get the pointed data</i>	185
15.4.2	<i>Example: linked lists</i>	185
15.5	<i>Advanced topics</i>	186
15.5.1	<i>Unique pointers</i>	186
15.5.2	<i>Shared pointer to ‘this’</i>	186
15.5.3	<i>Null pointer</i>	186
15.5.4	<i>Void pointer</i>	187
15.5.5	<i>Pointers to non-objects</i>	187
15.6	<i>Sources used in this chapter</i>	187
16	<b>C-style pointers and arrays</b>	193
16.1	<i>What is a pointer</i>	193
16.2	<i>Pointers and addresses, C style</i>	193
16.3	<i>Arrays and pointers</i>	195
16.4	<i>Pointer arithmetic</i>	196
16.5	<i>Multi-dimensional arrays</i>	197
16.6	<i>Parameter passing</i>	197
16.6.1	<i>Allocation</i>	198
16.6.2	<i>Use of new</i>	200
16.7	<i>Memory leaks</i>	201
16.8	<i>Sources used in this chapter</i>	201

17	<b>Prototypes</b>	205
17.1	<i>Prototypes for functions</i>	205
17.1.1	Separate compilation	206
17.1.2	Header files	206
17.1.3	C and C++ headers	207
17.2	<i>Prototypes for class methods</i>	208
17.3	<i>Header files and templates</i>	208
17.4	<i>Namespaces and header files</i>	208
17.5	<i>Global variables and header files</i>	208
18	<b>Namespaces</b>	211
18.1	<i>Solving name conflicts</i>	211
18.1.1	Namespace header files	212
18.2	<i>Best practices</i>	213
19	<b>Preprocessor</b>	215
19.1	<i>Textual substitution</i>	215
19.2	<i>Parametrized macros</i>	216
19.3	<i>Conditionals</i>	216
19.3.1	Check on a value	217
19.3.2	Check for macros	217
19.3.3	Including a file only once	217
20	<b>Templates</b>	219
20.1	<i>Templated functions</i>	219
20.2	<i>Templated classes</i>	220
20.3	<i>Specific implementation</i>	220
20.4	<i>Templating over non-types</i>	220
20.5	<i>Sources used in this chapter</i>	221
21	<b>Error handling</b>	223
21.1	<i>General discussion</i>	223
21.2	<i>Exception handling</i>	224
21.3	<i>Legacy mechanisms</i>	226
21.3.1	Legacy C mechanisms	226
21.4	<i>Tools</i>	226
22	<b>Standard Template Library</b>	229
22.1	<i>Complex numbers</i>	229
22.2	<i>Containers</i>	229
22.2.1	Maps: associative arrays	230
22.2.2	Iterators	230
22.3	<i>Tuples</i>	230
22.4	<i>Random numbers</i>	232
22.5	<i>Time</i>	232
22.6	<i>Sources used in this chapter</i>	232
23	<b>Obscure stuff</b>	235
23.1	<i>Const</i>	235
23.1.1	Const arguments	235
23.1.2	Const references	235
23.1.3	Const methods	237

23.2	<i>Auto</i>	238
23.2.1	<i>Declarations</i>	238
23.2.2	<i>Iterating</i>	239
23.3	<i>Iterating over classes</i>	239
23.4	<i>Lambdas</i>	242
23.5	<i>Casts</i>	243
23.5.1	<i>Static cast</i>	244
23.5.2	<i>Dynamic cast</i>	245
23.5.3	<i>Const cast</i>	246
23.5.4	<i>A word about void pointers</i>	246
23.6	<i>Ivalue vs rvalue</i>	247
23.6.1	<i>Conversion</i>	248
23.6.2	<i>References</i>	248
23.6.3	<i>Rvalue references</i>	249
23.7	<i>Move semantics</i>	249
23.8	<i>Sources used in this chapter</i>	250
24	<b>C++ for C programmers</b>	257
24.1	<i>I/O</i>	257
24.2	<i>Arrays</i>	257
24.3	<i>Strings</i>	257
24.4	<i>Pointers</i>	257
24.4.1	<i>Parameter passing</i>	258
24.5	<i>Objects</i>	258
24.6	<i>Namespaces</i>	258
24.7	<i>Templates</i>	258
24.8	<i>Obscure stuff</i>	258
24.8.1	<i>Lambda</i>	258
24.8.2	<i>Const</i>	258
24.8.3	<i>Lvalue and rvalue</i>	258
III Fortran 259		
25	<b>Basics of Fortran</b>	261
25.1	<i>Source format</i>	261
25.2	<i>Compiling Fortran</i>	262
25.3	<i>Main program</i>	262
25.3.1	<i>Program structure</i>	262
25.3.2	<i>Statements</i>	263
25.3.3	<i>Comments</i>	263
25.4	<i>Variables</i>	263
25.4.1	<i>Declarations</i>	264
25.4.2	<i>Precision</i>	264
25.4.3	<i>Initialization</i>	265
25.5	<i>Input/Output, or I/O as we say</i>	266
25.6	<i>Expressions</i>	266
25.7	<i>Review questions</i>	267

25.8	<i>Sources used in this chapter</i>	267
26	<b>Conditionals</b>	269
26.1	<i>Forms of the conditional statement</i>	269
26.2	<i>Operators</i>	269
26.3	<i>Select statement</i>	270
26.4	<i>Boolean variables</i>	270
26.5	<i>Review questions</i>	271
27	<b>Loop constructs</b>	273
27.1	<i>Loop types</i>	273
27.2	<i>Interruptions of the control flow</i>	274
27.3	<i>Implied do-loops</i>	274
27.4	<i>Review questions</i>	275
28	<b>Scope</b>	277
28.1	<i>Scope</i>	277
28.1.1	<i>Variables local to a program unit</i>	277
28.1.2	<i>Variables in an internal procedure</i>	278
29	<b>Subprograms and modules</b>	279
29.1	<i>Procedures</i>	279
29.1.1	<i>Subroutines and functions</i>	279
29.1.2	<i>Return results</i>	281
29.1.3	<i>Arguments</i>	284
29.1.4	<i>Types of procedures</i>	285
29.1.5	<i>More about arguments</i>	285
29.2	<i>Interfaces</i>	285
29.2.1	<i>Polymorphism</i>	286
29.3	<i>Sources used in this chapter</i>	286
30	<b>String handling</b>	291
30.1	<i>String denotations</i>	291
30.2	<i>Characters</i>	291
30.3	<i>Strings</i>	291
30.4	<i>Strings versus character arrays</i>	292
30.5	<i>Sources used in this chapter</i>	292
31	<b>Structures, eh, types</b>	293
32	<b>Modules</b>	295
32.1	<i>Modules for program modularization</i>	296
32.2	<i>Modules</i>	296
32.2.1	<i>Polymorphism</i>	297
32.2.2	<i>Operator overloading</i>	298
33	<b>Classes and objects</b>	299
33.1	<i>Classes</i>	299
34	<b>Arrays</b>	303
34.1	<i>Static arrays</i>	303
34.1.1	<i>Initialization</i>	304
34.1.2	<i>Array sections</i>	304
34.1.3	<i>Integer arrays as indices</i>	305
34.2	<i>Multi-dimensional</i>	305

34.2.1	<i>Querying an array</i>	307
34.2.2	<i>Reshaping</i>	307
34.3	<i>Arrays to subroutines</i>	307
34.4	<i>Allocatable arrays</i>	308
34.5	<i>Array output</i>	309
34.6	<i>Operating on an array</i>	309
34.6.1	<i>Arithmetic operations</i>	309
34.6.2	<i>Intrinsic functions</i>	309
34.6.3	<i>Restricting with where</i>	310
34.6.4	<i>Global condition tests</i>	311
34.7	<i>Array operations</i>	311
34.7.1	<i>Loops without looping</i>	311
34.7.2	<i>Loops without dependencies</i>	312
34.7.3	<i>Loops with dependencies</i>	313
34.8	<i>Review questions</i>	314
34.9	<i>Sources used in this chapter</i>	314
35	<b>Pointers</b>	317
35.1	<i>Basic pointer operations</i>	317
35.2	<i>Pointers and arrays</i>	319
35.3	<i>Example: linked lists</i>	319
35.4	<i>Sources used in this chapter</i>	322
36	<b>Input/output</b>	323
36.1	<i>Types of I/O</i>	323
36.2	<i>Print to terminal</i>	323
36.2.1	<i>Print on one line</i>	324
36.2.2	<i>Printing arrays</i>	324
36.2.3	<i>Formats</i>	324
36.3	<i>File and stream I/O</i>	325
36.3.1	<i>Units</i>	326
36.3.2	<i>Other write options</i>	326
36.4	<i>Unformatted output</i>	326
36.5	<i>Sources used in this chapter</i>	326
37	<b>Leftover topics</b>	329
37.1	<i>Timing</i>	329
IV	<b>Exercises and projects</b>	331
38	<b>Exercises</b>	333
38.1	<i>Arithmetic</i>	333
38.2	<i>Scope</i>	333
38.3	<i>Looping</i>	334
38.4	<i>Subprograms</i>	334
38.5	<i>Object oriented exercises</i>	335
38.6	<i>List access</i>	336
39	<b>Prime numbers</b>	337
39.1	<i>Arithmetic</i>	337

39.2	<i>Conditionals</i>	337
39.3	<i>Looping</i>	337
39.4	<i>Functions</i>	338
39.5	<i>While loops</i>	338
39.6	<i>Structures</i>	338
39.7	<i>Classes and objects</i>	339
39.8	<i>Eratosthenes sieve</i>	340
39.8.1	Arrays implementation	340
39.8.2	Streams implementation	340
39.9	<i>Range implementation</i>	341
40	<b>Geometry</b>	343
40.1	<i>Basic functions</i>	343
40.2	<i>Point class</i>	343
40.3	<i>Using one class in another</i>	344
40.4	<i>Is-a relationship</i>	346
40.5	<i>More stuff</i>	346
41	<b>Infectuous disease simulation</b>	347
41.1	<i>Model design</i>	347
41.1.1	Other ways of modeling	347
41.2	<i>Coding up the basics</i>	348
41.3	<i>Population</i>	349
41.4	<i>Contagion</i>	349
41.5	<i>Spreading</i>	350
41.6	<i>Project writeup and submission</i>	350
41.6.1	<i>Program files</i>	350
41.6.2	<i>Writeup</i>	351
42	<b>PageRank</b>	353
42.1	<i>Basic ideas</i>	353
42.2	<i>Clicking around</i>	353
42.3	<i>Graph algorithms</i>	354
42.4	<i>Page ranking</i>	354
42.5	<i>Graphs and linear algebra</i>	355
42.6	<i>Sources used in this chapter</i>	355
43	<b>Climate change</b>	357
43.1	<i>Reading the data</i>	357
43.2	<i>Statistical hypothesis</i>	357
44	<b>Redistricting</b>	359
44.1	<i>Basic concepts</i>	359
44.2	<i>Basic functions</i>	360
44.2.1	<i>Voters</i>	360
44.2.2	<i>Populations</i>	360
44.2.3	<i>Districting</i>	360
44.3	<i>Strategy</i>	361
44.4	<i>Efficiency: dynamic programming</i>	361
44.5	<i>Extensions</i>	362
45	<b>DNA Sequencing</b>	363

45.1 *Basic functions* 363

V Advanced topics 365

46 **Programming strategies** 367

46.1 *A philosophy of programming* 367

46.2 *Programming: top-down versus bottom up* 367

46.2.1 Worked out example 368

46.3 *Coding style* 369

46.4 *Documentation* 369

46.5 *Testing* 369

46.6 *Best practices: C++ Core Guidelines* 370

47 **Tiniest of introductions to algorithms and data structures** 371

47.1 *Data structures* 371

47.1.1 Stack 371

47.1.2 Linked lists 371

47.1.3 Trees 374

47.2 *Algorithms* 376

47.2.1 Sorting 376

47.3 *Programming techniques* 377

47.3.1 Memoization 377

48 **Complexity** 379

48.1 *Order of complexity* 379

48.1.1 Time complexity 379

48.1.2 Space complexity 379

VI Index and such 381

49 **Index** 383

## **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

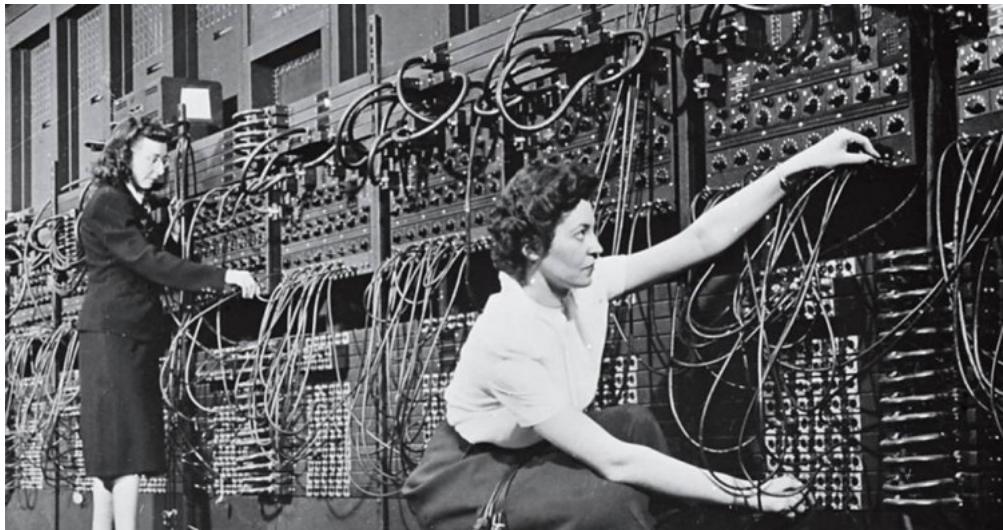


*Hands-on programming*

Very early computers were hardwired

## 1. Introduction

---



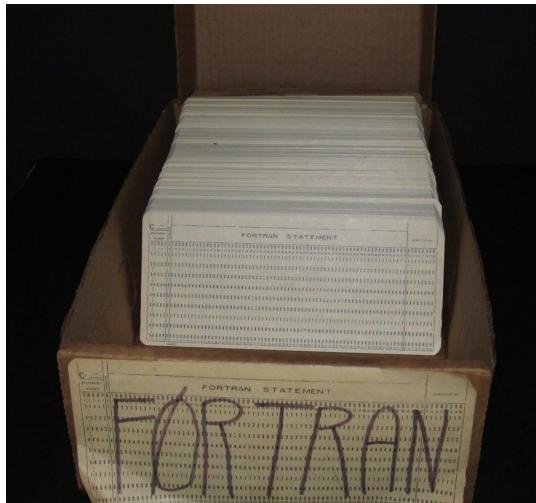
*Program entry*

Later programs were written on punchcards



*The first programming language*

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



*Programming is everywhere*

Programming is used in many different ways these days.

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

This course is aimed at people in the last category.

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

In the early days of computing, hardware design was seen as challenging, while programming was little more than data entry. The fact that Fortran stands for ‘formula translation’ speaks of this: once you have the math, programming is nothing more than translating the math into code. The fact that programs could have subtle errors, or *bugs*, came as quite a surprise.

The fact that programming was not as highly valued also had the side-effect that many of the early programmers were women. Two famous examples were Navy Rear-admiral Grace Hopper, inventor of the Cobol language, and Margaret Hamilton who led the development of the Apollo program software. This situation changed after the 1960s and certainly with the advent of PCs<sup>1</sup>.

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

### 1.1.3 Computational thinking

*Programming is not simple*

Programs can get pretty big:



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



It's not just translating formulas anymore.

Translating ideas to computer code: computational thinking.

*Computational thinking: elevator scheduling*

Mathematical thinking:

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

Sufficient condition  $\neq$  existence proof.

Computational thinking: actual design of a solution

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

Coming up with a strategy takes creativity!

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

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

*Data abstraction*

What is the structure of the data in your program?

Stack: you can only get at the top item



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



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

### 1.1.4 Hardware

*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.

# **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	4.12		??, 39.1		
2	Conditionals	5.3		39.2		
4	Looping	6.4		39.3, 39.4, 39.6		
5	continue					
6	Functions	39.5		39.5		
7	continue				40.1	
8	I/O					
9	Structs			39.7		
10	Objects			39.8, 39.10	40.3	41.1
11	continue					
12	has-a rela- tion				40.9, 40.10, 40.1, 40.12	41.2
13	inheritance				40.13, 40.14	
14	Arrays					41.2 and further
15	continue					
16	Strings					
Advanced						
Pointers and C-style addresses		Section				
		47.1.2				
Prototypes (and separate compi- lation) and namespaces						
Error handling and exceptions		21.1				
Lambdas		23.3				

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 39.

**geom** Geometry related concepts; this is mostly an exercise in object-oriented programming. Chapter 40.

**infect** The spreading of infectious diseases; these are exercises in object-oriented design. Students can explore various real-life scenarios. Chapter 41.

**pagerank** The Google Pagerank algorithm. Students program a simulated internet, and explore pageranking, including ‘search engine optimization’. This exercise uses lots of pointers. Chapter 42.

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

### 3.2.2 Choice: Fortran or advanced topics

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

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

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

### 3. Teachers guide

## **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 file(s) is/are translated by 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.)

To compile your program, you then type

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

On TACC machines, use the Intel compiler:

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

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

So 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●something.cc : your source
file.
```

- icpc : compiler. Alternative Gnu: Run this program (it gives no output):  
use g++ instead of icpc.                                   ./zeroprogram
- -o programname : output into a

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 sometimes used, but your instructor does not like that.
- 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.

## 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 an *executable*.
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.

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

- the string `One third is`, and
  - the result of the computation  $1/3$ ,
- with the same `cout` statement?

## 4.3 Variables

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

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

Think of a variable as a labeled place 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,
- can contain 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 `cout` are reserved words.
- Usually `i` and `j` are not the best variable names.

### *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 4.6. 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 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; section 22.1.
- For characters: `char`. Strings are complicated.
- Truth values: `bool`
- You can make your own types. Later.

## 4.3.3 Assignments

### *Setting a variable*

```
i = 5;
```

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

let  $i = 5$ .

### *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 4.7.** Which of the following are legal?

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

### *Initialization*

You can also give a variable a value a in *variable initialization*. Confusingly, there are several ways of doing that. Here's two:

---

```
int n = 0;
double x = 5.3, y = 6.7;
double pi{3.14};
```

Do not use uninitialized variables! Doing so is legal, but there is no guarantee about the initial value. Do not count on it being zero...

**Exercise 4.8.** Write a program that has several variables. Assign values either in an initialization or in an assignment. Print out the values.

**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.

### *Integer constants*

Iintegers 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; // C++14
long ell = 42L;
```

### 4.3.4 Floating point variables

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

- Within a certain range, roughly  $-2 \cdot 10^9, \dots, 2 \cdot 10^9$ , all integer values can be represented.
- 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 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.

*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.)

Exercise 4.10. It is not hard to see that the definition of floating point numbers leads to gaps.

Suppose in base 10 you type digits behind the decimal point, for instance 5.14, then 5.147 is not representable, and it will either be rounded to 5.15 or truncated to 5.14.

Define

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

and see what the smallest value `float eps` is that you can add to one so that the result is not one.

Complex numbers exist, see section 22.1.

### 4.3.5 Number values and undefined values

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

- Using 4 bytes, that is 32 bits, we can represent  $2^{32}$  integers. Typically this is the range  $-2^{31} \dots 0 \dots 2^{31} - 1$ .
- 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 `-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.

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

### 4.3.6 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;
```

### 4.3.7 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 4.11. Write a program that asks for the user's first name, and prints something like

Hello, Victor!

What happens if you enter first and last name?

## 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.

**Exercise 4.12.** 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.

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

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

### 4.5.1 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: `&` `|` `^`

### 4.5.2 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 4.13.

- 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 [23.5](#).

**Exercise 4.14.** Write a program that asks for two integer numbers `n1, n2`.

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

$$n_1 \bmod n_2$$

(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 4.15.** 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 4.16.** 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 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;
```

## 4.7 Review questions

Exercise 4.17. What is the output of:

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

Exercise 4.18. 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.



# 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'?

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

## 5.2 Operators

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

*What are logical expressions?*

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

Precendence 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_switch'
```

*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;
}
```

## 5. Conditionals

---

# Chapter 6

## Looping

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

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

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

- Inspect or change every element of a database table.

The 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.

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;  
}
```

## 6. Looping

---

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*.

If you want to perform  $N$  iterations you can write

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

or

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

The former is slightly more idiomatic to C++.

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

- An initialization. This is usually a declaration and initialization of an integer *loop variable*. Using floating point values is less advisable.
- A stopping test, usually involving the loop variable. If you leave this out, you need a different mechanism for stopping the loop; see [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.

**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? 1, 2, 3?  
`for (int i=1; i<=2; i=i+2)`
- What is the last iteration executed? 2, 4, 5?

```

    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--) { ... }
```

- Here are a couple of 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.

- 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_prete
```

*For the source of this example, see section 6.4.1*

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

- The loop body can be a single statement:

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

or a block:

```
if (x<0) {
    cout << "Error" << endl;
```

## 6. Looping

---

```
    x = -x;
}
```

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 i=0; i<m; i++)
    for (int j=0; j<n; j++)
        ...
    
```

This is called *loop nest*; the  $i$ -loop is called the *outer loop* and the  $j$ -loop the *inner loop*.

Traversing an index space (whether that corresponds to an array object or not) by  $i, j$  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?

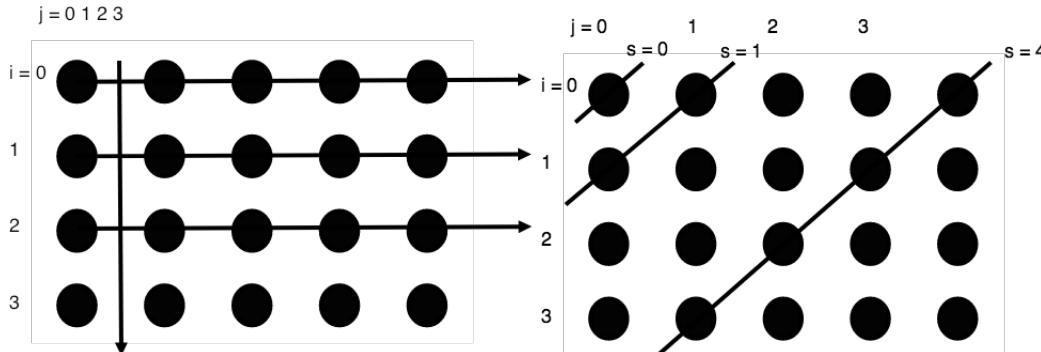


Figure 6.1: Lexicographic and diagonal ordering of an index set

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  $i + j = \text{constant}$ .

**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  $1, 2, 3, \dots$ , then find  $i, j$ .

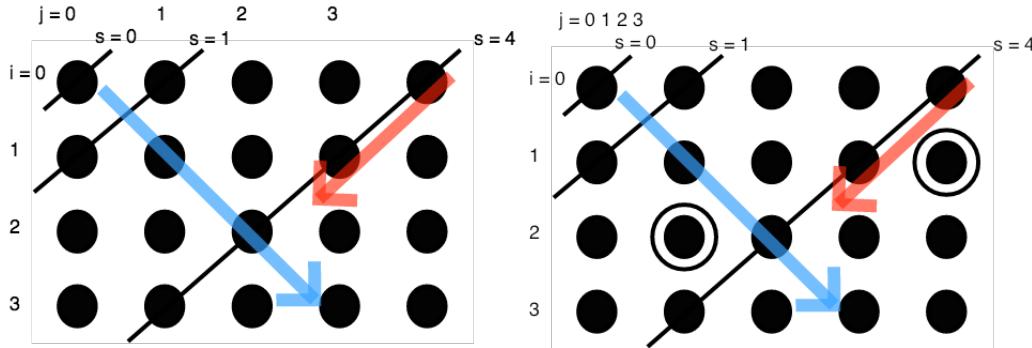


Figure 6.2: Illustration of the second part of exercise 6.4

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

## 6.2 Looping until

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

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

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

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

?

Where did the `break` happen?

A loop with a `break` is one case where you want the loop variable to be global:

```
int var;
... code that sets var ...
for ( ; var<upper; var++) {
```

## 6. Looping

---

```
... statements ...
if (some condition) break
... more statements ...
}
... code that uses the breaking value of var ...
```

### *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:

```
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.** 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 Exercises

**Exercise 6.8.** 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.

## 6. Looping

---

Exercise 6.9. 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 → 16 → 8 → 4 → 2 → 1

7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 · · ·

(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 6.10. 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 6.11. **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>

## 6.4 Sources used in this chapter

### 6.4.1 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.

#### Code reuse

Example: multiple norm calculations:

Repeated code:

```
float s = 0;
for (int i=0; i<nx; i++)
    s += abs(x[i]);
cout << "Inf norm x: " << s << endl;
s = 0;
for (int i=0; i<ny; i++)
    s += abs(y[i]);
cout << "Inf norm y: " << s << endl;
```

becomes:

```
int InfNorm( vector<float> a ) {
    float s = 0;
    for (int i=0; i<a.size(); i++)
        s += abs(a[i]);
    return s;
}
int main() {
    ... // stuff
    cout << "Inf norm x: "
        << InfNorm(x,nx) << endl;
    cout << "Inf norm y: "
        << InfNorm(y,ny) << endl;
```

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

Using a function can shorten your code, since it replaces two similar code blocks with one function definition and two calls. In this example, the code with loops is in fact longer, but 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;
- *function calls* can happen multiple times, inside the main or inside other functions.

*Function definition and call*

```

for (int i=0; i<N; i++) {
    cout << i;
    if (i%2==0)
        cout << " is even";
    else
        cout << " is odd";
    cout << endl;
}

void report_evenness(int n) {
    cout << n;
    if (n%2==0)
        cout << " is even";
    else
        cout << " is odd";
    cout << endl;
}

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

```

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, the value of the input argument.

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);

```

Functions are defined before the main program, and used in that program: Here is a program with a function that doubles its input:

#### *Program with function*

##### **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_twicein'
```

For the source of this example, see section [7.9.1](#)

## 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*

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

can be replaced by

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

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

A final argument for using functions is code maintainability:

- Easier to debug: if you use the same (or roughly the same) block of code twice, and you find an error, you need to fix it twice.

- Maintainance: if a block occurs twice, and you change something in such a block once, you have to remember to change the other occurrence(s) too.
- Localization: any variables that only serve the calculation in the function now have a limited scope.

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

**Review 7.1.** True or false?

- The purpose of functions is to make your code shorter.
- Using functions makes your code easier to read.
- 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`.)

#### 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;
}

```

*Functions 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;
}

```

*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.

Review 7.2. True or false?

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

## 7.4 Parameter passing

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

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

### 7.4.1 Pass by value

The following style of programming is very much inspired by mathematical functions, and is known as *functional programming*<sup>1</sup>.

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

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

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

**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_passva...
```

*For the source of this example, see section 7.9.2*

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:

**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_localp...
```

*For the source of this example, see section 7.9.3*

**Exercise 7.1.** If you are doing the prime project (chapter 39) you can now do exercise 39.5.

*Background Square roots through Newton*

---

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

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 to compute  $x \leftarrow \sqrt{y}$ . 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  $\sqrt{y}$ .

## 7.4.2 Pass by reference

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

We need the concept of a *reference*:

### 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_ref'.
```

*For the source of this example, see section 7.9.4*

(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 1** *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_setbyref'. Stop.
```

*For the source of this example, see section 7.9.5*

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)
        return false;
    else
        value = ...;
    return true;
}

```

```

        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 bool function that tests divisibility and returns a remainder:

```

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 40.1.

#### 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.

## 7.5 Recursive functions

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

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

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

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

and now it looks like a C++ function:

## 7. Functions

---

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

is the function header of a factorial function. 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 2** A function does not need to call itself directly to be recursive; if it does so indirectly we can call this mutual recursion.

**Remark 3** 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 47.3.1.

### 7.5.1 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 correct<sup>2</sup>, but it also means that if your function has both

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

it may lead to *stack overflow*.

---

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

## 7.6 More function topics

### 7.6.1 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.

### 7.6.2 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.

## 7.7 Library functions

### 7.7.1 Random function

There is an easy (but not terribly great) *random number generator* that works the same as in C:

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

The function `rand` yields an integer – a different one every time you call it – in the range from zero to `RAND_MAX`. Using scaling and casting you can then produce a fraction between zero and one with the above code.

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

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

seeds the random number generator from the current time.

## 7.8 Review questions

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

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

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

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

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.

## 7.9 Sources used in this chapter

### 7.9.1 Listing of code/func/twicein

```
*****  
**** This file belongs with the course  
**** Introduction to Scientific Programming in C++/Fortran2003  
**** copyright 2017/8 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;  
}
```

### 7.9.2 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;
}
```

### 7.9.3 Listing of code/func/localparm

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 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;
}
```

#### 7.9.4 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;
}
```

#### 7.9.5 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:**

**Output**

[basic] shadowtrue:

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

For the source of this example, see section [8.5.1](#)

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?

**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 23.4) 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; };
};

/* ... */
cout << "Before the nested scope" << endl;
{
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
}
cout << "After the nested scope" << endl;
```

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

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

For the source of this example, see section [10.5.10](#)

## 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/basic/shadowtrue

### 8.5.2 Listing of code/object/destructor

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 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 destructor  
    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;

## 9. Structures

---

- 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; } ;
// needs compiler option: -std=c++11
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_point'
```

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++.

- Same,

```
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:**

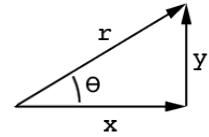
```
double distance
{
    struct vector v1,
        struct vector v2 )
{
    double
        d1 = v1.x-v2.x, d2 = v1.y-v2.y;
    return sqrt( d1*d1 + d2*d2 );
}
/* ... */
struct vector v1 = { 1.,1. };
cout << "Displacement x,y?";
double dx,dy; cin >> dx >> dy; cout << endl;
cout << "dx=" << dx << ", dy="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

For the source of this example, see section [9.3.2](#)



**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:**

**Output**

[struct] pointangle:

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

For the source of this example, see section [9.3.3](#)

**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:**

**Output**

[struct] pointflip:

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

For the source of this example, see section [9.3.4](#)

**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 39) you can now do exercise 39.7.

**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_point...
```

*For the source of this example, see section 9.3.5*

**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_point...
```

*For the source of this example, see section 9.3.6*

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

**Exercise 9.8.** Write a  $2 \times 2$  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 2016-8 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/pointangle

### 9.3.4 Listing of code/struct/pointflip

### 9.3.5 Listing of code/struct/pointdenote

### 9.3.6 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. Since these functions are really tied to the definition of the `struct`, shouldn't there be a way to make that tie explicitly?

That's what an object is:

- An object is like a structure in that it has data members.
- An object has *methods* which are the functions that operate on that object.

C++ does not actually have a ‘object’ keyword; instead you define a class with the `class` keyword, which describes all the objects of that class.

First of all, you can make an object look pretty much like a structure:

**Code:**

```
class Vector {  
public:  
    double x,y;  
};  
  
int main() {  
    Vector p1;  
    p1.x = 1.; p1.y = 2.; // This Is Not A Good Idea. See later.  
    cout << "sum of components: " << p1.x+p1.y << endl;
```

**Output**

[geom] pointstruct:

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

For the source of this example, see section 10.5.1

Observations about the above code snippet:

- Again we have separate definition of the class and declaration of the objects. You define the class only once, after which you can make as many objects of that class as you want.
- There are data members. We will get to the `public` in a minute.
- You make an object of that class by using the class name as the datatype.
- The data members can be accessed with the period.

### 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*. Above you had a declaration `Vector v1` which was not just a declaration: it called the so-called *default constructor*, which has no arguments, and does nothing.

Usually you write your own constructor, for instance to initialize data members:

```
class Vector {  
private:  
    double vx, vy;  
public:  
    // constructor  
    Vector( double userx, double usery ) {  
        vx = userx; vy = usery;  
    }  
  
    Vector p1(1.,2.), p2(3.7,-21./5);
```

### 10.1.2 Public and private

In the first example above, the data members of the `Vector` class were declared `public`, meaning that they are accessible from the calling (main) program. While this is initially convenient for coding, it is a bad idea in the long term. For a variety of reasons it is good practice to separate interface and implementation of a class.

*Example of accessor functions*

Getting and setting of members values is done through accessor functions:

```
class Vector {  
private: // recommended!  
    double vx, vy;  
public:  
    Vector( double x, double y ) {  
        vx = x; vy = y;  
    };  
    double y() { return vy; };  
    void setx( double newx ) {  
        vx = newx; };  
    void sety( double newy ) {  
        vy = newy; };  
}; // end of class definition  
  
public:  
    double x() { return vx; };  
  
    Vector p1(1.,2.);
```

Usage:

```
p1.setx(3.12);  
/* ILLEGAL: p1.x() = 5; */  
cout << "P1's x=" << p1.x() << endl;
```

### *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.

### **10.1.3 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) {
    }
```

### *advantages*

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_point...
```

*For the source of this example, see section 10.5.2*

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

### *Initializer lists*

*Initializer lists* can be used as denotations.

#### 10.1.4 Methods

With the accessors, you have just seen a first example of a class *method*: a function that is only defined for objects of that class, and that have 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_point...
```

*For the source of this example, see section 10.5.3*

**Exercise 10.1.** Add a `print` function to the `Vector` class.

How many parameters does it 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_points'
```

For the source of this example, see section [10.5.4](#)

### 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
}

```

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_pointscale'.

```

For the source of this example, see section [10.5.5](#)

### 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 p;
```

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 ) {
    vx = x; vy = y;
}

```

### 10.1.6 Accessors

It is a good idea to keep data members private, and use accessor functions.

*Use accessor functions!*

```
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() { /* ... */ };
};
```

Functionality is independent of implementation.

### 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\_stream'.

*For the source of this example, see section 10.5.6*

Exercise 10.2. If you are doing the prime project (chapter 39), now is a good time to do exercise 39.8.

### 10.1.8 Advanced topics

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

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\_access'

*For the source of this example, see section 10.5.7*

The function `the_x` 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 10.3. This is a good time to do the exercises in section 40.2.

### 10.1.8.1 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).

You can have multiple methods with the same name, as long as they can be distinguished by their argument types. This is known as *overloading*.

### 10.1.8.2 Operator overloading

Instead of writing

```
myobject.plus(anotherobject)
```

you can actually redefine the + operator so that

```
myobject + anotherobject
```

is legal.

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.

See section 14.4 for redefining the parentheses and square brackets.

**Exercise 10.4.** Write a Fraction class, and define the arithmetic operators on it.

### 10.1.8.3 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_copyscalar'
```

For the source of this example, see section [10.5.8](#)

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_copyve
```

For the source of this example, see section [10.5.9](#)

#### 10.1.8.4 Destructor

Just 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; };
};

/* ... */
cout << "Before the nested scope" << endl;
{
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
}
cout << "After the nested scope" << endl;
```

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

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

For the source of this example, see section [10.5.10](#)

**Exercise 10.5.** Write a class

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

used as

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

which gives output

```
**** creating object with 5 ****
**** setting object to 6 ****
**** setting object to -2 ****
**** object destroyed after 2 updates ****
```

### *Destructors and exceptions*

The destructor is called when you throw an exception:

**Code:**

**Output**

[object] exceptobj:

make[5]: \*\*\* No rule to make target 'run\_except'

For the source of this example, see section [10.5.11](#)

## 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 { /* ... */ }
class Course {
private:
    Person the_instructor;
    int year;
}
class Person {
    string name;
    ...
}
```

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 10.6.** If you are doing the geometry project, this is a good time to do the exercises in section 40.3.

## 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 10.7.** If you are doing the geometry project, you can now do the exercises in section [40.4](#).

### 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) = 0;
    virtual float operator[](int) = 0;
};

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.

#### 10.3.3.1 ‘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);
};
```

### 10.3.3.2 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;
```

## 10.4 Review question

Exercise 10.8. 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 10.9. Describe various ways to initialize the members of an object.

## 10.5 Sources used in this chapter

### 10.5.1 Listing of code/geom/pointstruct

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

```
**** pointstruct.cxx : make a Point class look just like a struct
*****
***** ****
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//codesnippet pointstruct
class Vector {
public:
    double x,y;
};

int main() {
    Vector p1;
    p1.x = 1.; p1.y = 2.; // This Is Not A Good Idea. See later.
    cout << "sum of components: " << p1.x+p1.y << endl;
//codesnippet end

    return 0;
}
```

### 10.5.2 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;
}
```

### 10.5.3 Listing of code/geom/pointfunc

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

### 10.5.4 Listing of code/geom/pointscaleby

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

```
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** pointscaleby.cxx : 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;
}
```

### 10.5.5 Listing of code/geom/pointscale

```
*****
**** This file belongs with the course
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
**** pointscale.cxx : Vector class with private data
*****
***** ****
```

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

#include <cmath>
using std::sqrt;

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

int main() {
    Vector p1(1.,2.);
//codesnippet pointscale
    cout << "p1 has length " << p1.length() << endl;
    Vector p2 = p1.scale(2.);
    cout << "p2 has length " << p2.length() << endl;
//codesnippet end

    return 0;
}
```

### 10.5.6 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.cxx : simulate an integer stream
*****
*****
```

---

```
#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;
}
```

### 10.5.7 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;
}
```

```
}
```

### 10.5.8 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;  
}
```

### 10.5.9 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;
}
```

### 10.5.10 Listing of code/object/destructor

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016-8 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 destructor
    cout << "Before the nested scope" << endl;
    {
        SomeObject obj;
        cout << "Inside the nested scope" << endl;
    }
    cout << "After the nested scope" << endl;
//examplesnippet end

    return 0;
}
```

### 10.5.11 Listing of code/object/exceptobj



# 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 11.7.

#### 11.1.1 Initialization

To define an array you need to declare its size, and you need to give it its contents. Those actions don't necessarily have to occur together. (And the contents can later change, as with any other variable; maybe even the size can change.) However, if you know the array size and contents already before you run your code, you can create the whole array in one statement. There is more than one syntax for doing so.

**Code:**

```
{  
    vector<int> numbers{5, 6, 7, 8, 9, 10};  
    cout << numbers[3] << endl;  
}  
  
{  
    vector<int> numbers = {5, 6, 7, 8, 9, 10};  
    numbers[3] = 21;  
    cout << numbers[3] << endl;  
}
```

**Output**

**[array] dynamicinit:**

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

For the source of this example, see section 11.10.1

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

- An array element `a[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.

### 11.1.2 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_dynamicmax'
```

*For the source of this example, see section 11.10.2*

(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.)

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[tmp_idx];
for (int i=0; i<5; i++) {
    int v = numbers[i];
    if (v>tmp_max) {
        tmp_max = v; tmp_idx = i;
    }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << endl;
```

**Output**

[array] idxmax:

make[5]: \*\*\* No rule to make target 'run\_idxmax'

*For the source of this example, see section [11.10.3](#)*

### *Indexing with pre/post increment*

Array indexing in `while` loop and such:

```
x = a[i++]; /* is */ x = a[i]; i++;
y = b[++i]; /* is */ i++; y = b[i];
```

**Exercise 11.1.** 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 11.2.** Find the location of the first negative element in an array.

Which mechanism do you use?

**Exercise 11.3.** 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:

**Code:**

```
vector<int> numbers = {1, 4, 2, 6, 5};
for ( auto &v : numbers )
    v *= 3;
cout << "Scale 0'th by 3: " << numbers[0] << endl;
```

**Output**

[array] dynamicscale:

make[5]: \*\*\* No rule to make target 'run\_dynamicscale'

*For the source of this example, see section [11.10.4](#)*

**Exercise 11.4.** If you do the prime numbers project, you can now do exercise [39.8.1](#).

### **11.1.3 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 [20](#) 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.

What we were doing above was creating an object and initializing it in one go. Let’s decouple these actions.

#### Vector definition

Definition, mostly without initialization.

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

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

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.

**Remark 4** There is also an `array` class, which at first glance looks like a non-resizable variant of `vector`. However, it is limited to arrays where the size is known at compile time.

##### 11.1.3.1 Vector initialization

###### Vector initialization

You can initialize a vector as a whole:

```
vector<int> odd_array{1, 3, 5, 7, 9};
vector<int> even_array = {0, 2, 4, 6, 8};
```

(This syntax requires compilation with the `-std=c++11` option.)

###### Vector 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.

##### 11.1.3.2 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.

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.

#### *Accessing vector elements*

You have already seen the square bracket notation:

```
vector<double> x(5, 0.1);
x[1] = 3.14;
cout << x[2];
```

Alternatively:

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

Safer, slower.

#### *Ranging over a vector*

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

Note that `e` is a copy of the vector element:

##### **Code:**

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

##### **Output**

**[array] vectorrangepcopy:**

make[5]: \*\*\* No rule to make target 'run\_vector'

*For the source of this example, see section 11.10.5*

#### *Ranging over a vector by reference*

To set array elements, make `e` 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[2] << endl;
```

**Output**

[array] vectorrangeref:

make[5]: \*\*\* No rule to make target 'run\_vector'

For the source of this example, see section [11.10.6](#)

Exercise 11.5. 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.

*Vector copy*

Vectors can be copied just like other datatypes:

**Code:**

```
vector<float> v(5, 0), vcopy;
v[2] = 3.5;
vcopy = v;
cout << vcopy[2] << endl;
```

**Output**

[array] vectorcopy:

make[5]: \*\*\* No rule to make target 'run\_vector'

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

#### 11.1.4 Vector methods

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?

## 11.2 Vectors are dynamic

There is an important difference between vectors and arrays: a vector can be grown or shrunk after its creation. 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_vector'
```

For the source of this example, see section [11.10.14](#)

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

Flexibility comes with a price.

This is not a good way of creating arrays. 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( ... );
```

Other methods that change the size: `insert`, `erase`.

## 11.3 Vectors and functions

### 11.3.1 Pass vector to function

*Vector as function argument*

You can pass a vector to a function:

```
void print0( vector<double> v ) {
    cout << v[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[0] = x;
}
/* ... */
vector<float> v(1);
v[0] = 3.5;
set0(v, 4.6);
cout << v[0] << endl;
```

**Output**

[array] vectorpassnot:

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

For the source of this example, see section [11.10.9](#)

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

### 11.3.2 Vector as function return

#### *Vector as function return*

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

**Code:**

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

**Output**

[array] **vectorreturn:**

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

*For the source of this example, see section 11.10.10*

**Exercise 11.8.** 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
```

#### *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[0] = x;
}
/* ...
vector<float> v(1);
v[0] = 3.5;
set0(v, 4.6);
cout << v[0] << endl;
```

**Output**

[array] **vectorpassref:**

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

*For the source of this example, see section 11.10.11*

**Exercise 11.9.** Write functions `random_vector` and `sort` to make the following main program work:

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

(This creates a vector of random values of a specified length, and then sorts it.)  
See section ?? for the random function.

**Exercise 11.10.** Revisit exercise 11.8.

Can you write a function that accepts a vector and returns two vectors with the above functionality?

## 11.4 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 ) {  
        thearray( ??? 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 write that over 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);  
};
```

You could read this as

- `vector<int> the_array` declares a int-vector variable, and
- `the_array = vector<int>(n)` assigns an array to it.
- Second form copy, first form only move.

However, technically, it actually does the following:

- The class object initially has a zero-size vector;

- the expression `vector<int> (n)` creates an anonymous vector of size n;
- which is then assigned to the variable `the_array`,
- so now you have an object with a vector of size n internally.

### 11.4.1 Dynamic size of vector

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.

### 11.4.2 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.

## 11.5 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 printable {
private:
    vector<int> values;
public:
    printable(int n) {
        values = vector<int>(n);
    };
    string stringed() {
        string p("");
        for (int i=0; i<values.size(); i++)
            p += to_string(values[i]) + " ";
        return p;
    };
}
```

Unfortunately this means you may have to recreate some methods:

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

## 11.6 Multi-dimensional cases

### 11.6.1 Matrix as vector of vectors

*Multi-dimensional vectors*

Multi-dimensional is harder with vectors:

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

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:
    int rows, cols;
    vector<vector<double>> elements;
public:
    matrix(int m, int n) {
        rows = m; cols = 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);
    };
};
```

*Matrix class'*

Better idea:

```
elements = vector<double>(rows*cols);
...
void get(int i, int j) {
    return elements.at(i*cols+j);
```

```
}
```

(Even more efficient: use cpp macro)

**Exercise 11.11.** Add methods such as transpose, scale to your matrix class.

Implement matrix-matrix multiplication.

### 11.6.2 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) {  
        this->m = m; this->n = n;  
        the_matrix.reserve(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 ];  
    };  
    /* ... */  
};
```

The most important advantage of this is that it is compatible with how many libraries and codes store a matrix traditionally.

The syntax for `set` and `get` can be improved.

**Exercise 11.12.** Write a method `element` of type `double&`, so that you can write

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

## 11.7 Static arrays

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_staticinit'
```

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_rangemax'
```

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

## 11.8 Advanced topics

### 11.8.1 Iterators

You have seen how you can iterate over a vector

- by a for 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 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.
- 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_vector'
```

For the source of this example, see section 11.10.14

**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_vector'
```

For the source of this example, see section 11.10.15

## 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.

### 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 11.13. Rewrite the above exercises where the sorting tester or the maximum finder is in a subprogram.

Unlike with scalar arguments, array arguments can be altered by a subprogram: it is as if the array is always passed by reference. This is not strictly true: what happens is that the address of the first element of the array is passed. Thus we are really dealing with pass by value, but it is the array address that is passed rather than its value.

### 11.8.2.2 Multi-dimensional arrays

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

```

float matrix[15][25];

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

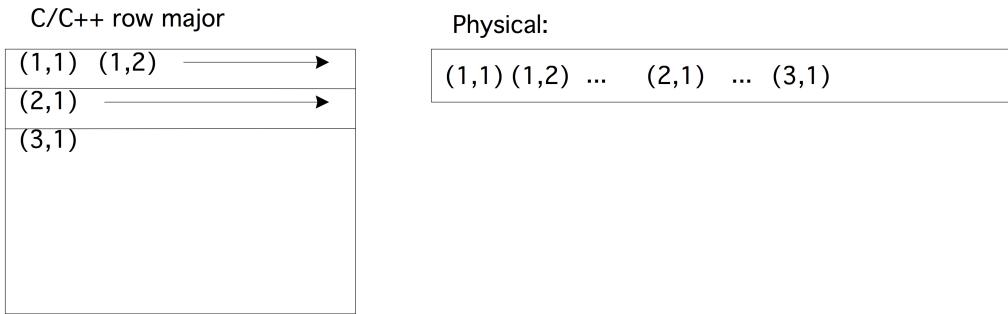
```

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

```

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

```



#### 11.8.2.3 Memory layout

Puzzling aspects of arrays, such as which dimensions need to be specified and which not in a function call, can be understood by considering how arrays are stored in memory. The question then is how a two-dimensional (or higher dimensional) array is mapped to memory, which is linear.

- A one-dimensional array is stored in contiguous memory.
- A two-dimensional array is also stored contiguously, with first the first row, then the second, et cetera.
- Higher dimensional arrays continue this notion, with contiguous blocks of the highest so many dimensions.

As a result of this, indexing beyond the end of a row, brings you to the start of the next row:

```

void print06( int ar[][6] ) {
    cout << "Array[0][6]: " << ar[0][6] << endl;
    return;
}
int array[5][6];
array[1][0] = 35;
print06(array);

```

We can now also understand how arrays are passed to functions:

- The only information passed to a function is the address of the first element of the array;
- In order to be able to find location of the second row (and third, et cetera), the subprogram needs to know the length of each row.
- In the higher dimensional case, the subprogram needs to know the size of all dimensions except for the first one.

## 11.9 Exercises

**Exercise 11.14.** 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 11.15.** Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

*Pascal's triangle*

*Pascal's triangle* contains binomial coefficients:

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

where

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

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 11.16.**

- Write a class `pascal` so that `pascal(n)` is the object containing  $n$  rows of the above coefficients. Write a method `get(i, j)` that returns the  $(i, j)$  coefficient.
- Write a method `print` that prints the above display.
- Write a method `print(int m)` that prints a star if the coefficient modulo  $m$  is nonzero, and a space otherwise.

```
*
* *
*   *
```

```
* * * *
*
* *      *
*   *   *
*   *   *   *
*   *   *   *   *
*   *
*   *
```

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

Exercise 11.17. Extend the Pascal exercise:

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

Exercise 11.18. 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 11.19. Put eight queens on a chessboard so that none threatens any other.

Exercise 11.20. 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 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[3] << endl;
    }
    {
        vector<int> numbers = {5,6,7,8,9,10};
        numbers[3] = 21;
        cout << numbers[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 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*** 
**** dynamicmax.cxx : static array length examples
*** 
***** 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/idxmax

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*** 
**** idxmax.cxx : static array length examples
*** 
***** idxmax.cxx : static array length examples
***** /
```

```
#include <iostream>
```

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

#include <vector>
using std::vector;

int main() {

    int numbers[] = {1,4,2,6,5};
    //examplesnippet idxmax
    int tmp_idx = 0;
    int tmp_max = numbers[tmp_idx];
    for (int i=0; i<5; i++) {
        int v = numbers[i];
        if (v>tmp_max) {
            tmp_max = v; tmp_idx = i;
        }
    }
    cout << "Max: " << tmp_max
        << " at index: " << tmp_idx << endl;
    //examplesnippet end

    return 0;
}
```

#### 11.10.4 Listing of code/array/dynamicscale

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
***** dynamicscale.cxx : static array length examples
*****
*****
```

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

#include <vector>
using std::vector;

int main() {

    //examplesnippet dynamicscale
    vector<int> numbers = {1,4,2,6,5};
    for ( auto &v : numbers )
        v *= 3;
    cout << "Scale 0'th by 3: " << numbers[0] << endl;
    //examplesnippet end
```

```
    return 0;
}
```

### 11.10.5 Listing of code/array/vectorrangecopy

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

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

#include <vector>
using std::vector;

int main() {

    //codesnippet vectorrangecopy
    vector<float> myvector
        = {1.1, 2.2, 3.3};
    for ( auto e : myvector )
        e *= 2;
    cout << myvector[2] << endl;
    //codesnippet end

    return 0;
}
```

### 11.10.6 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[2] << endl;
    //codesnippet end

    return 0;
}
```

### 11.10.7 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[2] = 3.5;
    vcopy = v;
    cout << vcopy[2] << endl;
    //codesnippet end

    return 0;
}
```

### 11.10.8 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;
}
```

### 11.10.9 Listing of code/array/vector passnot

```
*****  
***** This file belongs with the course  
***** Introduction to Scientific Programming in C++/Fortran2003  
***** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu  
*****  
***** vectorpassnot.cxx : 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[0] = x;
}
//codesnippet end

int main() {

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

    return 0;
}
```

#### 11.10.10 Listing of code/array/vectorreturn

```
*****
***** This file belongs with the course
***** Introduction to Scientific Programming in C++/Fortran2003
***** copyright 2016/7 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[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[0] << endl;
//codesnippet end

    return 0;
}
```

### 11.10.11 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
*****
**** vectorpassref.cxx : 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[0] = x;
}
//codesnippet end

int main() {

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

    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;
}
```

#### 11.10.14 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
*****
**** vectorendcxx : 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;  
}
```

**11.10.15 Listing of code/array/vectorenditerator**

## 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_intcha
```

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'
```

## 12. Strings

---

(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 (use `-std=c++11`), 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_quote'
```

*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_string'
```

For the source of this example, see section [12.5.3](#)

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_string'
```

For the source of this example, see section [12.5.4](#)

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_string'
```

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 cipher*. You would take an integer  $s$  and rotate every character of the text over that many positions:

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

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

Exercise 12.3. (this continues exercise [12.2](#))

If you find a message encrypted with the Caesar cipher, can you decrypt it? Take

your inspiration from the *Sherlock Holmes* story ‘The Adventure of the Dancing Men’, where he uses the fact that ‘e’ is the most common letter.

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

### 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](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 `\0`. 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 Victor Eijkhout eijkhout@tacc.utexas.edu
*****
```

```
**** intchar.cxx : 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/stringindex

### 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/stringrangeset



# 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_cunfo
```

For the source of this example, see section [13.7.1](#)

#### 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_width'
```

For the source of this example, see section [13.7.2](#)

#### 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_formatpad'
```

For the source of this example, see section [13.7.3](#)

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_formatleft'
```

For the source of this example, see section [13.7.4](#)

### 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_format16'
```

For the source of this example, see section [13.7.5](#)

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):

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

Back to decimal:

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

## 13.2 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_formatfloat'.  Stop.
```

*For the source of this example, see section 13.7.6*

(Notice the rounding)

### *Fixed point precision*

Fixed precision applies to fractional part:

## 13. Input/output

---

### 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_fix'.
```

For the source of this example, see section [13.7.7](#)

### 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_align'
```

For the source of this example, see section [13.7.8](#)

### 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
```

### 13.3 Saving and restoring settings

```
ios::fmtflags old_settings = cout.flags();  
  
cout.flags(old_settings);  
  
int old_precision = cout.precision();  
  
cout.precision(old_precision);
```

## 13.4 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);
```

#### 13.4.1 Output your own classes

You have used statements like:

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

## 13. Input/output

---

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;
```

## 13.5 Binary output

### Code:

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

For the source of this example, see section [13.7.9](#)

### Code:

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

For the source of this example, see section [13.7.10](#)

### Output

#### [io] fio:

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

### Output

#### [io] fiobin:

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

## 13.6 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/>.

### 13.6.1 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.
- It is not a good idea to assume an EOF character, or 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. If it actually returns an `ifstream`. However, a conversion operator

```
explicit operator bool() const;
```

exists for anything that inherits from `basic_ios`.

### 13.6.2 Input streams

Test, mostly for file streams: `is_eof` `is_open`

## 13.7 Sources used in this chapter

### 13.7.1 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;
}
```

### 13.7.2 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
*****
***** 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;
}
```

### 13.7.3 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;
}
```

### 13.7.4 Listing of code/io/formatleft

```
/*********************************************
*****
**** 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;
}
```

### 13.7.5 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;
}
```

### 13.7.6 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;
```

## 13. Input/output

---

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

return 0;
}
```

### 13.7.7 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;
}
```

### 13.7.8 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;
}
```

### 13.7.9 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;
}
```

### 13.7.10 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 7.4.2. Make sure you study that material first.

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

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

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

```
void change_scalar_by_reference(int &i) { i += 1; }
```

There is no change to the calling program. (Some people who are used to C find this bad, since you can not see from the use of a function whether it passes *by reference* or *by value*.)

#### 14.2 Pass by reference

If you use a mathematical style of subprograms, where some values go in, and a new entity comes out, in effect all the inputs can be copied. This style is called *functional programming*, and there is much to be said for it. For instance, it makes it possible for the compiler to reason about your program. The only thing you have to worry about is the cost of copying, if the inputs are of non-trivial size, such as arrays.

However, sometimes you want to alter the inputs, so instead of a copy you need a way of accessing the actual input object. That's what *references* are invented for: to allow a subprogram access to the actual input entity.

A bonus of using references is that you do not incur the cost of copying. So what if you want this efficiency, but your program is really functional in design? Then you can use a *const reference*: the

## 14. References

---

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*

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	<p>Instead write:</p> <pre>void f( BigDude &amp;thing ) { .... };</pre> <p>Prevent changes:</p> <pre>void f( const BigDude &amp;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_constref'.  
Stop.
```

For the source of this example, see section [23.8.1](#)

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_rhsref'
```

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

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 [23.1.1](#) 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 [10.1.8.2](#). 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;
```

Another reason for wanting to return a reference is 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 [23.6.3](#).

## 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 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;
//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. If you don't already speak C, skip to the first subsection.

If you do already know C, we remark that there is less need for pointers in C++ than there was in C.

- To pass an argument *by reference*, use a *reference*. Section 7.4.
- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see above.
- Traversing arrays and vectors can be done with ranges; section 11.1.2.
- 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 47.1.2.
- 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 */ );  
    // or:  
auto X = shared_ptr<Obj>( new Obj( /* args */ ) );  
  
X->method_or_member;
```

This 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_pointx'
```

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

### 15.2.1 Pointers and arrays

*Linked lists*

The prototypical example use of pointers is in linked lists. Let a class `Node` be given:

### *List usage*

Example use:

Code:

**Output**  
[tree] simple:

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

### *Linked lists and recursion*

Many operations on linked lists can be done recursively:

Exercise 15.1. Write a recursive `append` method that appends a node to the end of a list:

Code:

**Output**  
[tree] append:

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

Exercise 15.2. Write a recursive `insert` method that inserts a node in a list, such that the list stays sorted:

Code:

**Output**  
[tree] insert:

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

Exercise 15.3. For a more sophisticated approach to linked lists, do the exercises in section [47.1.2](#).

Exercise 15.4. If you are doing the prime numbers project (chapter [39](#)) you can now do exercise [39.8.2](#).

## 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
0x7ffeb9caf08: 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\_ptr1'.

*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_ptr2'.
```

*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_pointy'.
```

*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 [47.1.2](#).

## 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 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>
```

### 15.5.3 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.

### 15.5.4 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 ??). 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.

### 15.5.5 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.

#### Code:

```
auto array = make_shared<double>(50);
shared_ptr<double> other;
array.get()[2] = 3.;
// the following two are ILLEGAL:
// array->at(2) = 4.;
// array.get().at(2) = 4.;
other = array;
cout << other.get()[2] << endl;
```

#### Output

[pointer] pptrarray:

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

For the source of this example, see section 15.6.5

## 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
**** Introduction to Scientific Programming in C++/Fortran2003
**** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
**** ptr2.cxx : shared pointers
****

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

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

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

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

int main() {

//codesnippet shareptr2
cout << "set pointer2" << endl;
auto thing_ptr2 =
    make_shared<thing>();
cout << "set pointer3 by copy"
    << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
    << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
    << endl;
thing_ptr3 = nullptr;
//codesnippet end

#if 0
// alternatively
auto thing_ptr2 =
    shared_ptr<thing>
        ( new thing );
#endif
}

```

```
    return 0;  
}
```

#### 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  
****  
*****  
  
#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/ptrarray

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

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

int main() {

    //codesnippet ptrarray
    auto array = make_shared<double>(50);
    shared_ptr<double> other;
    array.get()[2] = 3.;
    // the following two are ILLEGAL:
    // array->at(2) = 4.;
    // array.get().at(2) = 4.;
    other = array;
    cout << other.get()[2] << endl;
    //codesnippet end

    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_printfpoint'. Stop.
```

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 of i, hex     : "
<< std::hex << &i << endl;
```

### Output

#### [pointer] coutpoint:

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

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 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;
```

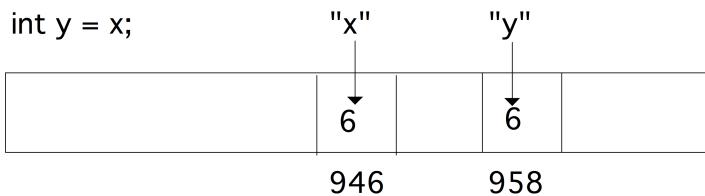
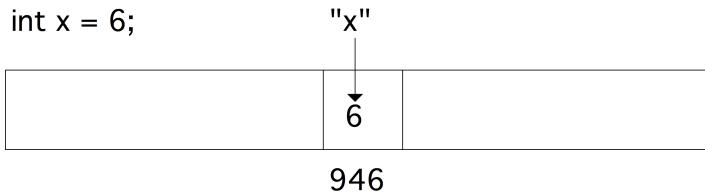
### Output

#### [pointer] cintpointer:

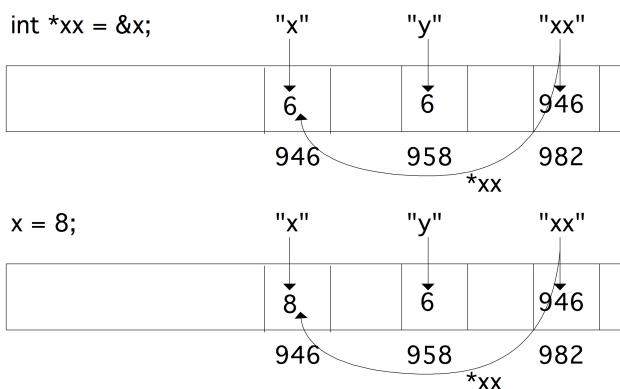
```
make[5]: *** No rule to make target 'run_cintpo
```

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 11.7 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_array...
```

*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: " << j << endl;
    return 0;
}
```

Hint: write C++ code, then insert stars where needed.

### 16.6.1 Allocation

In section 11.7 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 2017/8 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 if 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

## Prototypes

### 17.1 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*

A first use of prototypes is *forward declaration*:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

Prototypes are useful if you spread your program over multiple files. You would put your functions in one file and the main program in another. Splitting your code over multiple files is good software engineering practice for a number of reasons. For instance, it decreases compilation time: if you make a small change, only that file needs to be recompiled.

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:

#### *Prototypes for separate compilation*

```
// file: def.cxx                                // file : main.cxx
int tester(float x) {                           int tester(float);
    ....
}
```

```
int main() {
    int t = tester(...);
    return 0;
}
```

(However, you would not do things like this in practice. See the next section about header files.)

### 17.1.1 Separate compilation

Breaking your code in multiple files and using *separate compilation* has several advantages.

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.

*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.

### 17.1.2 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          // file : main.cxx
#include "def.h"          #include "def.h"
int tester(float x) {      int main() {
    ....                  int t = tester(...);
}                           return 0;
}                           }
```

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 ) { .... }
```

### 17.1.3 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>
```

## 17.2 Prototypes for class methods

*Class prototypes*

Header file:

```
class something {  
public:  
    double somedo(vector);  
};
```

Implementation file:

```
double something::somedo(vector v) {  
    .... something with v ....  
};
```

Strangely, data members also go in the header file.

Review 17.1. For each of the following answer: is this a valid function definition or function prototype. Are any of them a constructor?

- int foo();
- int foo() {};
- int foo(int) {};
- int foo(int bar) {};
- int foo(int) { return 0; };
- int foo(int bar) { return 0; };
- foo();
- foo() {};

## 17.3 Header files and templates

The use of *templates* often make separate compilation impossible: in order to compile the templated definitions the compiler needs to know with what types they will be used.

## 17.4 Namespaces and header files

Never put using namespace in a header file.

## 17.5 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 19.)

This also prevents recursive inclusion of header files.



# Chapter 18

## Namespaces

### 18.1 Solving name conflicts

In section 11.1.3 you saw that the C++ library 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 Standard Template Library (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();
```

### 18.1.1 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 );
};
}
```

## 18.2 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 19

### 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.

#### 19.1 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.

## 19.2 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 19.1.** 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) ] = ...
```

## 19.3 Conditionals

There are a couple of *preprocessor conditions*.

### 19.3.1 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
```

### 19.3.2 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 [17.5](#).

### 19.3.3 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
```



## Chapter 20

### 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.

#### 20.1 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 20.1. Machine precision, or ‘machine epsilon’, is sometimes defined as the smallest number  $\epsilon$  so that  $1 + \epsilon > 1$  in computer arithmetic.

Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

Output  
[template] eps:

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

For the source of this example, see section [20.5.1](#)

## 20.2 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
}
```

## 20.3 Specific implementation

```
template <typename T>
void f(T);
template <>
void f(char c) { /* code with c */ };
template <>
void f(double d) { /* code with d */ };
```

## 20.4 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 20.2. Write a class that contains an array. The length of the array should be templated.

## 20.5 Sources used in this chapter

### 20.5.1 Listing of code/template/eps



# Chapter 21

## Error handling

### 21.1 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 11.1.3.

**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

Assertions:

```
#include <cassert>
...
assert( bool )
```

assertions are omitted with optimization

Function return values

## 21.2 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 21.1.** 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 10.1.8.4.
- 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 { ... };
```

## 21.3 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.

### 21.3.1 Legacy C mechanisms

The `errno` variable and the `setjmp / longjmp` functions should not be used. These functions for instance do not the memory management advantages of exceptions.

## 21.4 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 22

### 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 18.1.

#### 22.1 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.

#### 22.2 Containers

Vectors (section 11.1.3) and strings (chapter 12) are special cases of a STL *container*. Methods such as `push_back` and `insert` apply to all containers.

### 22.2.1 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;
```

### 22.2.2 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](#).

## 22.3 Tuples

Remember how in section [7.4.2](#) we said that if you wanted to return more than one value, you could not do that through a return value, and had to use an *output parameter*? Well, using the STL there is a different solution.

You can make a *tuple*: an entity that comprises several components, possibly of different type, and which unlike a `struct` you do not need to define beforehand.

*C++11 style tuples*

```
|| std::tuple<int, double> id = std:
|| id = std::make_tuple<int, double>(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<bool, float>(false, -1);
    else
        return make_tuple<bool, float>(true, sqrt(x));
}
```

but more interestingly, you can use a *tuple denotation*:

```
tuple<bool, float> maybe_root2(float x) {
    if (x<0)
        return {false, -1};
    else
        return {true, sqrt(x)};
}
```

### 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_tuple'
```

For the source of this example, see section [22.6.1](#)

**22.4 Random numbers****22.5 Time****22.6 Sources used in this chapter****22.6.1 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;
}
```



# Chapter 23

## Obscure stuff

### 23.1 Const

#### 23.1.1 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.

#### 23.1.2 Const references

A more sophisticated use of `const` is the *const reference*:

```
void f( const int &i ) { .... }
```

This may look strange. After all, references, and the pass-by-reference mechanism, were introduced in section 7.4 to return changed values to the calling environment. The `const` keyword negates that possibility of changing the parameter.

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

Marking a vector argument as `const` allows *compiler optimization*. Assume the function `f` as above, used like this:

```
std::vector<double> v(n);  
for ( .... ) {  
    f(v);  
    y = v[0];  
    ... y ... // code that uses y  
}
```

Since the function call does not alter the vector, `y` is invariant in the loop iterations, and the compiler changes this code internally to

```
std::vector<double> v(n);
int saved_y = v[0];
for ( .... ) {
    f(v);
    ... saved_y ... // code that uses y
}
```

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\_constref'

For the source of this example, see section [23.8.1](#)

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_const...
```

*For the source of this example, see section [23.8.2](#)*

### 23.1.3 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.

- It allows the compiler to optimize your code. For instance:

```
class Things {
public:
    int f() const { /* ... */ };
    int g() const { /* ... */ };
```

```
}
```

...

```
Things t;
int x,y,z;
x = t.f();
y = t.g();
z = t.f();
```

Since the methods did not alter the object, the compiler can conclude that `x`, `z` are the same, and skip the calculation for `z`.

## 23.2 Auto

This is not actually obscure, but it intersects many other topics, so we put it here for now.

### 23.2.1 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:

*Type deduction in functions*

Return type can be deduced in C++17:

*Auto and references, 1*

`auto` discards references and such:

**Code:**

**Output**

**[auto] plainget:**

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

*For the source of this example, see section 23.8.3*

*Auto and references, 2*

Combine `auto` and references:

**Code:****Output****[auto] refget:**

make[5]: \*\*\* No rule to make target 'run\_refget

*For the source of this example, see section 23.8.4**Auto and references, 3*

For good measure:

**Code:****Output [auto] constrefget:**

make[5]: \*\*\* No rule to make target 'error\_constrefget

## 23.2.2 Iterating

*Auto iterators*

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
    s += copy_of_int;
for ( auto &ref_to_int : myvector )
    ref_to_int = s;

// short for:

for ( std::iterator it=myvector.begin() ;
      it!=myvector.end() ; ++it )
    s += *it ; // note the deref
```

Can be used with anything that is iterable  
(vector, map, your own classes!)

## 23.3 Iterating over classes

You know that you can iterate over `vector` objects:

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
    s += copy_of_int;
for ( auto &ref_to_int : myvector )
    ref_to_int = s;
```

(Many other STL classes are iterable like this.)

This is not magic: it is possible to iterate over any class: a *class* is *iterable* that has a number of conditions satisfied.

The class needs to have:

- a method `begin` with prototype

```
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: 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_bagfir
```

For the source of this example, see section [23.8.6](#)

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 23.1.** You can now do exercise 39.15, implementing a prime number generator with this mechanism.

If you think you understand `const`, consider that the `has` method is conceptually `cost`. But if you add that keyword, the compiler will complain about that use of `*this`, since it is altered through the `begin` method.

**Exercise 23.2.** Find a way to make `has` a `const` method.

## 23.4 Lambdas

The C++11 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.

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:

For the source of this example, see section [23.8.7](#)

**Exercise 23.3.** Refer to [7.15](#) 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, used as

**Exercise 23.4.** Can you write a derived class `rootfinder` used as

## 23.5 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](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)

### 23.5.1 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 = 100000000000;
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\_intlong'

For the source of this example, see section 23.8.8

There are no runtime tests on static casting.

Static casts are a good way of casting back void pointers to what they were originally.

### 23.5.2 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] deriveright:**

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

For the source of this example, see section [23.8.9](#)

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_derive'
```

For the source of this example, see section [23.8.10](#)

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.

### 23.5.3 Const cast

With `const_cast` you can add or remove `const` from a variable. This is the only cast that can do this.

### 23.5.4 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 23.4):

```
KSPSetMonitor( ksp,
    [mycontext] (KSP k,int ,PetscReal r) -> int {
        my_monitor_function(k,r,mycontext); } );
```

## 23.6 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.

### 23.6.1 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
```

### 23.6.2 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.

### 23.6.3 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;
}
```

## 23.7 Move semantics

With overloaded 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 = {};
    }
};
```

## 23.8 Sources used in this chapter

### 23.8.1 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;  
}
```

### 23.8.2 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;  
}
```

### 23.8.3 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;
}
```

#### 23.8.4 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;  
}
```

### 23.8.5 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;  
}
```

**23.8.6 Listing of code/loop/bagfind****23.8.7 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 lambdaclass  
#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 lambdaclass  
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 << ":" << endl
    << multiples.string() << endl;

return 0;
}
```

### 23.8.8 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;
}
```

### 23.8.9 Listing of code/cast/deriveright

#### 23.8.10 Listing of code/cast/derivewrong



## Chapter 24

### C++ for C programmers

#### 24.1 I/O

There is little employ for `printf` and `scanf`. Use `cout` (and `cerr`) and `cin` instead. There is also the `fmtlib` library.

Chapter 13.

#### 24.2 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.

#### 24.3 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.

#### 24.4 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 11.1.2.
- To pass an argument *by reference*, use a *reference*. Section 7.4.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

There are some legitimate needs for pointers, such as Objects on the heap. In that case, use `shared_ptr` or `unique_ptr`; section 15.2. The C pointers are now called *bare pointers*, and they can still be used for ‘non-owning’ occurrences of pointers.

#### 24.4.1 Parameter passing

No longer by address: now true references! Section [7.4](#).

### 24.5 Objects

Objects are structures with functions attached to them. Chapter [10](#).

### 24.6 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter [18](#).

### 24.7 Templates

If you find yourself writing the same function for a number of types, you'll love templates. Chapter [20](#).

### 24.8 Obscure stuff

#### 24.8.1 Lambda

Function expressions. Section [23.4](#).

#### 24.8.2 Const

Functions and arguments can be declared const. This helps the compiler. Section [23.1.1](#).

#### 24.8.3 Lvalue and rvalue

Section [23.6](#).

## **PART III**

### **FORTRAN**



## Chapter 25

### 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](http://www.pcc.qub.ac.uk/tec/courses/f77tof90/stu-notes/f90studentMIF_1.html)

#### 25.1 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.

## 25.2 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 25.1. 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_stop'.
```

For the source of this example, see section [25.8.1](#)

## 25.3 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.

### 25.3.1 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](#).)

### 25.3.2 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
```

### 25.3.3 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
```

## 25.4 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 25.4.1.
- *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 25.4.2.
- Logical: `Logical`.
- Character: `Character`. Strings are realized as arrays of characters; chapter 30.

#### 25.4.1 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 some 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
```

#### 25.4.2 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`.

### *Precision conversion*

```
real(8) :: x,y
x = 3.14
y = 6.022e-23
```

You can query how many bytes a data type takes with `kind`.

#### *Numerical precision*

Number of bytes determines numerical precision:

- Computations in 4-byte have relative error  $\approx 10^{-6}$
- Computations in 8-byte have relative error  $\approx 10^{-15}$

Also different exponent range: max  $10^{50}$  and  $10^{300}$  respectively.

#### *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_complex'
```

For the source of this example, see section [25.8.2](#)

### 25.4.3 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.

## 25.5 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.

## 25.6 Expressions

*Arithmetic expressions*

- Pretty much as in C++
- Exception: `r**2` for power.
- Modulus is a function: `MOD(7,3)`.

*Boolean expressions*

- Long form `.and.` `.not.` `.or.` `.lt.` `.eq.` `.ge.` `.true.` `.false.`
- Short form: `< <= == /= > >=`

*Conversion and casting*

Conversion is done through functions.

- `INT`: truncation; `NINT` rounding
- `REAL`, `FLOAT`, `SNGL`, `DBLE`
- `Cmplx`, `Conjg`, `AIMag`

<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 30.

## 25.7 Review questions

Exercise 25.2. What is the output for this fragment, assuming *i*, *j* are integers?

Exercise 25.3. What is the output for this fragment, assuming *i*, *j* are integers?

Exercise 25.4. 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 25.5. In the following code, if *value* is nonzero, what do expect about the output?

## 25.8 Sources used in this chapter

### 25.8.1 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
```

### 25.8.2 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 26

### Conditionals

#### 26.1 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
```

#### 26.2 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>&gt;</code>	<code>.gt.</code>	greater	<code>y&gt;x-1</code>
<code>&gt;=</code>	<code>.ge.</code>	greater or equal	<code>sqrt(y)&gt;=7</code>
<code>&lt;</code>	<code>.lt.</code>	less than	
<code>&lt;=</code>	<code>.le.</code>	less equal	
	<code>.and. .or.</code>	and, or	<code>x&lt;1 .and. x&gt;0</code>
	<code>.not.</code>	not	<code>.not.( x&gt;1 .and. x&lt;2 )</code>
	<code>.eqv.</code>	equiv	$(x \wedge y) \vee (\neg x \wedge \neg y)$
	<code>.neqv.</code>	not equiv	$(x \wedge \neg y) \vee (\neg x \wedge y)$

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

Exercise 26.1. Read in three grades: Algebra, Biology, Chemistry, each on a scale  $1 \cdots 10$ .

Compute the average grade, with the conditions:

- Algebra is always included.
- Biology is only included if it increases the average.
- Chemistry is only included if it is 6 or more.

### 26.3 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!

### 26.4 Boolean variables

The Fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`

Exercise 26.2. Print a boolean variable. What does the output look like in the true and false case?

## 26.5 Review questions

Exercise 26.3. What is a conceptual difference between the C++ `switch` and the Fortran `Select` statement?



## Chapter 27

### Loop constructs

#### 27.1 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.

## 27.2 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*

```
do
    call random_number(x)
    if (x>.9) exit
    print *, "Nine out of ten exes agree"
end do
```

Skip rest of 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
```

## 27.3 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 27.1. 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.

## 27.4 Review questions

Exercise 27.2. 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 28

### Scope

#### 28.1 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.

##### 28.1.1 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
```

##### 28.1.1.1 Variables in a module

Variables in a module (section 32.2) have a lifetime that is independent of the calling hierarchy of program units: they are *static variables*.

#### 28.1.1.2 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.

#### 28.1.2 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 29

## Subprograms and modules

### 29.1 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 [32.2](#).
- 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 [29.2](#).

#### 29.1.1 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 29.1.** 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_printo'
```

For the source of this example, see section [29.3.1](#)

#### *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
```

For the source of this example, see section [29.3.2](#)

**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
```

For the source of this example, see section [29.3.3](#)

Note the `result` clause. This prevents ambiguity.

**29.1.2 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*

- 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,intent(in) :: invalue
        plusone = invalue+1
    end function plusone
end program plussing
```

**Output**

**[funcf] plusone:**

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

For the source of this example, see section [29.3.4](#)

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 [32.2](#) 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 29.2.** 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
```

*For the source of this example, see section [29.3.5](#)*

*Why a ‘contains’ clause?*

```
Program ContainsScope
    implicit none
    call DoWhat()
end Program ContainsScope

subroutine DoWhat(i)
    implicit none
    integer :: i
    i = 5
end subroutine DoWhat
```

Warning only, crashes.

**Output****[funcf] readpos:**

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

```
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 ContainsScope
    implicit none
    integer :: i=5
    call DoWhat(i)
end Program ContainsScope

subroutine DoWhat(x)
    implicit none
    real :: x
    print *,x
end subroutine DoWhat
```

*For the source of this example, see section [29.3.6](#)*

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.

**Output****[funcf] nocontaintype:**

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

### 29.1.3 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)
```

#### *Parameter passing*

- Everything is passed by reference.
- Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.
- Terminology: Fortran talks about ‘dummy’ and ‘actual’ arguments. Dummy: in the definition; actual: in the calling program.

#### *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?*

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 `f` removed

`y2` is same as `y1` because `x` not changed

**Exercise 29.3.** Write a subroutine `trig` that takes a number  $\alpha$  as input and passes  $\sin \alpha$  and  $\cos \alpha$  back to the calling environment.

### 29.1.4 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.

### 29.1.5 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_keyword'
```

For the source of this example, see section [29.3.7](#)

#### Optional arguments

- Extra specifier: `Optional`
- Presence of argument can be tested with `Present`

## 29.2 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
```

**Subprogram:**

```
real*8 :: in1=1.5, in2=2.6, result      function f(x,y)
                                                implicit none
result = f(in1,in2)                      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.

### 29.2.1 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.

## 29.3 Sources used in this chapter

### 29.3.1 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
```

### 29.3.2 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
```

### 29.3.3 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

```

#### 29.3.4 Listing of code/funcf/plusone

```

!*****
!***
!*** This file belongs with the course
!*** Introduction to Scientific Programming in C++/Fortran2003
!*** copyright 2017 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,intent(in) :: invalue
        plusone = invalue+1
    end function plusone
end program plussing
!!codesnippet end

```

#### 29.3.5 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
```

### 29.3.6 Listing of code/funcf/nocontaintype

### 29.3.7 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 30

## String handling

### 30.1 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"
```

### 30.2 Characters

### 30.3 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_strlen'
```

*For the source of this example, see section 30.5.1*

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_concat'
```

For the source of this example, see section [30.5.2](#)

### 30.4 Strings versus character arrays

### 30.5 Sources used in this chapter

#### 30.5.1 Listing of code/stringf/strlen

#### 30.5.2 Listing of code/stringf(concat)

## Chapter 31

### 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 )
object2 = object1
```

*Member access*

Access structure members with %

```
Type(mytype) :: typed_object
type_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)
    implicit none
    type(point),intent(in) :: p
    length = sqrt( p%x**2 + p%y )
end function length
print *, "Length:",length(p2)
```

**Exercise 31.1.** 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 32

### 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 32.1. Take exercise 31.1 and put all type definitions and all functions in a module.

### 32.1 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.

### 32.2 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 32.2.** 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
```

### 32.2.1 Polymorphism

```
module somemodule

INTERFACE swap
MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
END INTERFACE

contains
subroutine swapreal
...
end subroutine swapreal
subroutine swapint
...
end subroutine swapint
```

### 32.2.2 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 33

## Classes and objects

### 33.1 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:

```
Module multmod
  type Scalar
    real(4) :: value
  contains
    procedure,public :: print
    procedure,public :: scaled
  end type Scalar
contains ! methods
end Module multmod
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
```

Program Multiply

#### *Method definition*

```
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
```

*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*

```
Module PointClass
    Type,public :: Point
        real(8) :: x,y
    contains
        procedure, public :: distance
    End type Point
contains
    ! ....
End Module PointClass

Program PointTest
    use PointClass
    implicit none
    type(Point) :: p1,p2
    p1 = point(1.d0,1.d0)
    p2 = point(4.d0,5.d0)
    print *, "Distance:", p1%distance(p2)
End Program PointTest
```

**Exercise 33.1.** Take the point example program and add a distance function:

```
Type(Point) :: p1,p2

! initialize
dist = p1%distance(p2)
```

**Exercise 33.2.** 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
```



## Chapter 34

### 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.

#### 34.1 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  $10 \times 20$ .

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 34.4 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.

### 34.1.1 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 /)
```

### 34.1.2 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_section'
```

For the source of this example, see section [34.9.1](#)

**Exercise 34.1.** 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
```

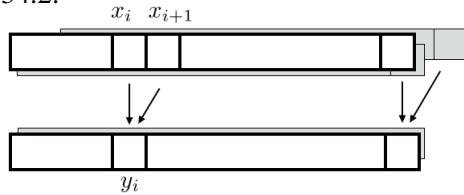
##### Output

[arrayf] sectionmg:

make[5]: \*\*\* No rule to make target 'run\_sectionmg'. Stop.

For the source of this example, see section [34.9.2](#)

**Exercise 34.2.**



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.

### 34.1.3 Integer arrays as indices

#### Index arrays

```
integer,dimension(4) :: i = [2,4,6,8]
real(4),dimension(10) :: x
print *,x(i)
```

## 34.2 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.

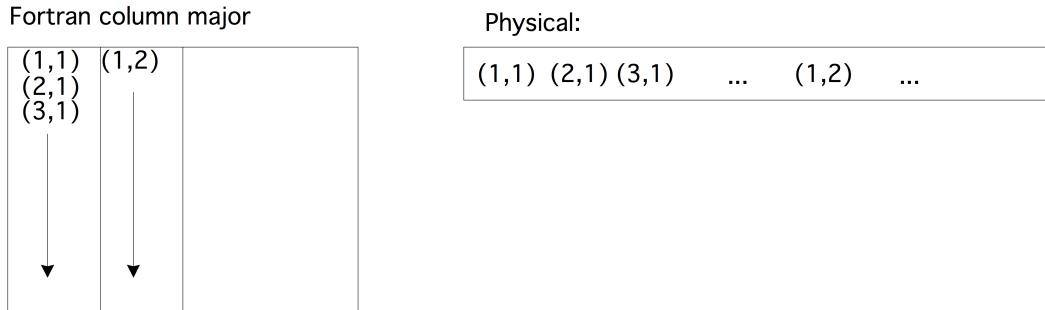
A rank-two array, or matrix, is defined like this:

#### Multi-dimension arrays

```
real(8),dimension(20,30) :: array
array(i,j) = 5./2
```

### Array layout

Sometimes you have to take into account how a higher rank array is laid out in (linear) memory:



‘First index varies quickest’

For 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:

$$\begin{matrix} 1 & 2 & \dots & N \\ N+1 & \dots & & \\ & \dots & & \\ & & MN & \end{matrix}$$

#### Code:

```
do i=1,M
  do j=1,N
    rect(i,j) = count
    count = count+1
  end do
end do
print *,rect
```

#### Output

##### [arrayf] printarray:

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

For the source of this example, see section [34.9.3](#)

### 34.2.1 Querying an array

We have the following properties of an array:

- The bounds are the lower and upper bound in each dimension. For instance, after

```
integer,dimension(-1:1,-2:2) :: symm
```

the array `symm` has a lower bound of `-1` in the first dimension and `-2` in the second. The functions `Lbound` and `Ubound` give these bounds as array or scalar:

```
array_of_lower = Lbound(symm)
upper_in_dim_2 = Ubound(symm, 2)
```

#### Code:

```
real(8),dimension(2:N+1) :: Afrom2 = &
[1,2,3,4,5]
lo = lbound(Afrom1,1)
hi = ubound(Afrom1,1)
print *,lo,hi
print '(i3,:",f5.3)', &
(i,Afrom1(i),i=lo,hi)
```

#### Output

##### [arrayf] fsection2:

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

*For the source of this example, see section 34.9.4*

- 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)
```

### 34.2.2 Reshaping

#### RESHAPE

```
array = RESHAPE( list, shape )
```

#### Example:

```
square = reshape( (/ (i, i=1, 16) /), (/4, 4/) )
```

#### SPREAD

```
array = SPREAD( old, dim, copies )
```

## 34.3 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:", arraysuum(x)
```

*Pass array: subprogram*

Note declaration as `dimension(:)`  
actual size is queried

```
real(8) function arraysuum(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
    arraysuum = tmp
end function arraysuum
```

The array inside the subroutine is known as a *assumed-shape array* or *automatic array*.

#### 34.4 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)
```

## 34.5 Array output

Use implicit do-loops; section 27.3.

## 34.6 Operating on an array

### 34.6.1 Arithmetic operations

Between arrays of the same shape:

```
A = B+C
D = D*E
```

(where the multiplication is by element).

### 34.6.2 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 34.3. 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 34.4. 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?

### 34.6.3 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
```

### 34.6.4 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 34.5. Use array statements (that is, no loops) to fill a two-dimensional array `A` with random numbers between zero and one. Then fill two arrays `Abig` and `Asmall` with the elements of `A` that are great than 0.5, or less than 0.5 respectively:

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i,j) = \begin{cases} 0 & \text{if } A(i,j) \geq 0.5 \\ A(i,j) & \text{otherwise} \end{cases}$$

Using more array statements, add `Abig` and `Asmall`, and test whether the sum is close enough to `A`.

Similar to `all`, there is a function `any` that tests if any array element satisfies the test.

```
if ( Any( Abs(A-B) >
```

## 34.7 Array operations

### 34.7.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

#### 34.7.1.1 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)
```

#### 34.7.1.2 ‘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.

#### 34.7.1.3 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`)

## 34.7.2 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 34.6. 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.

### 34.7.3 Loops with dependencies

For parallel execution of a loop, all iterations have to be independent. This is not the case if the loop has a *recurrence*, and in this case, the ‘do concurrent’ mechanism is not appropriate. Here are the above four constructs, but applied to a loop with a dependence.

```
do i=2,n
    counting(i) = 2*counting(i-1)
end do
```

Original	1	2	3	4	5	6	7	8	9	10
Recursiv	1	2	4	8	16	32	64	128	256	512

The slicing version of this:

```
counting(2:n) = 2*counting(1:n-1)
```

Original	1	2	3	4	5	6	7	8	9	10
Section	1	2	4	6	8	10	12	14	16	18

acts as if the right-hand side is saved in a temporary array, and subsequently assigned to the left-hand side.

Using ‘forall’ is equivalent to slicing:

```
forall (i=2:n)
    counting(i) = 2*counting(i-1)
end forall
```

Original	1	2	3	4	5	6	7	8	9	10
Forall	1	2	4	6	8	10	12	14	16	18

On the other hand, ‘do concurrent’ does not use temporaries, so it is more like the sequential version:

```
do concurrent (i=2:n)
    counting(i) = 2*counting(i-1)
end do
```

Original	1	2	3	4	5	6	7	8	9	10
Concurrent	1	2	4	8	16	32	64	128	256	512

Note that the result does not have to be equal to the sequential execution: the compiler is free to rearrange the iterations any way it sees fit.

## 34.8 Review questions

**Exercise 34.7.** 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?

## 34.9 Sources used in this chapter

### 34.9.1 Listing of code/arrayf/sectionassign

### 34.9.2 Listing of code/arrayf/sectionmrg

### 34.9.3 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
```

#### 34.9.4 Listing of code/arrayf/fsection2



# Chapter 35

## Pointers

Pointers in C/C++ are based on memory addresses; Fortran pointers on the other hand, are more abstract.

### 35.1 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_realp'
```

For the source of this example, see section [35.4.1](#)

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 35.1. Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

Code:

<b>Output</b>	<b>[pointerf] arpointf:</b>
	make[5]: *** No rule to make target 'run_arpoi

#### 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.

## 35.2 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.

## 35.3 Example: linked lists

For pictures of linked lists, see section [47.1.2](#).

#### 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 35.2. 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
```

## 35.4 Sources used in this chapter

### 35.4.1 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 36

### Input/output

#### 36.1 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.

#### 36.2 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.

### 36.2.1 Print on one line

The statement

```
print *,item1,item2,item3
```

will print the items on one line.

*Implicit do loops*

Parametrized printing with an *implicit do loop*:

```
print *,( i*i,i=1,n)
```

### 36.2.2 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)
```

### 36.2.3 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.

```
print '(a6,3f5.3)',"Result",x,y,z
```

The format specifier is inside single quotes and parentheses, and consists of comma-separated specifications for a single item:

- ‘*an*’ specifies a string of *n* characters. If the actual string is longer, it is truncated in the output.
- ‘*in*’ specifies an integer of up to *n* digits. If the actual number takes more digits, it is rendered with asterisks.
- ‘*fm.n*’ specifies a fixed point representation of a real number, with *m* total positions (including the decimal point) and *n* digits in the fractional part.
- ‘*em.n*’ Exponent representation.
- 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:****Output****[fio] asterisk:**

make[5]: \*\*\* No rule to make target 'run\_asterisk'. Stop.

For the source of this example, see section [36.5.1](#)

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 36.1. 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

### 36.3 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.

### 36.3.1 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.

### 36.3.2 Other write options

```
write(unit,fmt,ADVANCE="no") data
```

will not issue a newline.

open Close

## 36.4 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.

## 36.5 Sources used in this chapter

### 36.5.1 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 37

### Leftover topics

#### 37.1 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 38

### Exercises

#### 38.1 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;
```

#### 38.2 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;
}
```

### 38.3 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.

### 38.4 Subprograms

Exercise 38.1. 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 looppos
    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 looppos
```

**Output**

[funcf] looppos:

make[5]: \*\*\* No rule to make target 'run\_looppo

**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?

## 38.5 Object oriented exercises

**Exercise 38.2.** 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 38.3.** 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 38.4.** 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.

## 38.6 List access

**Exercise 38.5.** 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 39

## 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.

### 39.1 Arithmetic

*Before doing this section, make sure you study section 4.5.*

Exercise 39.1. 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.

### 39.2 Conditionals

*Before doing this section, make sure you study section 5.1.*

Exercise 39.2. 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.

### 39.3 Looping

*Before doing this section, make sure you study section 6.1.*

Exercise 39.3. 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 39.4. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

## 39.4 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 39.5. 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?

## 39.5 While loops

*Before doing this section, make sure you study section 6.2.*

Exercise 39.6. Take your prime number testing function `test_if_prime`, and use it to write 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.)

## 39.6 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 39.7. 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: 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?

## 39.7 Classes and objects

*Before doing this section, make sure you study section 10.1.*

In exercise 39.7 you made a structure that contains the data for a primesequence, and you have separate functions that operate on that structure or on its members.

**Exercise 39.8.** 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:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found() < nprimes) {
    int number = sequence.nextprime();
    cout << "Number " << number << " is prime" << endl;
}
```

In the previous exercise you defined the `primegenerator` class, and you made one object of that class:

```
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other.

**Exercise 39.9.** 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 39.10.** 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 two prime number generators, one for the  $p$ -sequence and one for the  $q$ -sequence. For each  $p$  value, when the program finds the  $q$  value, print the  $q, p, r$  triple and move on to the next  $p$ .

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  $p$ ?

## 39.8 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, ...

### 39.8.1 Arrays implementation

The sieve is easily implemented with an array that stores all integers.

**Exercise 39.11.** 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.

### 39.8.2 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 39.12.** Write a `stream` class that generates integers and use it through a pointer.

**Code:**

**Output**

**[sieve] ints:**

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

Next, we need a stream that takes another stream as input, and filters out values from it.

**Exercise 39.13.** 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:

```
Output
[sieve] odds:
make[5]: *** No rule to make target 'run_odds'
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

Exercise 39.14. 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.

## 39.9 Range implementation

*Before doing this section, make sure you study section 23.3.*

Exercise 39.15. Make a `primes` class that can be ranged:

Code:

```
Output
[primes] range:
make[5]: *** No rule to make target 'run_range'
```



## Chapter 40

### 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.

#### 40.1 Basic functions

Exercise 40.1. 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 40.2. Write a function with inputs  $x, y, \alpha$  that alters  $x$  and  $y$  corresponding to rotating the point  $(x, y)$  over an angle  $\theta$ .

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should look like:

```
double x,y,alpha;
// set x,y to some point
rotate(x,y,alpha);
// now x,y are rotated over alpha
```

#### 40.2 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 40.3. 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 40.4.** 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 40.5.** 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 40.6.** 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:**

**Output**

[geom] linearan:

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

Hint: see section 4.3.5.

**Exercise 40.7.** Revisit exercise 40.2 using the `Point` class. Your code should now look like:

```
newpoint = point.rotate(alpha);
```

**Exercise 40.8.** Advanced. Can you make a `Point` class that can accomodate any number of space dimensions? Hint: use a `vector`; section 11.1.3. Can you make a constructor where you do not specify the space dimension explicitly?

### 40.3 Using one class in another

*Before doing this section, make sure you study section 10.2.*

**Exercise 40.9.** 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 40.10.** 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 40.11.** Revisit exercises 40.2 and 40.7, 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/topleft points.

**Exercise 40.12.**

- 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.

#### 40.4 Is-a relationship

*Before doing this section, make sure you study section 10.3.*

Exercise 40.13. 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 40.14. 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.

#### 40.5 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 40.15. 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 41

## Infectuous disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

### 41.1 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.

#### 41.1.1 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.

## 41.2 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 41.1.** 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.

### 41.3 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 41.2.** 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.

### 41.4 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 41.3.** 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 41.4.** 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?

## 41.5 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](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 41.5.** 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.

## 41.6 Project writeup and submission

### 41.6.1 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 17.1.2.

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*.

### 41.6.2 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 42

### PageRank

#### 42.1 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 42.1. 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:

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:

Exercise 42.2. Add some more links to your homepage. Write a method `random_click` for the `Page` class. Intended code: How do you handle the case of a page without links?

#### 42.2 Clicking around

Exercise 42.3. 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:

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 42.4.** 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 42.5.** Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

Display the results and analyze. You may find that you finish on certain pages too many times. What's happening? Fix that.

### 42.3 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 42.6.** 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.

### 42.4 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:**

**Output**

[google] pdfsetup:

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

For the source of this example, see section [42.6.1](#)

**Exercise 42.7.** 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 42.8. 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 42.9. 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 42.10. Add a page that you will artificially made look important: add a number of pages (for instance four times the average number of links) that all link to this page, but no one links to them. (Because of the random clicking they will still sometimes be reached.)

Compute the rank of the artificially hyped page. Did you manage to trick Google into ranking this page high?

## 42.5 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 42.11. 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.

## 42.6 Sources used in this chapter

### 42.6.1 Listing of code/google/pdfsetup



## Chapter 43

### 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 34): 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.

#### 43.1 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 43.1. 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.

#### 43.2 Statistical hypothesis

We assume that Mr Shah was really saying that climate has a ‘stationary distribution’, meaning that highs and lows have a probability distribution that is independent of time. This means that in  $n$  data points,

each point has a chance of  $1/n$  to be a record high. Since over  $n + 1$  years each year has a chance of  $1/(n + 1)$ , the  $n + 1$ st year has a chance  $1/(n + 1)$  of being a record.

We conclude that, as a function of  $n$ , the chance of a record high (or low, but let's stick with highs) goes down as  $1/n$ , and that the gap between successive highs is approximately a linear function of the year<sup>1</sup>.

This is something we can test.

**Exercise 43.2.** 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 43.3.** 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 43.4.** 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 43.5.** Can you turn this exercise into a test of global warming? Can you interpret the deviations as the sum of a yearly increase in temperature plus a stationary distribution, rather than a stationary distribution by itself?

---

1. Technically, we are dealing with a uniform distribution of temperatures, which makes the maxima and minima have a beta-distribution.

## Chapter 44

### Redistricting

In this project you can explore ‘gerrymandering’, the strategic drawing of districts to give a minority population a majority of districts<sup>1</sup>.

#### 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:

Output  
[gerry] voters:

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

### 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:

Output  
[gerry] district:

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

Exercise 44.3. Implement a `Population` class that will initially model a whole state.

Code:

Output  
[gerry] populationexample:

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

In addition to an explicit creation, also write a constructor that specifies how many people and what the majority is: 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:

Output  
[gerry] gerryempty:

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

Exercise 44.5. Write methods to extend a Districting:

### 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.

Schematically:

- 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:

**Output**

[gerry] district5:

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

Note: the range for  $p$  given above is not quite correct: for instance, the initial part of the population needs to be big enough to accomodate  $n - 1$  voters.

Exercise 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.

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

### DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

#### 45.1 Basic functions

Refer to section 22.3.

First we set up some basic mechanisms.

Exercise 45.1. 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 45.2. 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 45.3.** 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?

**PART V**

**ADVANCED TOPICS**



## Chapter 46

### Programming strategies

#### 46.1 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.

#### 46.2 Programming: top-down versus bottom up

The exercises in chapter 39 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 39.8.

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...

### 46.2.1 Worked out example

Take a look at exercise 6.9. 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;
```

## 46.3 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.

## 46.4 Documentation

Take a look at Doxygen.

## 46.5 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.

## 46.6 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 47

### Tiniest of introductions to algorithms and data structures

#### 47.1 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.

##### 47.1.1 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 47.1. Write a class that implements a stack of integers. It should have methods

```
void push(int value);  
int pop();
```

##### 47.1.2 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 11.2.)

If you need to do lots of insertions, make a *linked list*. The basic data structure is a `Node`, which contains

1. Information, which can be anything; and
2. A pointer (sometimes called ‘link’) to the next node. If there is no next node, the pointer will be *null*. Every language has its own way of denoting a *null pointer*; C++ has the `nullptr`, while C uses the `NULL` which is no more than a synonym for the value zero.

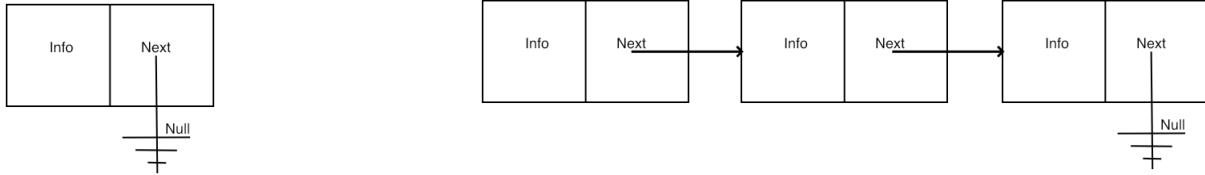


Figure 47.1: Node data structure and linked list of nodes

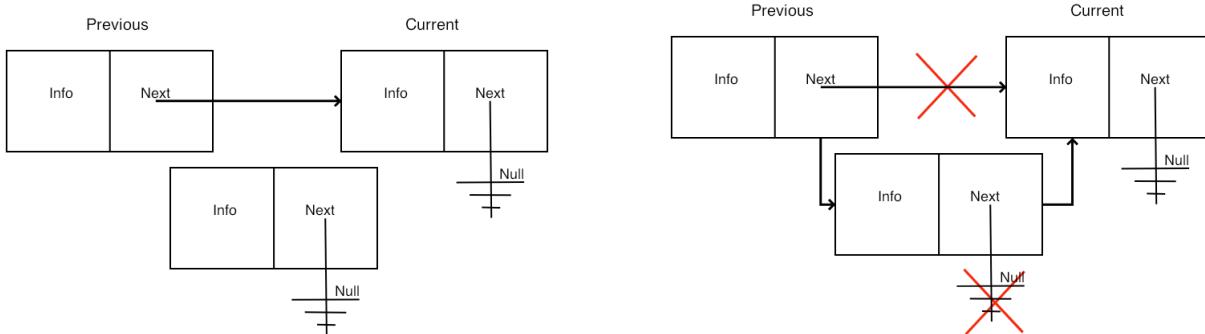


Figure 47.2: Insertion in a linked list

We illustrate this in figure 47.1.

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 47.2.

#### 47.1.2.1 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: Initially null for empty list.

##### *Definition of Node class*

A node has information fields, and a link to another node: A Null pointer indicates the tail of the list.

#### 47.1.2.2 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*

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. (You will get a compiler error if you try to make `current_node` a smart pointer.)

**Exercise 47.2.** Write a function

```
bool List::contains_value(int v);
```

to test whether a value is present in the list.

Try both recursive and iterative.

#### 47.1.2.3 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.

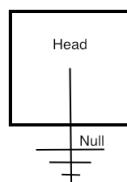
#### *Insert routine design*

We will write functions

```
void List::insert(int value);
void Node::insert(int value);
```

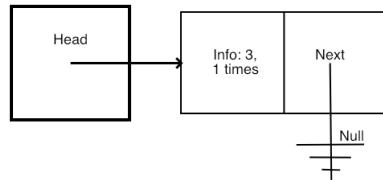
that add the value to the list. The `List::insert` value can put a new node in front of the first one; the `Node::insert` assumes the the value is on the current node, or gets inserted after it.

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.



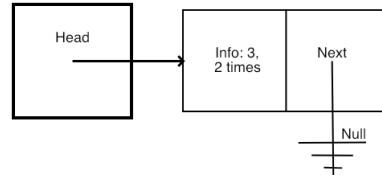
#### **Step 1: dealing with an empty list**

**Exercise 47.3.** Write a `List::length` method, so that this code gives the right output:



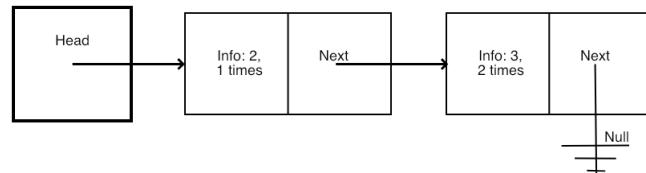
**Step 2: insert the first element**

Exercise 47.4. Next write the case of `Node::insert` that handles the empty list. You also need a method `List::contains` that tests if an item is in the list.



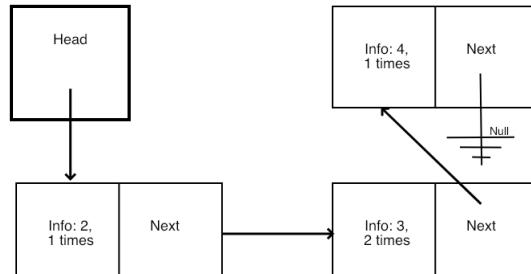
**Step 3: inserting an element that already exists**

Exercise 47.5. 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:



**Step 4: inserting an element before another**

Exercise 47.6. One of the remaining cases is inserting an element that goes at the head. Update your `insert` method to get this to work:



**Step 5: inserting an element at the end**

Exercise 47.7. Finally, if an item goes at the end of the list:

### 47.1.3 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
};
```

## 47.2 Algorithms

This *really really* goes beyond this book.

- Simple ones: numerical
- Connected to a data structure: search

### 47.2.1 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.

#### 47.2.1.1 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 48.1.1) 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.

#### 47.2.1.2 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 47.8. 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?

## 47.3 Programming techniques

### 47.3.1 Memoization

In section 7.5 you saw some examples of recursion. The factorial example could be written in a loop, and there are both arguments for and against doing so.

The Fibonacci example is more subtle: it can not immediately be converted to an iterative formulation, but there is a clear need for eliminating some waste that comes with the simple recursive formulation. The technique we can use for this is known as *memoization*: store intermediate results to prevent them from being recomputed.

Here is an outline.

```
int fibonacci(int n) {
    vector<int> fibo_values(n);
    for (int i=0; i<n; i++)
        fibo_values[i] = 0;
    fibonacci_memoized(fibo_values, n-1);
    return fibo_values[n-1];
```

```
}

int fibonacci_memoized( vector<int> &values, int top ) {
    int minus1 = top-1, minus2 = top-2;
    if (top<2)
        return 1;
    if (values[minus1]==0)
        values[minus1] = fibonacci_memoized(values,minus1);
    if (values[minus2]==0)
        values[minus2] = fibonacci_memoized(values,minus2);
    values[top] = values[minus1]+values[minus2];
    //cout << "set f(" << top << ") to " << values[top] << endl;
    return values[top];
}
```

## **Chapter 48**

### **Complexity**

#### **48.1 Order of complexity**

##### **48.1.1 Time complexity**

Exercise 48.1. 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 48.2. How many operations, as a function of  $n$ , are performed in these two solutions?

##### **48.1.2 Space complexity**

Exercise 48.3. 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 48.4. How much space do the two solutions require?



## **PART VI**

### **INDEX AND SUCH**



## Chapter 49

### Index

#ifdef, 217  
#ifndef, 217  
#pragma once, 217

abs, 42  
abstraction, 61  
algorithm, 42  
all (Fortran keyword), 311  
allocate, 308  
any (Fortran keyword), 311  
Apple, 23  
argument  
    actual, 284  
    default, 72  
    dummy, 284  
array, 121  
    associative, 230  
    assumed-shape, 308  
    automatic, 303, 308  
    bounds  
        checking, 125  
    index, 121  
    initialization, 304  
    rank, 305  
    static, 303  
    subscript, 121  
array, 124  
assignment, 36  
Associated (Fortran keyword), 319  
auto, 122

bad\_alloc, 226  
bad\_exception, 226  
basic\_ios, 164  
begin, 240

bit\_size, 265  
bottom-up, 367  
break, 55  
bubble sort, 137, 377  
bug, 17

C  
    parameter passing, 197–201  
    pointer, 193–201, 257  
    preprocessor, 261  
    string, 152, 257  
C preprocessor, see preprocessor  
C++, 249  
    C++17, 231  
    Core Guidelines, 370  
    C++11, 124  
    c\_sizeof, 265  
    Caesar cypher, 151  
    call, 280  
    call-back, 247  
    calling environment, 173  
    capture, 242  
    case sensitive, 34  
    cast, 42, 243  
    cerr, 257  
    char, 149  
    Character (Fortran keyword), 264  
    cin, 40  
    cin, 257  
class  
    abstract, 109  
    base, 107  
    derived, 107  
    iteratable, 240  
    class, 95, 300

Close (Fortran keyword), 323, 326  
closure, 81  
cmath, 40, 42, 242  
code  
  duplication, 63  
  maintainance, 369  
code duplication, 61  
code reuse, 63  
column-major, 324  
Common (Fortran keyword), 278  
compilation  
  separate, 206, 296, 350  
compiler, 24, 31  
  and preprocessor, 215  
  optimization, 235  
compiling, 24  
Complex (Fortran keyword), 264  
complex numbers, 35, 229  
conditional, 45  
connected components, see graph, connected  
const  
  reference, 235  
const, 235, 237  
const\_cast, 246  
constructor, 96, 181, 257  
  copy, 103, 236  
  default, 96  
container, 229  
Contains (Fortran keyword), 278, 279  
contains  
  for class functions, 299  
  in modules, 296  
contains, 285  
continuation character, 263  
continue, 56  
cout, 39, 257  
Cshift, 310  
  
datatype, 34  
deallocate, 309  
debugger, 227  
definition vs use, 79  
dereference, 194  
derefencing, 135  
destructor, 82, 105  
  at end of scope, 81

dimension (Fortran keyword), 303  
do  
  concurrent, 51  
do (Fortran keyword), 273  
do concurrent, 312  
do loop  
  implicit, 324  
  and array initialization, 304  
  implied, 274  
Dot\_Product, 310  
dynamic  
  programming, 362  
dynamic\_cast, 244  
  
efficiency gap, 362  
emacs, 23, 23, 262  
end, 240, 262  
eof, 163  
errno, 226  
error  
  compile-time, 33  
  run-time, 33  
  syntax, 33  
exception  
  catch, 224  
  throwing, 224  
exception, 226  
executable, 24, 32, 33  
exit, 274  
expression, 36  
extent  
  of array dimension, 307  
external, 286  
  
F90, 261  
false, 39, 40  
Fasta, 363  
Fastq, 364  
file  
  binary, 31  
  executable, 206  
  handle, 105  
  object, 206, 296  
  source, 31  
floating point, 37  
fmtlib, 257  
for

indexed, 122  
range-based, 122  
`forall`, 311  
`Format` (Fortran keyword), 323  
Fortran  
    90, 261  
    case ignores, 262  
    comments, 263  
    forward declaration, 205  
        of classes, 81  
        of functions, 81  
    friend, 110  
    function, 61, 281, 282  
        argument, 64  
        arguments, 64  
        body, 64  
        call, 62  
        defines scope, 65  
        definition, 62  
        parameter, 64  
        parameters, 64  
        result type, 64  
        standard, 65  
    function try block, 225  
functional programming, 66, 173  
  
`g++`, 31  
`gdb`, 227  
gerrymandering, 359  
`get`, 185  
`getline`, 163  
`gfortran`, 262  
GNU, 31  
Goldbach conjecture, 339  
Google, 353  
graph  
    connected, 354  
    diameter, 354  
  
has-a relation, 106  
`hdf5`, 323  
header, 350  
header file, 206, 215  
    and global variables, 209  
    treatment by preprocessor, 209  
hexadecimal, 193  
Holmes  
    Sherlock, 152  
homebrew, 23  
host association, 278  
  
I/O  
    formatted, 323  
    list-directed, 323  
    unformatted, 323  
`icpc`, 31  
`if` (Fortran keyword), 269  
`ifort`, 262  
index  
    section, 304  
`Inf`, 38  
inheritance, 107  
initialization  
    variable, 34  
initializer list, 98  
inline, 285  
`Integer` (Fortran keyword), 264  
`Interface` (Fortran keyword), 279  
interface, 282, 285, 286  
is-a relation, 107  
`is_eof`, 164  
`is_open`, 164  
`isnan`, 225  
`iso_c_binding`, 265  
iteration  
    of a loop, see loop, iteration  
iterator, 134, 230  
  
keywords, 34  
`kind`, 264, 265  
  
label, 325  
lambda, see closure, 247  
    expression, 242  
`Lbound`, 307  
`len` (Fortran keyword), 291  
lexicographic ordering, 54  
linear regression, 358  
linker, 206, 296  
Linux, 23  
list  
    linked, 371–374  
        in Fortran, 319–322  
    Logical (Fortran keyword), 264, 270

longjmp, 226  
loop, 51  
    body, 52  
    counter, 51  
    for, 51  
    header, 52  
    inner, 54  
    iteration, 52  
    nest, 54  
    outer, 54  
    variable, 52  
    while, 51  
lvalue, 247  
  
macports, 23  
Make, 206  
makefile, 207, 350  
malloc, 181, 196, 199, 201, 257  
map, 230  
MatMul, 310  
max, 42  
MaxLoc (Fortran keyword), 309  
MaxVal, 309  
member  
    of struct, 87  
memoization, 361, 377  
memory  
    leak, 105, 199, 199, 309  
memory leak, 184  
memory leaking, 201  
method, 98  
    abstract, 109  
    overriding, 108  
methods (of an object), 95  
Microsoft  
    Windows, 23  
    Word, 17  
MinLoc, 310  
MinVal, 309  
Module (Fortran keyword), 278, 279  
move semantics, 249  
  
namespace, 211  
NaN, 38  
new, 198, 200, 201  
Newton's method, 67  
noexcept, 226  
  
NULL, 186  
NULL, 152, 371  
null terminator, 257  
Nullify (Fortran keyword), 319  
nullptr, 181, 186, 244, 371  
nullptr\_t, 186  
  
object file, see file, object  
once, see #pragma once  
Open (Fortran keyword), 323  
open, 163  
open (Fortran keyword), 326  
operators  
    shortcut, 41  
Optional (Fortran keyword), 285  
output  
    binary, 326  
    raw, 326  
    unformatted, 326  
overloading, 103  
override, 109  
  
package manager, 23  
Pagerank, 353  
parameter, see also function, parameter  
    formal, 285  
    input, 68  
    output, 68, 230  
    pass by value, 66  
    passing, 66  
        by reference, 173, 181, 257  
        by value, 173  
    passing by reference, 66, 68  
        in C, 68  
        passing by value, 66  
        throughput, 68  
Pascal's triangle, 137  
pass by reference, see parameter, passing by reference  
pass by value, see parameter, passing by value  
PETSc, 226  
pointer, 110  
    arithmetic, 135, 196  
    bare, 186, 257, 373  
    dereferencing, 317  
    null, 186, 371  
    unique, 186

void, 246  
    in C++, 187  
pointer, 317  
pop, 371  
precedence, 47  
preprocessor  
    and header files, 209  
    conditionals, 216–217  
macro  
    parametrized, 216  
macros, 215–216  
Present (Fortran keyword), 285  
Print (Fortran keyword), 323  
print, 323  
printf, 257  
private, 297  
procedure  
    internal, 278  
procedure, 300  
procedures  
    internal, 285  
    module, 285  
Product, 309  
program  
    statements, 32  
programming  
    dynamic, 361  
    parallel, 51  
protected, 108, 297  
prototype, 205  
public, 297  
punch cards, 261  
push, 371  
push\_back, 126  
putty, 23  
python, 21  
quicksort, 377  
rand, 72  
RAND\_MAX, 72  
random number  
    generator, 72  
    seed, 72  
Read (Fortran keyword), 323  
Real (Fortran keyword), 264  
recurrence, 313  
recursion, see function, recursive  
    depth, 71  
    mutual, 71  
Recursive (Fortran keyword), 281  
reference, 67, 173  
    argument, 181, 257  
    const, 173, 177  
    to class member, 175  
    to class member, 174  
reference count, 186  
reinterpret\_cast, 243  
reserved words, 35  
RESHAPE (Fortran keyword), 307  
result (Fortran keyword), 281  
return, 64  
    makes copy, 177  
return, 280  
root finding, 58  
roundoff error analysis, 38  
runtime error, 223  
rvalue, 247  
    reference, 249  
Save (Fortran keyword), 278  
scanf, 257  
scope, 64, 79  
    dynamic, 81  
    in conditional branches, 48  
    lexical, 79, 81  
    of function body, 65  
segmentation fault, 125  
select (Fortran keyword), 270  
selected\_int\_kind, 264  
selected\_real\_kind, 264  
setjmp, 226  
shape  
    of array, 307  
shared\_ptr, 257, 372  
Single Source Shortest Path, 354  
SIR model, 350  
size, 307  
size\_t, 244  
sizeof, 199  
smartphone, 17  
source  
    format

fixed, 261  
free, 261  
source code, 24  
SPREAD (Fortran keyword), 307  
srand, 72  
stack, 71, 82, 308, 371  
    overflow, 71, 308  
Standard Template Library, 229  
statement functions, 285  
static\_cast, 244, 246  
std::any, 187  
stop (Fortran keyword), 262  
storage\_size, 265  
string, 150  
    concatenation, 150  
    null-terminated, 152  
    size, 150  
string, 257  
struct  
    denotation, 90  
struct, 87, 230  
subprogram, see function  
Sum, 309  
swap, 249  
switch, 48  
syntax  
    error, 223  
system\_clock (Fortran keyword), 329  
  
target, 318  
templates  
    and separate compilation, 208  
test-driven development, 370  
testing, 369  
this, 110, 186  
timer  
    resolution, 329  
top-down, 367  
Transpose, 310  
trim (Fortran keyword), 291  
  
true, 39, 40  
tuple, 230  
    denotation, 231  
type (Fortran keyword), 293  
  
Ubound, 307  
unique\_ptr, 186, 257, 372  
unit, 325  
unit testing, 370  
Unix, 23  
use, 296  
use (Fortran keyword), 295  
  
valgrind, 227  
values  
    boolean, 39  
variable, 34  
    assignment, 34  
    declaration, 34, 35  
    global, 208  
        in header file, 209  
    initialization, 36  
    lifetime, 80  
    numerical, 35  
    shadowing, 80  
    static, 81, 277  
vector, 211  
vector, 121, 123, 257, 371  
vi, 23  
Virtualbox, 23  
Visual Studio, 23  
VMware, 23  
void, 62, 64  
  
where (Fortran keyword), 310  
while, 56  
Write (Fortran keyword), 323  
write (Fortran keyword), 325  
  
Xcode, 23  
XQuartz, 23