C++ lecture notes

François Fleuret
<francois.fleuret@epfl.ch>

November 21, 2005

Note

This document is based on a C++ course given at the University of Chicago in spring of 2001 and was modified for a course at EPFL in fall of 2004. It is still a work in progress and needs to be polished to be a reference text.

The tools for this course are free-softwares. Mainly the GNU/Linux operating system, the GNU C++ compiler, the emacs text editor, and a few standard UNIX commands.


```
$Id: cpp-course.tex,v 1.33 2004/12/19 21:04:27 fleuret Exp $
```

Contents

1 Memory, CPU, files **1**
- 1.1 Memory, files, CPU and compilation . . . 1
  - 1.1.1 Memory . . . 1
  - 1.1.2 Data types . . . 2
  - 1.1.3 Signal quantification . . . 2
  - 1.1.4 File system . . . 3
  - 1.1.5 Size orders of magnitude . . . 3
- 1.2 CPU . . . 3
  - 1.2.1 What is a CPU . . . 3
  - 1.2.2 Speed orders of magnitude . . . 4
- 1.3 Compilation . . . 5
  - 1.3.1 Role of compilation . . . 5
  - 1.3.2 Example . . . 5
- 1.4 Object-Oriented Programming . . . 7

2 Shell and basic C++ **9**
- 2.1 GNU/Linux . . . 9
  - 2.1.1 What is Linux . . . 9
  - 2.1.2 What is Open-Source . . . 9
  - 2.1.3 Tools for this course . . . 11
- 2.2 Shell and simple file management . . . 11
  - 2.2.1 File names and paths . . . 11
  - 2.2.2 Shell . . . 12
  - 2.2.3 Basic commands . . . 12
  - 2.2.4 References for documentation . . . 14
- 2.3 First steps in C++ . . . 14
  - 2.3.1 Data types . . . 14
  - 2.3.2 A simple example of variable manipulation . . . 15
  - 2.3.3 Variable naming conventions . . . 16
  - 2.3.4 Streams, include files . . . 16
  - 2.3.5 The sizeof operator . . . 17
  - 2.3.6 The `if` statement . . . 17
  - 2.3.7 The `for` statement . . . 18
  - 2.3.8 The `while` statement . . . 19

  - 2.3.9 The `do { } while` statement . . . 19
  - 2.3.10 The `continue` statement . . . 20
  - 2.3.11 The `switch` / `case` statements . . . 20
  - 2.3.12 Computation errors with floating point counters . . . 21
- 2.4 An example of extreme C syntax . . . 22

3 Expressions, variable scopes, functions **23**
- 3.1 Expressions . . . 23
- 3.2 Arithmetic operators . . . 23
  - 3.2.1 List of operators . . . 23
  - 3.2.2 Operators depend on the types of the operands . . . 24
  - 3.2.3 Implicit conversions . . . 24
  - 3.2.4 Arithmetic exceptions . . . 25
  - 3.2.5 Boolean operators . . . 26
  - 3.2.6 Comparison operators . . . 27
  - 3.2.7 Assignment operator . . . 27
  - 3.2.8 Increment and decrement operators . . . 27
  - 3.2.9 Precedence of operators . . . 28
  - 3.2.10 Grammar, parsing and graph of an expression . . . 29
  - 3.2.11 Summary . . . 29
- 3.3 lvalue vs. rvalue . . . 30
- 3.4 Scopes of variables . . . 30
- 3.5 Functions . . . 31
  - 3.5.1 Defining functions . . . 31
  - 3.5.2 `void` return type . . . 32
  - 3.5.3 Argument passing by value . . . 33
  - 3.5.4 Argument passing by reference . . . 33
  - 3.5.5 Recursive function call . . . 34
  - 3.5.6 Stopping condition . . . 35
- 3.6 The `abort()` function . . . 35

4 Arrays and pointers, dynamic allocation **37**
- 4.1 Arrays and pointers . . . 37
  - 4.1.1 Character strings . . . 37
  - 4.1.2 Built-in arrays . . . 37
  - 4.1.3 Index of arrays, the `[ ]` operator, out of bounds exception 38
  - 4.1.4 Pointers, the `*`, and `&` operators . . . 39
  - 4.1.5 Pointers to pointers to pointers to 39
  - 4.1.6 Dereference operator `*` . . . 40
  - 4.1.7 Pointers to arrays . . . 41
  - 4.1.8 Pointers do not give information about pointed array sizes 41
  - 4.1.9 Box and arrows figures . . . 42
  - 4.1.10 The `main` function's parameters . . . 42
  - 4.1.11 Adding integers to pointers . . . 43
- 4.2 Dynamic allocation . . . 44
  - 4.2.1 Why ? How ? . . . 44

4.2.2 Dynamic arrays . . . 45
4.2.3 Test of a null pointer . . . 46
4.2.4 A non-trivial example using dynamic memory allocation . 46
4.2.5 Dynamically allocated bi-dimensional arrays . . . 47
4.2.6 What is going on inside: the stack and the heap . . . 48
4.3 Miscellaneous . . . 49
4.3.1 Declaration vs. definition . . . 49
4.3.2 The `const` statements . . . 50
4.3.3 The `enum` type . . . 51
4.3.4 The `break` statement . . . 51
4.3.5 Bitwise operators . . . 52
4.3.6 The comma operator . . . 52

5 War with the bugs 55
5.1 Preamble . . . 55
5.2 The Bug Zoo . . . 55
5.2.1 The program crashes: `Segmentation fault` . . . 55
Unauthorized memory access . . . 56
Incorrect system call . . . 57
5.2.2 The program crashes: `Floating point exception` . . . 57
5.2.3 The program never stops . . . 58
5.2.4 The program uses more and more memory . . . 58
5.2.5 The program does not do what it is supposed to do . . . 58
5.3 How to avoid bugs . . . 60
5.3.1 Write in parts . . . 60
5.3.2 Good identifiers . . . 60
5.3.3 Use constants instead of numerical values . . . 60
5.3.4 Comment your code . . . 61
5.3.5 Symmetry and indentation . . . 61
5.3.6 Use a DEBUG flag . . . 62
5.4 How to find bugs . . . 63
5.4.1 Print information during execution . . . 63
5.4.2 Write the same routine twice . . . 63
5.4.3 Heavy invariants . . . 64
5.5 Anti-bug tools . . . 64
5.5.1 gdb . . . 64
5.5.2 Valgrind . . . 65

6 Homework 69

7 Cost of algorithm, sorting 73
7.1 Big-O notation . . . 73
7.1.1 Why ? How ? . . . 73
7.1.2 Definition of $O(.)$ . . . 74
7.1.3 Some $O(.)$ . . . 74
7.1.4 Summing $O(.)$s . . . 74

7.1.5 Combining $O(.)$s . . . 75
7.1.6 Family of bounds . . . 75
7.1.7 Some examples of $O(.)$ . . . 76
7.1.8 Estimating the cost of an algorithm . . . 76
Succession of statements . . . 76
Conditional execution . . . 77
Loops . . . 77
7.1.9 Cost and recursion . . . 77
7.1.10 Average cost . . . 78
7.2 Some algorithms . . . 79
7.2.1 Searching a value in a sorted array . . . 79
7.2.2 Pivot sort . . . 80
7.3 Simple questions . . . 81
7.4 Fusion sort . . . 81
7.5 Quick sort . . . 81
7.6 Strategies when two parameters are involved ? . . . 83

8 Creating new types 85
8.1 Preamble . . . 85
8.2 A simple example . . . 85
8.3 Pointers to defined types, and the `->` operator . . . 86
8.4 Operator definitions, a complex class . . . 86
8.5 Passing by value vs. passing by reference . . . 87
8.6 Some timing examples . . . 88

9 Object-Oriented programming 91
9.1 Intro . . . 91
9.2 Vocabulary . . . 92
9.3 Protected fields . . . 92
9.4 Methods . . . 93
9.5 Calling methods . . . 93
9.6 Some memory figures . . . 94
9.7 Separating declaration and definition . . . 95
9.8 Protection of data integrity . . . 95
9.9 Abstraction of concepts . . . 96
9.10 Constructors . . . 97
9.11 Default constructor . . . 98
9.12 Destructor . . . 99
9.13 Tracing precisely what is going on . . . 99
9.14 The member operators . . . 100
9.15 Summary for classes . . . 102

10 Homework 103

11 Detail of class definitions 105
11.1 Example . . . 105

11.2 An "integer set" example . . . 106
11.3 The `const` keyword . . . 108
11.4 The `this` pointer . . . 109
11.5 The `=` operator vs. the copy constructor . . . 109
11.6 Default copy constructor and default `=` operator . . . 110
11.7 Some memory figures . . . 112
11.8 A matrix class . . . 112

12 More details about class definitions 117
12.1 Back to operators . . . 117
12.2 Implicit conversion . . . 118
12.3 Private methods . . . 120
12.4 Hiding the copy constructor . . . 120
12.5 A linked list class . . . 121
12.5.1 The `Node` class . . . 122
12.5.2 The `LinkedList` class . . . 123
12.6 The graphical library . . . 126

13 More about methods 129
13.1 Rvalues, lvalues, references, and `const` qualifier . . . 129
13.2 Methods can be called through standard functions . . . 130
13.3 Overloading the `<<` operator . . . 130
13.4 Overloading the `>>` operator . . . 131
13.5 An example about what has been said before . . . 132
13.6 A bit more about streams : output formats . . . 133
13.7 A bit more about streams : files . . . 133
13.8 Inline functions . . . 133
13.9 First steps with inheritance . . . 134
13.10Adding methods . . . 134
13.11Adding data fields . . . 135
13.12Multiple inheritance . . . 136
13.13Tricky issues with multiple inheritance . . . 136

14 Homework 139
14.1 Costs and big-O (10 points) . . . 139
14.2 Quick-sort (30 points) . . . 140
14.3 The Mandelbrot set (30 points) . . . 140

15 Inheritance 143
15.1 Adding member data field and functions . . . 144
15.2 Syntax to inherit . . . 144
15.3 Syntax to call constructors . . . 145
15.4 An example . . . 146
15.5 Tracing what's going on "inside" . . . 148
15.6 The `protected` keyword . . . 149
15.7 Hiding the superclass . . . 149

15.8 Ambiguities between different members with the same name . . . 150
15.9 method overload and calls . . . 151
15.10What's going on in the memory ? . . . 152
15.11Memory addresses can change! . . . 154

16 Exercises 155
16.1 Find the bug! . . . 155
16.2 Find the bug! . . . 155
16.3 Find the bug! . . . 156
16.4 Find the bug! . . . 157
16.5 Find the bug! . . . 158
16.6 What is printed ? . . . 159
16.7 What is printed ? . . . 160
16.8 What is printed ? . . . 160
16.9 What is printed ? . . . 161

17 Exercices 163
17.1 Find the bug! . . . 163
17.2 Find the bug! . . . 163
17.3 Find the bug! . . . 164
17.4 Find the bug! . . . 164
17.5 Find the bug! . . . 164
17.6 When does it bug ? . . . 165
17.7 Find the bug! . . . 165
17.8 Find the bug! . . . 165
17.9 What is printed ? . . . 166
17.10What is printed ? . . . 166
17.11Non trivial inheritance . . . 167

18 Homework 169
18.1 Various questions (20 points) . . . 169
18.2 A polynomial class (80 points) . . . 169

19 Mid-term preparation 171
19.1 Variables, types, scope, default initialization . . . 171
19.2 Variables, pointers, dynamic allocation . . . 171
19.3 Expressions, operators, implicit conversion, precedence . . . 172
19.4 `if`, `while`, `for`, `while/do` . . . 173
19.5 Declaring and defining functions . . . 173
19.6 Parameters by value or by reference . . . 174
19.7 Functions, recursion . . . 174
19.8 Algorithm costs, Big-O notation . . . 174
19.9 Sorting algorithms . . . 175
19.10`class` keyword . . . 175
19.11Constructors / destructor, `=` operator . . . 176
19.12A matrix class . . . 176

19.13Inheritance 179

20 Homework **181**
20.1 Introduction 181
20.2 A window to draw lines in the complex plane (40 points) 181
20.3 A window to draw a mesh in the complex plane (60 points) . . . 182

21 Virtual methods **185**
21.1 One major weakness of inheritance 185
21.2 Using virtual methods 186
21.3 Precisions about virtual methods 187
21.4 Pure virtual methods 188
21.5 Pure virtual classes 189
21.6 Pointers to virtual classes 190
21.7 Non-trivial example 193

22 Boxes and arrows **197**

23 References and virtual classes **203**
23.1 References to virtual classes 203
23.2 References, `const` qualifier, and temporary objects 203
23.3 Exercises 204
23.3.1 What does it print ? 204
23.3.2 What does it do ? 205
23.3.3 What does it do ? 205

24 Homework **207**
24.1 Z-buffer 207
24.2 Introduction 207
24.3 Some math 209
24.3.1 Intersection with a ball 209
24.3.2 Intersection with a triangle 209
24.4 Class to write 210
24.5 Some maths 210
24.5.1 Intersection between a line and a plane 211

25 Design patterns : sets and iterators **213**
25.1 Example : integer sets and iterators 213
25.2 Exercices 213
25.3 Economy of CPU usage : smart copies 215
25.4 Example : back to mappings 220
25.5 Cast 223
25.6 `dynamic_cast<type *>` 224
25.7 Summary about inheritance 224
25.8 Weirdness of syntax 225
25.8.1 Explicit call to the default constructor 225

25.8.2 Hidden methods 226

26 Strings and more iterators **227**
26.1 The `string` class 227
26.1.1 Introduction 227
26.1.2 Example 227
26.1.3 Principal methods and operators 228
26.1.4 example 228
26.2 Exercises 229
26.2.1 A small iterator 229
26.2.2 Write the class 230
26.2.3 What does it do ? 231
26.2.4 Write the class 232

27 Homework **235**
27.1 Ray-tracing 235
27.2 Introduction 235
27.3 Description of the algorithm 235
27.4 Some maths 237
27.4.1 Parameterization of a ray 237
27.4.2 Sphere 237
27.4.3 Chessboard 237
27.5 OO organization 238
27.6 Example of `main` 239

28 Templates **241**
28.1 Introduction 241
28.2 Examples of template 242
28.3 Syntax 242
28.4 Template class 243
28.5 Inheritance from template class 243
28.6 Separate definition of methods 244
28.7 Template compilation type-checking 245
28.8 Remark about compilation 246
28.9 Exercise 246
28.9.1 Write a sum function 246
28.9.2 Write a template stack class 247

29 Tree structures **249**
29.1 Introduction 249
29.2 A simple implementation 249

30 Summary of everything **255**
30.1 Variables, types, scope, default initialization 255
30.2 Variables, pointers, dynamic allocation 255
30.3 Expressions, operators, implicit conversion, precedence 256

30.4 `if`, `while`, `for`, `while/do` 257
30.5 Declaring and defining functions 257
30.6 Parameters by value or by reference 258
30.7 Functions, recursion 258
30.8 Algorithm costs, Big-O notation 258
30.9 Sorting algorithms 259
30.10OO programming 259
30.11`class` keyword 259
30.12Constructors / destructor, `=` operator 260
30.13Inheritance 260
30.14virtual methods and classes 261
30.15Exercises 261

A Midterm Exam **265**
A.1 Cost (15 points) 265
A.2 Some boxes and arrows! (15 points) 265
A.3 Find the bug!!! (25 points) 266
A.4 What does it print ? (25 points) 266
A.5 Class design (20 points) 267

B Final Exam **269**
B.1 Some boxes and arrows (15 points) 269
B.2 What does it print ? (25 points) 270
B.3 Class design (25 points) 271
B.4 Virtual class design (35 points) 272

Chapter 1

Memory, CPU, files

1.1 Memory, files, CPU and compilation

1.1.1 Memory

- Used to store informations ;
- 0 / 1 representation with electric voltage ;
- the memory or a file is a gigantic set of **bytes**, each of them composed of 8 **bits** ;
- each of the bytes has an index called its **address**.

	7	6	5	4	3	2	1	0
#0	0	1	1	1	0	0	0	0
#1	1	0	1	1	1	0	0	1
#2	1	0	0	0	0	0	0	0
#3	1	0	1	0	1	0	1	1
#4	0	1	1	0	0	0	0	1
...				...				

Figure 1.1: The memory is a succession of binary values called **bits**. A group of 8 such bits is called a **byte** and can be indexed by its **address**.

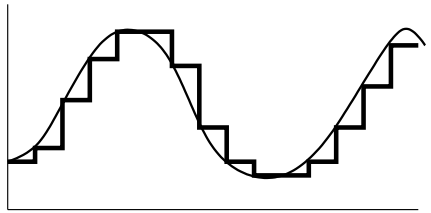


Figure 1.2: Quantification consist of transforming a signal of continuous values, for instance a sound signal, into a succession of integer values.

1.1.2 Data types

Bytes can represent either integer or floating point numbers, images, sounds, texts, programs, etc. We call **type** the semantic associated to a group of bytes.

For example, a byte alone can carry $256 = 2^8$ different values. Depending on how we consider it, it can represent either an integer value between 0 and 255, or an integer value between -128 and $+127$, or a character (in that case we use a table to define the correspondence between the bit configurations and the characters). It could also be considered as two integers between 0 and 15.

Bytes can be grouped, for example to represent larger integers. The standard `integer` in `C++` on a x86 processor is composed with 4 bytes, thus 32 bits, and can encode a value between -2147483648 and 2147483647.

The address of a byte in the memory is itself an integer, and can also be represented in the memory. Because of the binary representation, the most convenient notation for memory address is in **hexadecimal**, i.e. base 16. In this base, the digits are $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$, and each of them represents 4 bits. For example $26 = 16 + 10$ will be denoted $1a$ in hexadecimal.

1.1.3 Signal quantification

Finally everything has to be represented with integers. To encode images or sounds, softwares use **quantifications**. For example, the standard CD encoding uses a $44khz$ sampling of the volume with 16 bits.

Similarly, images are represented as map of small squares called pixels, each of them having a uniform color defined by its red, green and blue components. Today standard encode each components with one byte, which lead to the famous

$24 - bits$ color encoding.

This is as simple as it sounds: in the computer memory, an image is encoded as a succession of groups of three bytes, each one of those triplets corresponding to the three component red, green and blue of a point on the screen.

1.1.4 File system

To store information when the computer is turned off, or to manipulate larger set of bytes, one uses magnetic storage devices. The most common are the **hard disks**. Informations are stored under the forms of **files**, each of them having a name and a size.

A file is very similar to the memory : a set of bytes indexed by their positions.

- Files can be very large, up to hundred times the memory ;
- the file access is very slow compared to memory access.

In practice the files are organized on a hard disk with **directories** (also called **folder** under Microsoft or Apple Operating systems). A directory can contains files and other directories (we can imagine it as a file containing a list of names of files and directories). This leads to a very powerful hierarchical organization.

1.1.5 Size orders of magnitude

Because a very large number of bytes is required to encode useful objects, the standard units go up to very large numbers : 1Kbyte = 1024 bytes, 1Mbytes = 1024 Kbytes, 1Gbyte = 1024 Mbytes, 1Tbyte = 1024 Gbytes.

1.2 CPU

1.2.1 What is a CPU

The **Central Processing Unit** (CPU) is a very fast electronic device, able to read from and write into the memory, and to do arithmetic operations. The native language of a CPU is called the **assembler**. A CPU has a frequency, which indicates roughly how many operations it can do each second.

RAM memory	128 Mbyte ($65)
CD-Rom (DVD-Rom)	650 Mbytes (4 to 16 Gbytes)
hard disk	30 Gbyte ($150)
phone modem	56 Kbyte/s
dsl	128 Kbyte/s ($50 / month)
Optic fiber backbones	$\simeq$ 5 Gbyte/s (record is $\simeq$ 400 Gbyte/s)
text (200 pages of 2,500 characters)	$\simeq$ 500 Kbytes
image (1024 x 768 x 16M colors)	$\simeq$ 2 Mbyte (jpeg $\simeq$ 250 Kbytes)
sound (44khz stereo)	$\simeq$ 150 Kbyte/s (mp3 $\simeq$ 16 Kbyte/s)
movie (640 x 480 x 16M colors x 25hz)	$\simeq$ 20 Mbyte/s (DivX $\simeq$ 125 Kbytes/s)

Table 1.1: Order of magnitudes of memory, bandwidth and various digital objects.

The well known CPUs today (in 2001) are the Intel Pentium, AMD athlon, and PowerPC.

Good programming sometime requires to have a precise idea of the inner functioning of a CPU. Keep in mind :

1. Certain operations take far more time than others ones. for instance floating point computations ;
2. the memory is 10 times slower than the CPU, which means that reading from and writing to memory will be the expensive operations ;
3. a faster and expensive intermediate memory called **cache memory** stores information frequently used by the CPU (it simply keeps a copy of the last bytes read from the main memory).

The principle of cache memory will make a programmer prefer an algorithm that concentrates its accesses to a small part of the memory at a given time instead of "jumping around" all the time.

1.2.2 Speed orders of magnitude

The order of magnitude today (2001) of either Athlon, PowerPC or Pentium is between 500Mhz and 1Ghz. The memory is more between 100Mhz and 200Mhz.

Compression of a 3 min song in mp3	$\simeq$ 3 minutes with a P600
Compression of a 90 min movie in divx	$\simeq$ 15 hours with a P600
Neural network training	Up to days

Table 1.2: Speed of processors and computation time.

1.3 Compilation

1.3.1 Role of compilation

The compilation operation consists of translating a program written in a complex and human-readable language like `C++`, `C`, `PASCAL`, `ADA`, etc. into assembler code, which can be directly understood by the CPU.

Practically, a compiler is itself a program which reads **source files** written by a human, for instance in `C++`, and generate **object files** which contains assembler code.

Why using a language like `C++` and a compiler instead of writing assembler :

1. CPUs have different assembler codes, so using a compiler allows to use the same language on different CPUs ;
2. compilers are able to spot in your program things that can be rewritten differently and more efficiently ;
3. languages like `C++` or Java allow to manipulate complex structures of data in an abstract way.

Finally, to write a `C++` program, the programmer first has to write a file containing the `C++` source code itself. Then, he runs the compiler, giving the name of the source file and the name of the object file as parameters. The resulting object file can then be run as a program itself.

1.3.2 Example

Below is an example of a shell session to create, compile and execute a program. The `emacs` command is a text editor used to create the file `something.cc`. The `g++` command is the compiler, which generates the object file `something` from the source file `something.cc`. Figure 1.3 shows the `emacs` window while editing the source file.

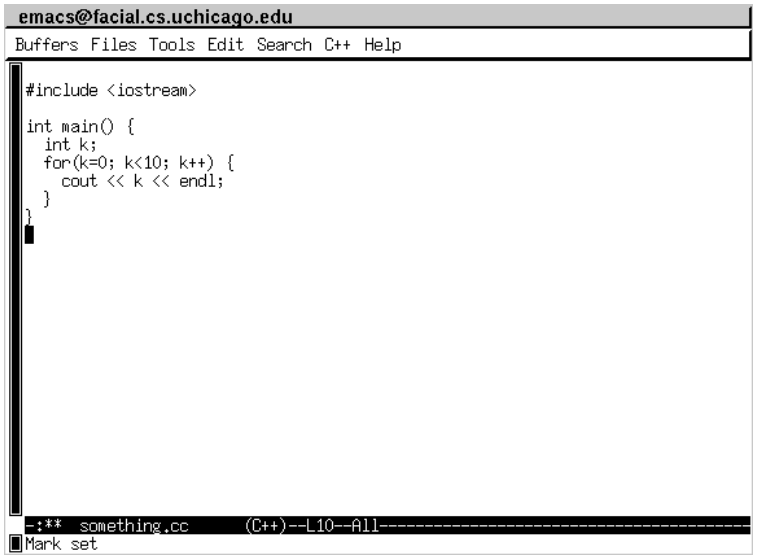


Figure 1.3: Window of the `emacs` text editor while editing the `something.cc` file.

```
> emacs something.cc
> g++ -o something something.cc
> ./something
0
1
2
3
4
5
6
7
8
9
>
```

1.4 Object-Oriented Programming

The "object approach", which is the fundamental idea in the conception of `C++` programs, consists of building the program as an interaction between objects :

1. In all part of the program that use a given object, it is defined by the **methods** you can use on it ;
2. you can take an existing object type (called a **class**) and add to it data and methods to manipulate it.

The gains of such an approach are :

1. Modularity : each object has a clear semantic (`Employer` or `DrawingDevice`), a clear set of methods (`getSalary(), getAge()`, or `drawLine(), drawCircle()`.
2. Less bugs : the data are accessed through the methods and you can use them only the way to object's creator wants you to.
3. Reutilisability : you can extend an existing object, or you can build a new one which could be used in place of the first one, as long as it has all the methods required (for example the `Employer` could be either the CEO or a worker, both of them having the required methods but different data associated to them, `DrawingDevice` could either be a window, a printer, or anything else).

Chapter 2

Shell and basic C++

2.1 GNU/Linux

2.1.1 What is Linux

1. A big piece of software called the **kernel** which run as soon as you turn on a Linux computer, and is able to deal with all devices (hard disk, keyboard, mouse, network cards, etc.)
2. **X-Window**, which controls the graphical display ;
3. a large list of programs that allow you to edit texts, manage your files, compile source codes, do network administration, etc.

The main goal of this course is to learn C++, not to go into the specificities of neither Linux or X-Window. We will focus on standard C++ which can be used on other operating systems (Windows, MacOS, BeOS, etc.)

2.1.2 What is Open-Source

Note that all softwares we will use in this course are free **open-source softwares**. Which means :

1. you can get them for free ;
2. you can get their source codes ;

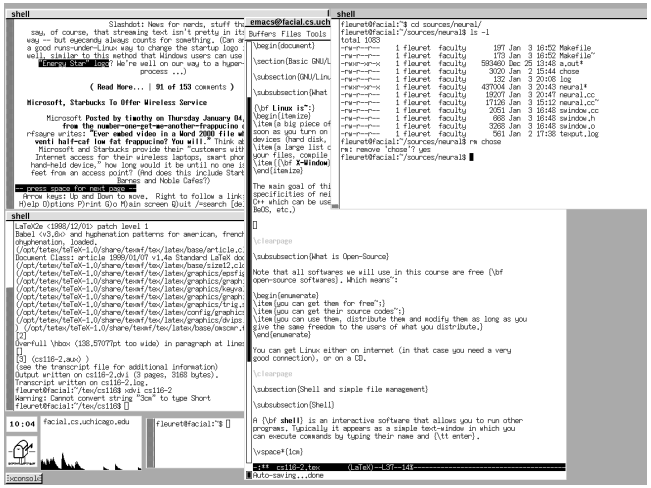


Figure 2.1: A working screen of a GNU/Linux computer.

3. you can use them, distribute them and modify them as long as you give the same freedom to the users of what you distribute.

The main license for such software is the GPL or the BSD one. You can get Linux either on Internet (in that case you need a very good connection), or on a CD.

The Linux kernel was originally written by a student called Linus Torvald, and has been since then heavily improved by thousands of programmers. Its existence proves by itself the incredible power of the open source model of development.

2.1.3 Tools for this course

The main setup for this course will be a GNU/Linux computer, the gcc compiler, the emacs text editor and the standard shell commands. Since all this is very standard, any Linux distribution should be fine.

MS-Windows users who are reluctant to install this free open-source operating system have two options:

1. Use a one-CD live distribution, for instance Knoppix[1] which allows to run a complete GNU/Linux system without installing any data on the hard disk. For such a configuration, the data can be saved on an usb key or floppy disk.
2. Use Cygwin[2] which is free but only partially open-source and provides all the classical UNIX tools under Windows, thus allowing to read and write on the Windows partitions.

2.2 Shell and simple file management

2.2.1 File names and paths

We will call **path** a list of directory names indicating a *location* where can be found either files or other directories. The convention is to separate directory names with a '`/`'. If a directory `a` contains a directory `b`, we will say `a` is the **parent directory** of `b`.

[1] `http://www.knoppix.org/`
[2] `http://www.cygwin.com/`

We will call **filename** the name of a file, and sometime we will make a confusion between the name of the file and the concatenation of the path and the name of the file.

A filename can contain almost any possible characters, including spaces. Anyway, the convention is to use only letters, digits, '`.`', '`_`' and '`-`'.

All the files are organized under Linux from a main directory called the **root directory**. For example my directory is called `/home/fleuret` : thus, this is a directory `fleuret`, which is into a directory called `home`, which is into the root directory.

The directory names '`.`' and '`..`' are reserved and means respectively the directory itself and the parent directory. Thus, the paths

```
/home/fleuret/sources
/home/fleuret/./sources
/home/fleuret/../fleuret/sources
```

are the same.

The files for the linux systems are organized into many directories. For example, the standard programs we will use are into `/usr/bin`, `/bin` and `/usr/local/bin`.

2.2.2 Shell

A **shell** is an interactive software that allows you to run other programs. Typically it appears as a simple text-window in which you can execute commands by typing their name and `enter`.

A shell has at any moment a **current directory**, which means that the path to this directory will be automatically added to all the file names you will specify. For security reasons, this path is not added by default to the command name itself. You can force it (for example to execute the result of a compilation) by using the '`./`' path.

2.2.3 Basic commands

You can run a command by typing its name + `enter` in a shell. The options between `[ ]` are optional.

`ls` $[-l]$ $[-t]$ $[< dirname >]$

Displays the list of files present in the current or specified directory. Option `-l`

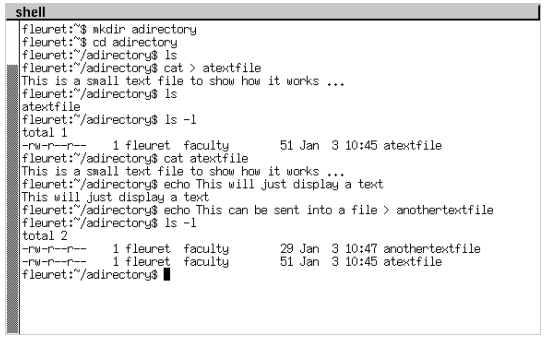


Figure 2.2: A classical shell session in the XTerm application under X-Window.

selects a long display, and `-t` sorts the files by modification date.

`mv` $<$ *initialname* $>$ $<$ *newname* $>$

Renames a file, and/or move it. If the *initialname* is a list of names separated by spaces, the *newname* has to be a directory and all the files will be move into it.

`rm` $<$ *filename* $>$

Removes a file names *filename*

`mkdir` $<$ *dirname* $>$

Creates a directory named *dirname*

`rmdir` $<$ *dirname* $>$

Removes a directory named *dirname*

`cd` [$<$ *dirname* $>$]

Selects *dirname* as the current directory

`pwd` [$<$ *dirname* $>$]

Displays the path to the current directory

`man` $<$ *commandname* $>$]

Shows the manual page for a given command

`less` [$<$ *filename* $>$]

Displays a file. Type the `space` key to go one page forward, the `b` key to go one page backward, `q` to quit.

`emacs` [$<$ *filename* $>$]

Edits a text file : `^x^s` saves the current file (the convention is to denote `^` the use of the `Ctrl` key, thus this shortcut means press `Ctrl` and `x` simultaneously, then release them and press `Ctrl` and `s` simultaneously), `^x^c` quits emacs, `^x^f` load a new file, `^x^w'` save the file under another name and finally `^_` undo the last command.

`g++` [$-O3$] [$-o <$ *objname* $>$] $<$ *sourcename* $>$

Compiles a file. Option `-O3` tells the compiler to optimize the result as much as it can. Thus, the compilation takes a bit more time, but the resulting program is faster. If the `-o` option is not used, the result file has the default name `a.out`.

`time` $<$ *command* $>$

Measures the time required to execute a given command

2.2.4 References for documentation

For more details, you can check the Linux Tutorial at :

`http`//www.linuxhq.com/guides/GS/gs.html:
`http`//www.cc.gatech.edu/linux/LDP/LDP/gs/gs.html:

2.3 First steps in C++

2.3.1 Data types

We can manipulate in C++ programs two different categories of types :

- **built-in types** which are defined in the C++ compiler itself ;
- **class type** which are defined in C++ source code from the built-in types.

We will focuse on four different built-in types :

- `bool` is a boolean value (true / false) ;
- `int` is an integer value (-2147483648 to 2147483647) ;
- `double` is a floating-point value (precision goes from $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{308}$) ;
- `char` is a character (letter, digit, punctuation, etc.)

Beside the four built-in types presented above, other ones exist. The main idea behind this large number of different types is to allow the programmer to control precisely the efficiency in term of memory usage and speed of its programs considering his needs.

For instance, the `unsigned int` encode an integer value between 0 and 4294967295. Both `int` and `unsigned int` use four bytes of memory on a x86 CPU. If we need to store a large number of smaller integer values, we can use the `short` type (or `unsigned short`), which takes only two bytes.

For the floating point values, the `float` type is less precise but takes less memory. Also, computation goes faster with this type.

2.3.2 A simple example of variable manipulation

```
int main(int argc, char **argv) {
  int i, j;
  i = 4;
  j = 12 * i + 5;
  exit(0);
}
```

`int main(int argc, char **argv)` is by convention the declaration of the part of the program which is run when the program starts, we will come back to this syntax later ;

`int i, j;` declares two **variables** of type int, called respectively `i` and `j`. It reserves two areas in the memory, and name them so that we can refere to them later in the program. The name of variables are called their **identifiers** ;

`i = 4;` copies the value 4 in the area of the memory called `i` ;

`j = 12 * i + 5;` reads the value in the area called `i`, multiplies it by 12, adds 5, and copies the result to the area of memory called `j` ;

`exit(0);` terminates the program and indicates to the shell from where the program is run that there was no error.

We have here made arithmetic operations between variables (`i` and `j`) and **literal constants** (`12` and `5`). Variable types are defined in their declarations, constant types are defined by the syntax itself. Basically, an `int` constant is a number with no dot, a `double` constant is a number with a dot, a `bool` constant is either `true` or `false`, and a `char` constant is a character between ‘‘ (for example “`char c = 'z';`”).

For floating point constant, we can use the `e` notation for powers of ten. For example `x = 1.34e-3` makes 0.00134 in `x`.

2.3.3 Variable naming conventions

A variable identifier can be composed of letters, digits and underscore character '_'. It must begin with either a letter or an underscore.

Usually, one concatenate words to build long identifier either using the underscore character '-' as a space, or by using upper caps for first letter of each word (except the first letter) :

```
int numberOfCars;
double first_derivative;
```

We will reserve identifiers starting with an upper caps for our class names.

2.3.4 Streams, include files

Beside the built-in types, the C++ compiler is often accompanied with lot of files containing predefined types. The main one we will use in our example programs is the `stream` type.

To use it, we need to indicate to the compiler to **include** in our source file another file called `iostream` (where this class type is defined).

```
#include <iostream>

int main(int argc, char **argv) {
  int k;
  k = 14;
```

```
  k = k + 4;
  k = k * 2;
  cout << k << '\n';
}
```

The variable `cout` is defined in the included file and is of type `ostream`. The only thing we need to know for now is that we can display a variable of a built-in type by using the `<<` operator.

We can also read values with `cin` and the `>>` operator. The following program gets two float values from the user and prints the ratio.

```
#include <iostream>

int main(int argc, char **argv) {
  double x, y;
  cin >> x >> y;
  cout << x / y << '\n';
}
```

Note that with recent versions of the GNU C++ compiler, you have to add `using namespace std;` after the `#include`.

2.3.5 The sizeof operator

We can know the memory usage of a given type using the `sizeof` operator. It takes either a variable name or a type name as parameter.

```
#include <iostream>

int main(int argc, char **argv) {
  int n;
  cout << sizeof(n) << ' ' << sizeof(double) << '\n';
}
```

The result is `4 8`.

2.3.6 The `if` statement

The `if` statement executes a part of a program only if a given condition is true.


```
if(condition)
 <statement to execute if the condition is true>
```

or

```
if(condition)
 <statement to execute if the condition is true>
else
 <statement to execute if the condition is false>
```

A **statement** here is either a part of a program enclosed in `{ }`, or a single line terminated with a ';'. For example :

```
#include <iostream>

int main(int argc, char **argv) {
  int n;
  cin >> n;
  if(n < 0) n = 0;
  else {
    n = 2 * n;
    n = n - 1;
  }
  cout << n << '\n';
}
```

2.3.7 The `for` statement

The `for` statement repeats the execution of a part of a program.

```
for(initialisation; condition; increment)
  <statement to repeat>
```

For example, to display all positive integers stricly smaller than a value given by the user :

```
#include <iostream>

int main(int argc, char **argv) {
  int n, k;
```

```
  cin >> n;
  for(k = 0; k < n; k++) cout << k << '\n';
}
```

Note that we have declared two variables of type `int` on the same line. The `k++` notation means in that context simply `k = k+1`.

The `for` can be used to make more complex loops :

```
#include <iostream>

int main(int argc, char **argv) {
  double x;
  for(x = 1; fabs(cos(x) - x) > 1e-6; x = cos(x));
  cout << x << ' ' << cos(x) << '\n';
}
```

2.3.8 The `while` statement

The `while` statement repeats the execution of a statement as long as a condition is true. For instance :

```
#include <iostream>

int main(int argc, char **argv) {
  double a, b, c;
  a = 0.0; b = 2.0;
  while(b-a > 1e-9) {
    c = (a+b)/2.0;
    if(c*c - 2.0 > 0.0) b = c; else a = c;
  }
  cout << c << '\n';
}
```

2.3.9 The `do { } while` statement

Similar to `while`, but the statement is always executed at least once.

```
#include <iostream>

int main(int argc, char **argv) {
  double a, b, c;
  a = 0.0; b = 2.0;
  do {
    c = (a+b)/2.0;
    if(c*c - 2.0 > 0.0) b = c; else a = c;
  } while(fabs(c*c - 2.0) > 1e-4);
  cout << c << '\n';
}
```


2.3.10 The `continue` statement

The `continue` statement forces the current execution of the loop to stop. It is equivalent to jump to the end of the statement, so that the next iteration can start :

```
#include <iostream>

int main(int argc, char **argv) {
  for(int n = 0; n<6; n++) {
    cout << "n = " << n << '\n';
    if(n%2 == 1) continue;
    cout << "This is even\n";
  }
}
```

Displays

```
n = 0
This is even
n = 1
n = 2
This is even
n = 3
n = 4
This is even
n = 5
```

2.3.11 The `switch` / `case` statements

When the behavior of the program can be organized as a succession of separate cases, selected by an integer value, the `switch` statement is more efficient and

elegant than a succession of `if` :

```
#include <iostream>

int main(int argc, char **argv) {
  int k;
  cout << "Enter a value between 1 and 3 : ";
  cin >> k;
  switch(k) {
    case 1:
      cout << "one!\n";
      break;
    case 2:
      cout << "two!\n";
      break;
    case 3:
      cout << "three!\n";
      break;
    default:
      cout << "Didn't get it, did you ?\n";
      break;
  }
}
```

2.3.12 Computation errors with floating point counters

Keep in mind that due to approximations in the computations, using a floating point counter is most of the time not safe. For example :

```
#include <iostream>

int main(int argc, char **argv) {
  double x;
  for(x = 0; x < 1.0; x = x + 0.1) cout << 1.0 - x << ' ';
  cout << '\n';
}
```

displays

```
1 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 1.11022e-16
```


2.4 An example of extreme C syntax

Avoid this kind of syntax in your homeworks.

```
#include <iostream>

int main(int argc, char **argv) {
  char text[] = "Hello!";
  char buffer[256];
  char *t, *s;

  s = text; t = buffer;
  while(*t++ = *s++);   // This is too much

  cout << text << ' ' << buffer << '\n';
}
```

Chapter 3

Expressions, variable scopes, functions

3.1 Expressions

An **expression** is a sequence of one or more **operands**, and zero or more **operators**, that when combined, produce a value. For example :

```
x - 3
```

```
cos(y) + y
```

```
x + y + z
```

```
x <= y * 7 - 2
```

3.2 Arithmetic operators

3.2.1 List of operators

The standard mathematic symbols can be used to write arithmetic expressions :

Symbol	Function
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	reminder

The `%` computes the reminder of an integer division (for instance `17 % 5` has the value `2`) ; the result is well defined only if both operands are positive.

All those operators but `%` can be used with either integer or floating point operands.

3.2.2 Operators depend on the types of the operands

Internally, each symbol corresponds to different operations, depending with the type of the operands :

```
#include <iostream>

int main(int argc, char **argv) {
  cout << 15 / 4 << ' ' << 15.0 / 4.0 << '\n';
}
```

displays `3 3.75`.

3.2.3 Implicit conversions

Basically operators are defined for two operands of the same type. The compiler can automatically convert a numerical type into another one so that the operator exists :

```
#include <iostream>

int main(int argc, char **argv) {
  cout << 3 + 4.3 << '\n';
}
```

result is `7.3`

The implicit conversion can not be done to a type that loses information (i.e. `double` to `int` for instance). For example the `%` operators is only defined for integer operands :

```
#include <iostream>

int main(int argc, char **argv) {
  cout << 3.0%4.0 << '\n';
}
```

the compilation generates :

```
/tmp/something.cc: In function ‘int main(int, char **)’:
/tmp/something.cc:4: invalid operands ‘double’ and ‘double’
                     to binary ‘operator %’
```

3.2.4 Arithmetic exceptions

Arithmetic computations can lead to **arithmetic exceptions**, either because the computation can not be done mathematically, or because the used type can not carry the resulting value. In that case the result is either a wrong value or a non-numerical value :

```
#include <iostream>
#include <cmath>

int main(int argc, char **argv) {
  int i, j;
  j = 1;
  for(i = 0; i<20; i++) j = j*3;
  cout << j << '\n';

  double y;
  y = 0.0;
  cout << 1.0 / y << ' ' << (-1.0) / y << '\n';

  cout << log(-1.0) << '\n';
}
```

displays


```
-808182895
inf -inf
nan
```

Note that those exceptions *do not stop* the execution of the program because `nan`, `inf` and `-inf` are legal values for floating point variables. Integer null division does :

```
int main(int argc, char **argv) {
  int i, j;
  i = 0;
  j = 3 / i;
}
```

compiles with no errors but the execution produces

```
Floating point exception
```

3.2.5 Boolean operators

We can define more precisely what we called a *condition* in the description of the `if`, `for`, and `while` syntax. It is a **boolean expression**, which is an expression whose value is of type `bool`.

A few operators have a boolean value and takes boolean operands :

Symbol	Function
`&&`	conjunction (AND)
`||`	disjunction (OR)
`!`	logical NOT

```
#include <iostream>

int main(int argc, char **argv) {
  bool c1, c2;
  c1 = true;
  c2 = !c1 && false;
  cout << c1 << ' ' << c2 << '\n';
}
```

The compiler is smart and will compute the value of the second operand of a boolean operation only if this is necessary.

3.2.6 Comparison operators

The comparison operators take two numerical operands and have a boolean value :

Symbol	Function
`<=`	less or equal $\leq$
`<`	less $<$
`>=`	greater or equal $\geq$
`>`	greater $>$

The equality and inequality are defined for any types and return a boolean value :

Symbol	Function
`==`	equal $=$
`!=`	different $\neq$

3.2.7 Assignment operator

A strange thing in C++ is that assignments are also expressions :

```
j = 3 + (i = 5);
```

is legal, and will assign to `i` the value 5, and to `j` the value 8. But feel free **not to use** such weird tricks.

3.2.8 Increment and decrement operators

The `++` operator, as we have seen in the `for` loops, can increment a variable. But, like the assignment operator, it is also an expression. The delicate point is that you can either use it as **post-increment** or **pre-increment**.

When placed on the left (resp. on the right) of the variable to increment, the value of the expression will be the value of the variable *after* the increment (resp. before).

For instance :

```
#include <iostream>
```



```
int main(int argc, char **argv) {
  int i, j, k;
  i = 4; j = ++i;
  i = 4; k = i++;
  cout << j << ' ' << k << '\n';
}
```

Displays `5 4`.

The `--` operator does the same for decrement.

3.2.9 Precedence of operators

The precedence of operators is the order used to evaluate them during the evaluation of the complete expression. To be compliant with the usual mathematical notations, the evaluation is not left-to-right. For example

```
3 + 4 * 5 + 6 * 7
```

is considered by the compiler as

```
3 + (4 * 5) + (6 * 7)
```

and **NOT AS**

```
(((3 + 4) * 5) + 6) * 7
```

When two operators have same precedence (i.e. when we have the same operator twice), the evaluation is left-to-right.

The specification of C++ do **not** specify the order of evaluation when operators have the same precedence, except for logical operations (see above §3.2.5). For example

```
i = 0;
cout << i++ << ' ' << i++ << '\n';
```

prints `1 0`.

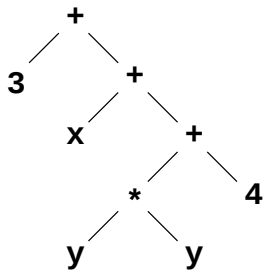


Figure 3.1: Graph for the expression `3 + x + (y * y + 4)`

3.2.10 Grammar, parsing and graph of an expression

The usual way to define the syntax of a language is to use generative grammar. Typically it consists in recursive definition of the syntax. For instance, we could define an arithmetic expression $< expr >$ as, either :

- A litteral constant (`4`, `-34.567`, `1.234e485`, etc) ;
- a variable (`x`, `AgeOfMyCat`, `number_of_cars`, etc.) ;
- $(< expr >)$;
- $< expr > + < expr >$;
- $< expr > * < expr >$;
- $< expr > - < expr >$;
- $< expr > / < expr >$.

From such a definition, the compiler is able to build a tree to encode the expression. The leaves of this tree are either variables or literal constants and internal nodes are operators. Each subtree of this tree is an expression itself.

3.2.11 Summary

1. The operation (what the computer does) associated to an operator (the symbol) depends on the type of the operands (the things combined into the operations) ;

2. the compiler can do implicit conversions (only if no precision is lost) so that the expression has a meaning. ;
3. some arithmetic operations produce an arithmetic exceptions, leading either to a wrong answer or to non-numeric values ;
4. assignment and increment operators are expressions ;
5. operators have precedence consistent with the mathematical conventions ;
6. an expression is represented by the compiler as a graph, this is the good way to see it.

3.3 lvalue vs. rvalue

In many situation we have to make a difference between expressions defining a value that can be addressed (and changed), which are called **lvalue** and value that can be only read, called **rvalue**. For example, the assignment operator expect a lvalue on the left and a rvalue on the right.

So far the only lvalue we have seen are variables.

```
#include <iostream>

int main(int argc, char **argv) {
  int i;
  i+3 = 5; // does not compile
  45 = i; // does not compile
}
```

leads to the following compilation errors :

```
/tmp/something.cc: In function ‘int main()’:
/tmp/something.cc:4: non-lvalue in assignment
/tmp/something.cc:5: non-lvalue in assignment
```

3.4 Scopes of variables

We can define variables almost everywhere in the program. Of course, when a program is several thousands line, we have to be able to use in different places the same identifiers.

Thus, each identifiers can be used only in a partial part of the program, we call it a **scope**.

Roughly, a identifier can be used everywhere from its declaration point to the end of the block defined by a couple of `{ }`.

```
#include <iostream>

int main(int argc, char **argv) {
  int i;
  i = 3;
  if(i == 3) {
    int j;
    j = i + 4;
  }
  j = i + 3;
}
```

leads to :

```
/tmp/something.cc: In function ‘int main(int, char **)’:
/tmp/something.cc:10: ‘j’ undeclared (first use this function)
/tmp/something.cc:10: (Each undeclared identifier is reported
                       only once
/tmp/something.cc:10: for each function it appears in.)
```

Variables can also be declared in the `for` statement. In that case the scope of the identifier is the loop :

```
int main(int argc, char **argv) {
  int j;
  j = 0;
  for(int i = 0; i < 10; i++) j = j + i;
}
```

3.5 Functions

3.5.1 Defining functions

To re-use the same part of a program, we can define a **function**, which takes parameters, and returns a result of various type. Typical definition contains the

type of the value it returns, an identifier for its name, and the list of parameters with their types. The evaluation of a function is done when the **call operator** `()` is used. One **argument** (i.e. an expression) is provided to each parameter. An example makes things clearer.

```
#include <iostream>

// This function has one parameter called x
double square(double x) { return x*x; }

// This one has two parameters called x and y
// It returns the largest k so that y^k <= x
int maxexpon(double x, double y) {
  double z = 1;
  int result = 0;
  while(z <= x) { result++; z = z * y; }
  return result-1;
}

int main(int argc, char **argv) {
  double a, b;
  cin >> a >> b;
  // The argument is a for the first call and a+b for the second
  cout << square(a) + square(a + b) << '\n';
  // The two arguments are and b
  cout << maxexpon(a, b) << '\n';
}
```

Note that as for loops, the scope of the parameters and variables defined in the function definition is the function statement itself.

3.5.2 `void` return type

If a function is supposed to return no value, you can declare the return type as `void`.

```
#include <iostream>

void printSmallerSquares(int x) {
  int y;
  for(y = 0; y * y <= x; y++) cout << y * y << ' ';
  cout << '\n';
```

```
}

int main(int argc, char **argv) {
  printSmallerSquares(17);
}
```

displays

```
0 1 4 9 16
```

3.5.3 Argument passing by value

By default, functions pass arguments **by value**, which means that when the function is used, the rvalues of **arguments** are copied into the **parameters**.

The main effect is that even if the argument is a lvalue, modifying the corresponding parameter will not change the argument's value. For example :

```
#include <iostream>

void stupidfunction(int x) {
  x = 4;
}

int main(int argc, char **argv) {
  int y;
  y = 12;
  stupidfunction(y);
  cout << y << '\n';
}
```

This prints 12.

Here we have a parameters x in the function definition, and an argument y when the function is called. Modifying x in the function does **not** change the value of y in the main part of the program.

3.5.4 Argument passing by reference

In some certain situations, it is more efficient or convenient to be able to modify the argument(s). To do that, the & symbol specifies that the parameter and the

argument correspond to the same rvalue. This is called passing an argument by reference.

```
#include <iostream>

// This is the include file containing the math functions
#include <cmath>

int normalize(double &x, double &y) {
  double d;
  d = sqrt(x*x + y*y);
  x = x/d;
  y = y/d;
}

int main(int argc, char **argv) {
  double a, b;
  a = 17.3; b = -823.21;
  cout << sqrt(a*a + b*b) << '\n';
  normalize(a, b);
  cout << sqrt(a*a + b*b) << '\n';
}
```

Displays :

```
823.392
1
```

3.5.5 Recursive function call

A function can call itself in its definition statement. We call such a scheme a **recursive function**. In fact this is possible because at each call, new variables are allocated in the memory.

Example :

```
#include <iostream>

int fact(int k) {
  cout << k << '\n';
  if(k == 0) return 1;
  else return k*fact(k-1);
```

```
}

int main(int argc, char **argv) {
  int n = fact(4);
  cout << '\n' << n << "\n";
}
```

```
4
3
2
1
0

24
```

3.5.6 Stopping condition

The only (and small) difficulty is the necessary existence of a **stopping condition**. This ensure that at one point the function will not call itself anymore, whatever the initial value of the parameters was :

```
// Oooops ... will not work
int factorial(int k) {
  return k*factorial(k-1);
}
```

3.6 The `abort()` function

The `abort()` function is wich interrupt the execution of your program as if there was a serious error. Use it to handle non-expected behavior like out-of bounds exceptions :

```
#include <iostream>

int main(int argc, char **argv) {
  int x;
  cout << "Enter a non-null value : ";
  cin >> x;
  if(x == 0) {
    cerr << "Null value!\n";
```



```
    abort();
  }
  cout << 1/x << '\n';
}
```

The execution is the following :

```
Enter a non-null value : 0
Null value!
Aborted
```

Chapter 4

Arrays and pointers, dynamic allocation

4.1 Arrays and pointers

4.1.1 Character strings

So far, we have only printed isolated characters. C++ provides a syntax to define a string of characters :

```
cout << "What a beautiful weather!!!\n";
```

Precisely such a character string is a succession of characters stored in memory, followed by a null character (this is a convenction in C/C++). This constant is finally of type **array of char**, denoted `char[]`. The compiler refers to it internally with the address of its first character and keeps tracks of its size.

4.1.2 Built-in arrays

The "" operator defines arrays of char. Similarly, we can define an array of any type with the `[ ]` operator :

```
int n[4] = { 1, 2, 3, 4 };
```


The compiler is able to determine by itself the size of an array, so you do not have to specify it :

```
int poweroftwo[] = { 1, 2, 4, 8, 16, 32, 64, 128 };
```

As we said, the compiler keeps the information about the size of arrays (or strings), so the `sizeof` operator returns the size of the array as expected :

```
#include <iostream>

int main(int argc, char **argv) {
  int hello[] = { 1, 2, 3 };
  char something[] = "abcdef";
  cout << sizeof(hello) << ' ' << sizeof(something) << '\n';
}
```

The size of arrays is always known and has to be known at compilation time.

Note : from that, you can compute the number of element of an array by dividing the `sizeof` the array by the `sizeof` the element type.

4.1.3 Index of arrays, the `[ ]` operator, out of bounds exception

The `[ ]` operator allows to access (as a lvalue) to a given element of an array. The first element has for index **0 (and not 1!)**.

```
#include <iostream>

int main(int argc, char **argv) {
  int x[] = { 3, 1, 4, 1, 5, 9 };
  for(int i = 0; i < 6; i++) cout << x[i] << "\n";
}
```

If you try to access an element out of the array (negative index or greater than the size of the array -1), the program may or may not crash. **The behavior is not well defined, this is the source of the majority of bugs**.

```
#include <iostream>
```

```
int main(int argc, char **argv) {
  int x[3];
  for(int i = 0; i<1000; i++) {
    x[i] = 0;
    cout << "Erasing x[" << i << "]\n";
  }
}
```

The result is a lot of lines, the two last ones being :

```
Erasing x[326]
Segmentation fault
```

This means that even if the array was only of size 3, the program did not crash until we finally start to write part of the memory we were not allowed to.

4.1.4 Pointers, the *, and & operators

A **pointer** is a variable containing a **reference** to a variable of a given type instead of a value of a given type.

The * operator allows to declare pointers to variables. The & operator, also called the **address-of operator**, allows you to get the address of a given variable.

```
#include <iostream>

int main(int argc, char **argv) {
  int i = 15;
  char *s = "What a beautiful weather!!!";
  int *p = &i;
}
```

4.1.5 Pointers to pointers to pointers to ...

The * operator allows you to declare also pointers to pointers :

```
int main(int argc, char **argv) {
  int i = 15;
```

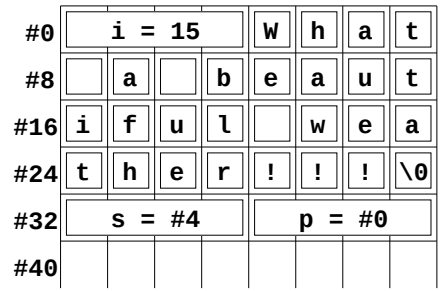



Figure 4.1: This figure of the memory does not give realistic values for the locations of variables. The specifications of C++ do not give any informations about the locations of the declared variables.

```
  int *j = &i;
  int **something = &j;
  int ***ptrToSomething = &something;
}
```

4.1.6 Dereference operator *

The * operator also allows you to access to pointed variables (as a lvalue) ; in that last case it is called the **dereference operator**.

```
#include <iostream>

int main(int argc, char **argv) {
  int i;
  int *p = &i;

  cout << "p = " << p << "\n";
  i = 4;
  cout << "i = " << i << "\n";
  *p = 10;
  cout << "i = " << i << "\n";
}
```

Gives the following :

```
p = 0xbffffb04
i = 4
i = 10
```

4.1.7 Pointers to arrays

An array type can be implicitly transformed into a pointer by the compiler, and the `[ ]` operator can be used to access to an element of a pointed array.

For example, we can define a function to sum the terms of an array of `int` :

```
#include <iostream>

int sum(int *x, int sz) {
  int s = 0;
  for(int k = 0; k < sz; k++) s = s + x[k];
  return s;
}

int main(int argc, char **argv) {
  int cool[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  cout << sum(cool, 10) << "\n";
}
```

Displays `55`.

4.1.8 Pointers do not give information about pointed array sizes

Note that while the compiler has informations about the size of an array type, it does not have such information for a pointer. The pointer is *just* the reference to a given location in the memory. Nothing more.

The compiler has no way to know that a '`int *`' points to an array of 10 values and not to a single integer.

If you try to apply the `sizeof` operator to a dereferenced pointer, you will obtain as a result the size of the pointed type (**one** element alone) :

```
#include <iostream>
```



```
int main(int argc, char **argv) {
  int cool[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int *coolptr = cool;
  cout << sizeof(cool) << ' ' << sizeof(*coolptr) << '\n';
}
```

displays

```
40 4
```

4.1.9 Box and arrows figures

A very convenient way to represent the configuration of the memory at a given time is to use boxes to represent variables and arrows to represent references. Those figures forget the absolute locations of variable in memory (which are not well-defined) and emphasis on their values and the referencing relations.

Each box corresponds to a variable and contains, when defined, the identifier (the variable name), followed by, if defined, the current value. In the case of initialized pointers (ones actually pointing to some allocated variable), the value is absent and replaced by an arrow starting in the rectangle and pointing to the allocated variable.

For example, the memory state after the following piece of code

```
int x = 13;
char *s = "Linux rocks!";
int *y = &x;
int k[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }};
```

is represented on Figure 4.2.

4.1.10 The `main` function's parameters

From what we have seen about arrays and pointers, we can now interpret the meaning of the `main` function declaration. The first parameter, of type `int` is the number of arguments passed to the program when run from a shell (**including the name of the program itself**) and the second parameter is a pointer to an array of pointers to strings containing the arguments themselves :

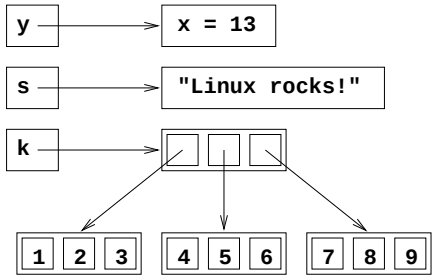


Figure 4.2: Memory configuration after the piece of code given page 42

```
#include <iostream>

int main(int argc, char **argv) {
  for(int i = 0; i<argc; i++)
    cout << "argument #" << i << " is '" << argv[i] << "'\n";
}
```

We can run this program with arguments separated with spaces :

```
> ./test this is just a bunch of arguments
argument #0 is './test'
argument #1 is 'this'
argument #2 is 'is'
argument #3 is 'just'
argument #4 is 'a'
argument #5 is 'bunch'
argument #6 is 'of'
argument #7 is 'arguments'
```

4.1.11 Adding integers to pointers

The `+` can have for operands a pointer and an integer value. In such a situation, the compiler will implicitely multiply the integer operand by the size of the pointed type. Thus if `p` is a pointer and `k` an integer, `p[k]` and `*(p+k)` are the same.

```
#include <iostream>
```



```
int main(int argc, char **argv) {
  int n[] = {0, 1, 2, 3, 4, 5};
  int *p = n;
  int *p2 = p+3;
  cout << *p << " " << *p2 << "\n";
}
```

displays :

```
0 3
```

This kind of operation is not very secure and should be handled with care.

4.2 Dynamic allocation

4.2.1 Why ? How ?

In many situations the programmer does not know when he writes a program what objects he will need. It can be that he does not know if he will need a given object, or or that he does not know the size of a required array.

The `new` operator allows to create an object at run-time. This operator takes as an operand a type, which may be followed either by a number of elements between `[ ]` or by an initial value between `( )` :

```
char *c = new char;
int *n = new int(123);
double *x = new double[250];
```

The area of memory allocated by `new` is still used out of the pointer's scope!

Bugs due to missing object deallocation are called *memory leaks*.

To free the memory, the programmer has to explicitly indicate to the computer to do so. The `delete` operator (resp. `delete[]`) takes a pointer to a single object (resp. an array of objects) as an operand and free the corresponding area of the memory ;

```
delete n;
delete[] x;
```

The variables declared in the source code, without the usage of `new` and `delete` are called **static variables**. Their existence is completely handled by the compiler who implicitly adds *invisible* `new`s and `delete`s.

The operand pointer can be equal to 0 ; in that case `delete` (or `delete[]`) does nothing. But deallocating an area which has already been deallocated has a non-defined behavior (i.e. crashes most of the time).

4.2.2 Dynamic arrays

The typical usage of dynamically allocated arrays is the following :

```
#include <iostream>

int main(int argc, char **argv) {
  int n;

  cout << "What array size ? ";
  cin >> n;

  int *x = new int[n];
  for(int k = 0; k < n; k++) x[k] = k*k;
  delete[] x;
}
```

If the memory can not be allocated (not enough memory basically), the program crashes :

```
#include <iostream>

int main(int argc, char **argv) {
  for(int k = 0; k < 10000; k++) {
    int *x = new int[100000];
    cout << "k = " << k << " (x = " << x << ")\n";
  }
}
```

displays lot of lines, the two last being :

```
k = 3370 (x = 0x3ff4acc8)
Aborted
```


4.2.3 Test of a null pointer

A pointer can be implicitly converted into a `bool`. All non-null pointers are equivalent to `true` and the null one is `false`. The convention is that the null pointer correspond to a non-existing object.

Static pointers are not initialized to 0 **when the program starts, so you have to be very careful when using this convention**

For example, if we want to write a function that count the number of characters into a character strings, and returns 0 if the parameter pointer is null :

```
#include <iostream>

int length(char *s) {
  if(s) {
    char *t = s;
    while(*t != '\0') t++;
    // The difference of two pointers is an integer
    return t-s;
  } else return 0; // the pointer was null
}

int main(int argc, char **argv) {
  char *s = 0;
  cout << length(s) << '\n';
  s = "It's not personal, it's business";
  cout << length(s) << '\n';
}
```

The `delete` and `delete[]` operators do not set the value of the deallocated pointer to zero.

4.2.4 A non-trivial example using dynamic memory allocation

We can write a function that takes as parameters a vector under the form of a dimension and an array of coefficients, and returns an array containing the components of the normalized vector :

```
#include <iostream>
#include <cmath>
```

```
double *normalize(double *a, int d) {
  // First we compute the norm of the vector
  double s = 0.0;
  for(int k = 0; k < d; k++) s += a[k]*a[k];
  s = sqrt(s);

  // Then we declare a result vector
  double *result = new double[d];

  // And we fill it with the normalized components
  for(int k = 0; k < d; k++) result[k] = a[k]/s;
  return result;
}
```

When we use this function we must keep in mind we have to **deallocate** the result vector.

```
int main(int argc, char **argv) {
  int dim;
  cin >> dim;

  // Enter the vector
  double *v = new double[dim];
  for(int k = 0; k<dim; k++) cin >> v[k];

  // Computes the normalized version
  double *w = normalize(v, dim);

  // Prints it
  for(int k = 0; k < dim; k++) cout << w[k] << ' ';
  cout << '\n';

  // Free everything (do NOT forget the vector returned by the
  // function)
  delete[] v;
  delete[] w;
}
```

4.2.5 Dynamically allocated bi-dimensional arrays

As we have seen, a bi-dimensional array is an array of pointers to arrays. Allocating dynamically such an object is a bit more tricky and requires a loop :


```
#include <iostream>

int main(int argc, char **argv) {
  double **m;
  int w, h;

  cin >> w >> h;

  // Allocation requires a loop
  m = new (double *)[w];
  for(int k = 0; k<w; k++) m[k] = new double[h];

  // Deallocation also
  for(int k = 0; k<w; k++) delete[] m[k];
  delete[] m;
}
```

We will see later how C++ allows to create objects to deal with bi-dimensional arrays in a more elegant and efficient way.

4.2.6 What is going on inside: the stack and the heap

Internally, the computer allocates variables in two ways. Static variables, because it is known by advance where they are going to be deallocated can be are allocated in a ordered way. This strategy is called a **stack** because the last one allocated is the first one deallocated (like a stack of plates: the last one put in the stack will be the first one taken). The same stack is used for many other purposes and suffers from one main limitation: the maximum size of an array be allocated that way is small. Allocating too large arrays or too many static variables (for instance because of too many recusrive calls) leads to extremely strange behavior and crashes.

```
#include <iostream>

int main(int argc, char **argv) {
  int s = 10000000;
  double values[s];
  cout << "Hello!\n";
}
```

```
(gdb) run
Starting program: /home/fleuret/sources/a.out
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0804858d in main (argc=1, argv=0xbffff844) at ./bug.cc:6
6         cout << "Hello!\n";
```

However, the main strength of that stack strategy is its speed. Since variables are allocated next to another in memory, there is no need to find a free area: it is always instantaneously known.

The dynamic variables managed with `new` and `delete[]` are allocated in a **heap**, which means that there is no structure in the way they are organized in memory. Such allocations can be very large. The main drawback is the allocation time, which requires an (implicit in C++, explicit in C) system call.

4.3 Miscellaneous

4.3.1 Declaration vs. definition

The **declaration** of a function specifies the return and parameter types. The **definition** specifies the part of the program associated to the function, i.e. the statement between `{ }`.

Up to now, we have always done the declaration and the definition simultaneously, but declaration can be done before the definition. This is useful because a function can be called at a given location in the program only if the declaration we done before.

```
#include <iostream>

int function1(int x);

int function2(int x) {
  if(x == 0) return 0;
  else return function1(x-1);
}

int function1(int x) {
  if(x == 0) return 1;
  else return function2(x/2);
}

int main(int argc, char **argv) {
```



```
  cout << function1(15) << '\n';
}
```

4.3.2 The `const` statements

To define constant values, using a variable can be dangerous. We could, by mistake modify it, corrupting the global behavior of the software.

To control that what we want to be constant is actually constant, the compiler can check that no *writing* access is made to a given variable.

To specify such a protection, just declare a variable as `const` :

```
int main(int argc, char **argv) {
  const int x = 15;
  // Ooooopsss, we try erase the value by accident
  bool cond = (x = 13);
}
```

The compilation returns :

```
/tmp/something.cc: In function `int main(int, char **)':
/tmp/something.cc:4: assignment of read-only variable `x'
```

The compiler is also able to prevent you from fooling it by using pointers :

```
#include <iostream>

int main(int argc, char **argv) {
  const int x = 15;
  cout << "x = " << x << "\n";
  int *p = &x;
  *p = 98;
  // let's try to avoid the const qualifier
  cout << "x = " << x << "\n";
}
```

the compilation returns :

```
/tmp/something.cc: In function `int main(int, char **)':
/tmp/something.cc:6: initialization to `int *' from
                     `const int *' discards qualifiers
```

4.3.3 The enum type

In many case, we need to define a type that takes a finite set of values. Instead of defining the symbols with a succession of `const` declaration, we can use the `enum` keyword :

```
enum { FACULTY, STUDENT, VISITOR } status;
```

Such a variable can be implicitly converted to `int` (use with care, this is not a very natural operation) :

```
#include <iostream>

int main(int argc, char **argv) {
  enum { FACULTY, STUDENT, VISITOR } status;
  status = STUDENT;
  cout << status + 14 << '\n';
}
```

displays

```
15
```

4.3.4 The break statement

The C++ language allows to bypass the natural ending of statements by using the `break`. It terminates the current `for`, `while` or `switch` statement (roughly, jump to the part of the code after the next closing `}`) :

```
#include <iostream>

int main(int argc, char **argv) {
  int k;
  cin >> k;
  for(int n = 0; n<100; n++) {
    cout << "n = " << n << '\n';
    if(n == k) break;
    cout << "We go on\n";
  }
}
```


if we enter the value `3`, we obtain :

```
n = 0
We go on
n = 1
We go on
n = 2
We go on
n = 3
```

4.3.5 Bitwise operators

Various operators allow to apply boolean operations on bits individually :

```
#include <iostream>

int main(int argc, char **argv) {
  cout << "128 | 15 = " << (128 | 15) << '\n';
  cout << "254 & 15 = " << (254 & 15) << '\n';
  cout << "~15 = " << (~15) << '\n';
}
```

displays

```
128 | 15 = 143
254 & 15 = 14
~15 = -16
```

4.3.6 The comma operator

A succession of expressions separated by commas are evaluated from left to right, the result of the global expression being the value of the last one to be evaluated :

```
#include <iostream>

int main(int argc, char **argv) {
  int i, j, k;
  i = 0; j = 0; k = 0;
  cout << (i++, j = j+14, k = k-3) << '\n';
```

```
  cout << i << ' ' << j << ' ' << k << '\n';
}
```

displays

```
-3
1 14 -3
```

Beware of the very low precedence of the comma operator.

Chapter 5

War with the bugs

"The only good bug is a dead bug"
– *Starship Troopers*

5.1 Preamble

For the sake of performance and compatibility with C, C++ provides very few mechanisms to avoid bugs. The programmer style is thus far more important than for other languages like Java, Caml or C#.

A compilation error is not called a **bug**. It is a syntaxic error, which is usually easy to find and fix. Except if you have an amazing style, the fact that a program compiles does not ensure you at all that it will work.

5.2 The Bug Zoo

5.2.1 The program crashes: `Segmentation fault`

This family of problem is extremely large and contains two main sort of errors: access to non-authorized part of the memory and system calls with incorrect parameter values.

Unauthorized memory access

It when you try to read or write to a memory address

1. totally meaningless (for instance a non-initialized pointer)
2. out of bounds
3. not allocated anymore

Note that a memory access with an unitialized pointer may corrupt the memory without actually crashing the program. For instance

```
#include <iostream>

int main(int argc, char **argv) {
  int b;
  int a[10];
  b = 4;
  for(int i = 0; i < 100; i++) {
    a[i] = 12;
    cout << b << " "; cout.flush();
  }
}
```

displays

```
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
Segmentation fault
```

First, the loop fills the `a` array, then it erases `b` (which is just after `a` in the memory) but the program still does not crash because the operating system has allocated a minimum amount of memory larger than what is specified in the source code. When the counter leaves the allocated area, the CPU tries to access a non-authorized part and the operating system kills it.

Such errors can be extremely tricky, since an incorrect memory access can crash the program after a while

```
#include <iostream>

int main(int argc, char **argv) {
  int *b;
  int a[10];
  b = new int[10];
  for(int i = 0; i < 20; i++) a[i] = 12;
  cout << "We are here!\n";
  b[2] = 13; // kaboom
}
```

prints out

```
We are here!
Segmentation fault
```

Here `b` was correctly initialized, then erased by an out-of-bound access of array `a`, and the crash occurs when then not-anymore correct value of `b` is used.

Incorrect system call

The second case occurs when you use a system call with wrong parameter values. It can be explicit (for instance the UNIX `fclose` with a non-initialized value) or implicit through the C++ memory allocation / deallocation system (for instance if you `delete[]` the same array twice)

```
#include <iostream>

int main(int argc, char **argv) {
  int *b = new int[123];
  delete[] b;
  delete[] b; // kaboom
}
```

5.2.2 The program crashes: `Floating point exception`

This happens when you try a division by 0 with integer numbers. Note that the floating-point types are extremely tolerant to meaningless operations. Since those types can carry values such as `nan`, `inf` and `-inf`, computating values such as logarithm of negative numbers, square root of negative numbers and inverse of 0 will not crash the program (but will most of the time lead to a wrong result).

5.2.3 The program never stops

Programs can remain stuck in a loop if either the end condition can not be reached, or because the state does not change.

```
for(int i = 1; i > 0; i++) cout << "i = " << i << "\n";
for(int j = 0; j < 100; j = j*2) cout << "j = " << j << "\n";
```

It may stop eventually in certain cases, but if the computation requires ten years to complete, it is very similar to being frozen from a user perspective.

5.2.4 The program uses more and more memory

A **memory leaks** occurs when memory is allocated several times for the same purpose and is not deallocated when the task is over. Those ones are tricky to find since the program does not crash quickly and can slow down the whole system by exhausting the ressources.

```
double x[10];
x[10] = 4.0; // Out of bound

double *y, *z;
*y = 4;      // unitialized pointer

y = new double[10];
z = y;
delete[] y;
delete[] z;  // Already deallocated

for(int i = 0; i < 1000; i++) {
  double *x = new double[1000];
  for(int j = 0; j < 1000; j++) x[j] = j;
  // A delete[] is missing!
}
```

5.2.5 The program does not do what it is supposed to do

They can be caused by plain mistakes

```
int x = 3, y = 4;
int product_of_both = x + y; // It seems that the computation
```

```
                                // is not what was wanted here
```

by non-initialized variables

```
int sum; // This one should be set to 0
for(int j = 1; j <= 100; j++) sum += j;
cout << "Sum of the first 100 integers = " << sum << "\n";
```

or by tricky floating point computation errors

```
#include <iostream>

int main(int argc, char **argv) {
  cout.precision(30);

  float a = 0;
  for(int i = 0; i < 10; i++) a += (1.0/10.0);
  cout << "a = " << a << "\n";

  float b = 0;
  for(int i = 0; i < 100; i++) b += (1.0/100.0);
  cout << "b = " << b << "\n";

  double c = 0;
  for(int i = 0; i < 10; i++) c += (1.0/10.0);
  cout << "c = " << c << "\n";
}
```

prints

```
a = 1.00000011920928955078125
b = 0.99999934434890747070312 5
c = 0.99999999999999988977697537484
```

Never expect two floating point computations supposed to be equal from a mathematical perspective to be actually equal.

5.3 How to avoid bugs

5.3.1 Write in parts

Finding one bug in the 20 lines you typed for the last fifteen minutes is easier than finding fifty bugs in the two thousands lines you have typed for one month.

5.3.2 Good identifiers

Use long identifiers such as `sum_of_the_weights` instead of short ones. Use longer identifiers for variables and functions used in many parts of the program. If a variable is used only in a 5 line loop, it can be called `s` with no risk. If you are really lazy, at least use acronyms (`ws` for weight sum for instance).

Also, reuse the same names and parameter order everywhere. Avoid at all cost this sort of mess

```
int rectangle_surface(int xmin, int ymin,
                      int xmax, int ymax) {
  return (xmax - xmin) * (ymax - ymin);
}

int rectangle_diagonal_length(int xmin, int xmax,
                              int ymin, int ymax) {
  return sqrt(double(   (xmax - xmin) * (xmax - xmin)
                      + (ymax - ymin) * (ymax - ymin) ) );
}
```

5.3.3 Use constants instead of numerical values

It is extremely dangerous to have a consistency between values which is not made explicit. For instance, the size of an array which appears both for the allocation and in a loop should always be specified by the mean of a constant with a name. That way, it can be changed without having to change it in many places, and it also reminds you the semantic of that value (i.e. it is a number of elements).

5.3.4 Comment your code

Comments help the one who is going to use the source code later. It can be somebody else, or it can be you in one month, or you in fifteen minutes. Depending upon your goal – are you going to work in team ? who are you going to work with ? are you planning to maintain this code ? will severe teachers read it ? – your comments have to be more or less precise.

Always put comments if a piece of code has a non-obvious behavior, for instance if there is a constraint on the parameters of a function, or if it returns values in a strange way.

```
// Angle in degrees, radius in meter, returns square meters
double piece_of_pie_surface(double angle, double radius) {
  return M_PI * radius * radius * angle / 180.0;
}
```

5.3.5 Symmetry and indentation

Arrange your source so that obvious missing or incorrect elements will be instantaneously spotted.

Which of the two sources below is easier to debug

```
int size; cin >> size;  double *a[size];
if(size > 0)
 {
for(int i = 0; i < size; i++) {
a[i] = new double[i];
  for(int j = 0; j < i; j++) a[i][j] = j + i;}
  delete a[i];
}
```

```
int size;
cin >> size;

double *a[size];

if(size > 0) {
  for(int i = 0; i < size; i++) {
    a[i] = new double[i];
    for(int j = 0; j < i; j++) a[i][j] = j + i;
  }
  delete a[i];
}
```


Note that in a given block of instructions, the number of `new` is equal to the number of `delete`, except in rare cases. The example above does not respect this rule.

5.3.6 Use a DEBUG flag

The C++ provides the concept of conditional compilation. We will not go into the details of it but we can use it in a simple way to increase the robustness of our code.

The idea is to write some parts of the code to check conditions and to actually compile them only if something goes wrong. That way, when we have tested the program with those conditions, we can remove them and run the program at full speed.

```
int rectangle_surface(int xmin, int ymin, int xmax, int ymax) {
#ifdef DEBUG
  if(xmin > xmax || ymin > ymax) {
    cerr << "Something bad happened.\n";
    abort();
  }
#endif
  return (xmax - xmin) * (ymax - ymin);
}
```

When the compilation is done with the `-DDEBUG` options passed to the compiler, the checking piece of code is actually compiled. Without that option, the part between the `#ifdef` and `#endif` is ignored by the compiler.

Note that you can also put a lot of tests which are always executed. The cost in term of performance is usually very small.

5.4 How to find bugs

5.4.1 Print information during execution

The best way to find errors is to print a lot of information about the internal state of the program. For instance, if a program remains frozen, the first thing to do is to print something when a few checkpoints are met

```
cout << "Checkpoint #1\n";
for(int i = 1; i < 1000; i++) cout << "i = " << i << "\n";
cout << "Checkpoint #2\n";
for(int j = 0; j < 100; j = j*2) cout << "j = " << j << "\n";
cout << "Checkpoint #3\n";
```

Also, printing values supposed to vary or to remain constant is a good way to spot errors.

5.4.2 Write the same routine twice

Usually, any routine can be written in a short, dirty, computationally expensive and maybe even numerically approximative way. This is a good technique to check that the fancy and correct version does what it is supposed to do. For instance, computation of a derivative

```
double f(double x) {
  return sin(sin(x) + cos(x));
}

double derivative_of_f(double x) {
  // should be (cos(x) - sin(x)) * cos(sin(x) + cos(x));
  return (cos(x) + sin(x)) * cos(sin(x) + cos(x));
}

double derivative_of_f2(double x) {
  return (cos(x) - sin(x)) * cos(sin(x) + cos(x));
}

double dirty_derivative_of_f(double x) {
  double epsilon = 1e-5;
  return (f(x + epsilon) - f(x - epsilon))/(2 * epsilon);
}
```



```
int main(int argc, char **argv) {
  double x= 0.2345;
  cout << "The 1st fancy one: " << derivative_of_f(x) << "\n";
  cout << "The 2nd fancy one: " << derivative_of_f2(x) << "\n";
  cout << "The dirty one:     " << dirty_derivative_of_f(x) << "\n";
}
```

produces

```
The 1st fancy one: 0.43103
The 2nd fancy one: 0.2648
The dirty one:     0.2648
```

5.4.3 Heavy invariants

A last way consists of checking a global property of the result. For instance

```
sort_my_array(a, size);
#ifdef DEBUG
for(int i = 0; i < size-1; i++) if(a[i] > a[i+1]) {
  cerr << "hoho ...\n";
  abort();
}
#endif
```

5.5 Anti-bug tools

5.5.1 gdb

The most standard debugging tool on UNIX is the GNU Debugger `gdb`. Its main functionnality is to display the piece of code which procuced a crash. To do it, compile your code with the `-g` option, so that debugging information will be added to the executable. This information is mainly a correspondance between the machine langage instructions and locations in the source. Then, execute the program from gdb. For instance

```
int main(int argc, char **argv) {
  int size = 100;
```

```
  int a[size];
  for(int i = 0; i < 100 * size; i++) a[i] = i;
}
```

```
> g++ -o bang -g bang.cc
> gdb ./bang
GNU gdb 6.1-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...\
Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) run
Starting program: /tmp/bang

Program received signal SIGSEGV, Segmentation fault.
0x080483e3 in main (argc=1, argv=0xbffff8e4) at bang.cc:4
4         for(int i = 0; i < 100 * size; i++) a[i] = i;
(gdb) l
1       int main(int argc, char **argv) {
2         int size = 100;
3         int a[size];
4         for(int i = 0; i < 100 * size; i++) a[i] = i;
5       }
```

Note that `gdb` is a very primitive tool unable to spot tricky errors such as memory leaks or forbidden access which do not crash the program.

5.5.2 Valgrind

The `valgrind` command is an open-source tool originally developped for the KDE project. It is extremely powerful and simple to use.

You do not need to use special option during compilation, and just have to run your program through `valgrind`. If the program was compiled with the `-g` option, `valgrind` is able to tell what line caused the problem. For instance

```
> valgrind ./bang
==3348== Memcheck, a memory error detector for x86-linux.
==3348== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
```



```
==3348== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==3348== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==3348== For more details, rerun with: -v
==3348==
==3348== Invalid write of size 4
==3348==    at 0x80483E3: main (bang.cc:4)
==3348==  Address 0x202 is not stack'd, malloc'd or (recently) free'd
==3348==
==3348== Process terminating with default action of signal 11 (SIGSEGV)
==3348==  Access not within mapped region at address 0x202
==3348==    at 0x80483E3: main (bang.cc:4)
==3348==
==3348== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 17 from 1)
==3348== malloc/free: in use at exit: 0 bytes in 0 blocks.
==3348== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==3348== For a detailed leak analysis,  rerun with: --leak-check=yes
==3348== For counts of detected errors, rerun with: -v
Segmentation fault
```

Also, valgrind can spot memory leaks. For detailed information, use the `--leak-check=yes` option. For instance, if we compile the following

```
int main(int argc, char **argv) {
  int *a = new int[1000];
}
```

We get with `valgrind`

```
> valgrind --leak-check=yes ./bang
==3376== Memcheck, a memory error detector for x86-linux.
==3376== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==3376== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==3376== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==3376== For more details, rerun with: -v
==3376==
==3376==
==3376== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 17 from 1)
==3376== malloc/free: in use at exit: 4000 bytes in 1 blocks.
==3376== malloc/free: 1 allocs, 0 frees, 4000 bytes allocated.
==3376== For counts of detected errors, rerun with: -v
==3376== searching for pointers to 1 not-freed blocks.
==3376== checked 2388892 bytes.
==3376==
```

```
==3376== 4000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3376==    at 0x1B9072D9: operator new[](unsigned) (vg_replace_malloc.c:139)
==3376==    by 0x804846F: main (bang.cc:2)
==3376==
==3376== LEAK SUMMARY:
==3376==    definitely lost: 4000 bytes in 1 blocks.
==3376==    possibly lost:   0 bytes in 0 blocks.
==3376==    still reachable: 0 bytes in 0 blocks.
==3376==         suppressed: 0 bytes in 0 blocks.
==3376== Reachable blocks (those to which a pointer was found) are not shown.
==3376== To see them, rerun with: --show-reachable=yes
```

Chapter 6

Homework

Submission guidelines

1. Some problems require no programming. Turn them in on paper as usual ;
2. Some problems require programming. Turn in a hard copy of the code ;
3. Some of the programming problems also require you to generate an output. Turn in a hardcopy of the output ;
4. Staple your submission in order. Write your name and account id in the Ryerson Linux lab on top of your submission.

We won't look in your directory necessarily, but we might if something is not clear or if something you have done is particularly intriguing. However, you always need to make your code accessible to us.

Create a directory called `CS116` in your home directory. Make a subdirectory called `hwn` for the n-th homework. Leave the code for problem 1 in `p1.C`, for problem 2 in `p2.C`, etc. If the code for two or more problems, say 3 and 4, is in the same file call it `p34.C`.

Follow some basic basic principles of style :

1. Try to use mnemonic names for variables ;
2. Write brief comments following the declaration of functions and other places where clarification is needed ;
3. Format your code nicely.

Problems

1. (15 points) Some GNU/Linux commands. Use the Linux Tutorial at

   `http`//www.linuxhq.com/guides/GS/gs.html: or
   `http`//www.cc.gatech.edu/linux/LDP/LDP/gs/gs.html:

   (a) Give an exact sequence of shell commands to create in the current directory a directory `sources` containing a directory `project1` and a directory `project2` ;

   (b) Use the man command to find the use of the option `-S` of the `ls` command ;

   (c) What is a wildcard ?

   (d) How would you move all files containing a 'a' from directory `project1` to directory `project2` ;

   (e) Use the man command to find the command and options to remove a directory and all files and directories it contains, recursively (**use with care in real world**).

2. (10 points) Write a program that makes that output :

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9
```

3. (15 points) Write a program that displays a square filled with `.` and whose borders are made of `x` and whose size is given by the user. For example if the user enters 5, he will obtain :

```
xxxxx
x...x
x...x
x...x
xxxxx
```

4. (25 points) Write a program that display the 100 first terms of the Fibonacci sequence

5. (35 points) Write a program that estimates PI by counting the number of points of a square which are in a given disc.

Chapter 7

Cost of algorithm, sorting

7.1 Big-O notation

7.1.1 Why ? How ?

To estimate the efficiency of an algorithm, the programmer has to be able to estimate the number of operations if requires to be executed.

Usually the number of operations is estimated as a function of a parameter (like the number of data to work on, or the expected precision of a computation, etc.)

For example :

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

has a cost proportional to n.

```
for(i = 1; i < n; i = i*2) { ... }
```

has a cost proportional to $\log_2 n$

```
for(i = 0; i < n; i++) for(j = 0; j<n*n; j++) { ... }
```

has a cost proportional to n^3.

7.1.2 Definition of $O(.)$

The classical way to denote an approximation of a complexity is to use the $O(.)$ notation (called "big-O").

If n is a parameter and $f(n)$ the exact number of operations required for that value of the parameter, then we will denote $f(n) = O(T(n))$ and say that f is a big-O of T if and only if :

$$\exists c,\ N,\ \ \forall n \geq N,\ f(n) \leq c.T(n)$$

it means that f is **asymptotically bounded** by a function proportional to T.

Note : a same function can be bounded by different expressions, and the $=$ symbol is odd. Using $\in$ would have been a better choice.

7.1.3 Some $O(.)$

Usually the $O(.)$ notation is useful to hide some superfluous details.

For example if $f(n) = n^2 + 3n$ than for

$$n \geq 3$$

we have

$$3n \leq n^2$$

and thus

$$f(n) \leq 2n^2$$

Finally $f(n) = O(n^2)$.

7.1.4 Summing $O(.)$s

$$f_1(n) = O(T(n)) \text{ and } f_2(n) = O(T(n)) \quad \Rightarrow \quad f_1(n) + f_2(n) = O(T(n))$$

Proof :

$f_1(n) = O(T(n)) \Rightarrow \exists c_1, N_1, \forall n \geq N_1, f_1(n) \leq c_1\, T(n)$

$f_2(n) = O(T(n)) \Rightarrow \exists c_2, N_2, \forall n \geq N_2, f_2(n) \leq c_2\, T(n)$

than we have

$\forall n \geq \max(N_1, N_2), \quad f_1(n) + f_2(n) \leq (c_1 + c_2) T(n)$

The same proof works for products.

7.1.5 Combining $O(.)$s

$$f(n) = O(T(n)) \text{ and } T(n) = O(S(n)) \quad \Rightarrow f(n) = O(S(n))$$

Proof :

$f(n) = O(T(n)) \Rightarrow \exists c, N, \forall n \geq N, f(n) \leq c\, T(n)$

$T(n) = O(S(n)) \Rightarrow \exists d, M, \forall n \geq M, T(n) \leq d\, S(n)$

than we have

$\forall n \geq \max(N, M), \quad f(n) \leq c\, d\, S(n)$

7.1.6 Family of bounds

Most of the bounds can be expressed with powers and log.

Any power of n is a $O(.)$ of any greater power :

$$\forall\ \beta \geq \alpha \geq 0,\ n^\alpha = O(n^\beta)$$

Also, any power of $\log(n)$ is a $O(.)$ of any power of n

$$\forall\ \alpha > 0,\ \beta > 0,\ \log(n)^\alpha = O(n^\beta)$$

$\log(n)$ **is always dominated by any power of** n

For high value of n, one can almost considere $\log(n)$ as a constant.

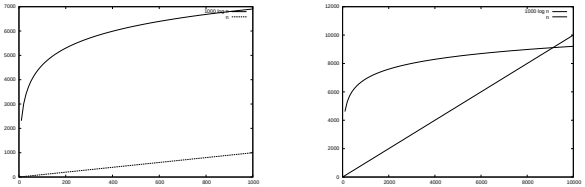


Figure 7.1: Graphs at two different scales of 1000 $\log(n)$ and n. The logarithm of n, even with a big multiplicative constant, is negligeable compared to any power of n.

7.1.7 Some examples of $O(.)$

- $n^2 + n = O(n^2)$
- $sin(n) = O(1)$
- $\log(n^2) = O(\log(n))$
- $\log(n^5) + n = O(n)$
- $n\, \log(n) + n = O(n\, \log(n))$

7.1.8 Estimating the cost of an algorithm

We call **cost** of an algorithm the number of operations it requires to be performed. The $O(.)$ notation is used to give an approximation of this value in **the worst case**.

We have the following rules :

Succession of statements

The cost is the sum of the costs of the statements taken separately :

```
void f(int n) {
  int k;
  for(k = 0; k < n; k++) { .. statement of fixed cost ... }
  for(k = 1; k < n; k = k * 2) { ... statement of fixed cost ... }
}
```

The cost of `f` is $k_1 n + k_2 \log(n)$ so $O(n)$.

Conditional execution

In the following case :

```
if(condition) { statement1; }
else { statement2; }
```

The number of operations is the worst of both, which is actually equal to their sum.

Loops

If the statement cost does not depend upon the value of the loop counter, we can just multiply the cost by the number of loops :

```
void f(int n) {
  int k, j;
  for(k = 0; k<n; k++) for(j = 0; j<n; j++) {
     // statement of constant cost
  }
}
```

If the cost of the statement is a function of the counter, we need to go into details :

```
int triangleSum(int n) {
  int k, k, s;
  for(k = 0; k<n; k++) for(j = 0; j<k; j++) s += j;
  return s;
}
```

In that case the inner loop takes `k` operations, and the main loop is executed `n` times. The complete cost is $1 + 2 + \ldots + (n-1) + n = \frac{n\,(n+1)}{2} = O(n^2)$.

7.1.9 Cost and recursion

The estimation of a recursive function cost leads to recursive expressions.

For example to compute the sum of integers from 0 to n :


```
int sum(int n) {
  if(n == 0) return 0;
  else return n + sum(n-1);
}
```

Denoting f the number of + operations, we have obviously $f(0) = 0$, and $\forall n \geq 1,\ f(n) = f(n-1) + 1$. Which leads to $f(n) = n$.

7.1.10 Average cost

The worst case can be a very bad approximation of the cost of an algorithm. We can consider a case where we want to test if an array of integer contains at least one non-null value :

```
bool thereIsOneNonNull(int *a, int n) {
  for(int k = 0; k<n; k++) if(a[k] != 0) return true;
  return false;
}
```

This procedure terminates as soon as a non-null value is found. In the worst case, the cost is the size of the array.

But if we know that those values have a probability 0.1 to be null. Then, there is a probability 0.1 for the loop to terminate after the first iteration, 0.9×0.1 to terminate after the second, and more generally $0.9^{n-1} \times 0.1$ to terminate after the nth iteration.

We know that

$$\sum_{k=1}^{\infty} x^k\, k \;=\; \frac{1}{(1-x)^2}$$

Finally the average cost is bounded by

$$0.1 \sum_{k=1}^{n} 0.9^{k-1}\, k \;\leq\; 0.1\, \frac{1}{(1-0.9)^2} \;=\; 10$$

and is a $O(1)$!

7.2 Some algorithms

7.2.1 Searching a value in a sorted array

Given a **sorted** array of integers. If we want to find the rank of a given value in that array, we can consider the following routine :

```
// Returns the rank of x in a and -1 if not found
int rank(int x, int *a, int n) {
  for(int k = 0; k<n; k++) if(a[k] == x) return k
  return -1;
}
```

With no hypothesis about the frequency of presence of x in the array, the cost of this routine is $O(n)$

An other implementation would be :

```
#include <iostream>

// Returns the rank of x in a and -1 if not found
int rank2(int x, int *a, int n) {
  // Let's optimize a bit
  if(a[0] > x) return -1;
  if(a[n-1] < x) return -1;

  int i, j;
  i = 0; j = n-1;
  while(i+1 < j) {
    int k = (i+j)/2;
    if(a[k] <= x) i = k; else j = k;
  }
  if(a[i] == x) return i; else if(a[j] == x) return j;
  else return -1;
}

int main(int argc, char **argv) {
  int a[] = {1, 5, 6, 7, 9, 12, 14, 23, 24, 24, 123};
  cout << rank2(14, a, sizeof(a)/sizeof(int)) << '\n';
}
```

The cost here is $O(\log_2(n))$.

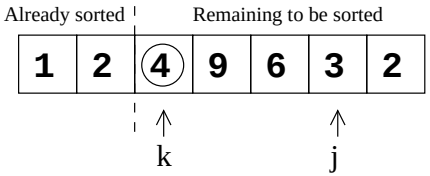


Figure 7.2: The pivot sort consists of swapping the current *pivot* successively with any lesser element located on his right, and then to use as a pivot the element next on his right.

Note that this is dichotomy: looking for a certain value in a sorted table is like looking for the root of a discrete monotonous function.

7.2.2 Pivot sort

A very simple way to sort numbers is to use the **pivot sort** algorithm :

```
#include <iostream>

// We sort in the array itself!
void pivotSort(int *a, int n) {
  int k, j;
  for(k = 0; k<n; k++)
    for(j = k+1; j<n; j++) if(a[k] > a[j]) {
      // Swap a[k] and a[j]
      int t = a[k]; a[k] = a[j]; a[j] = t;
    }
}

int main(int argc, char **argv) {
  int a[] = { 23, 45, 23, 546, 679, 3, 4, 32, 567, 34, 23, 465,
              78, 456, 23 };
  pivotSort(a, sizeof(a)/sizeof(int));
  for(int k = 0; k<sizeof(a)/sizeof(int); k++)
    cout << a[k] << '\n';
}
```

For a given value of `k` the situation is depicted on figure 7.2.

7.3 Simple questions

If, given an array of n `doubles`, we want to find the couple so that the sum is maximal, what is the best strategy ? And if we want to find the two elements so that the absolute value of their difference is the smallest ?

What is the best costs you can imagine for those two problems?

7.4 Fusion sort

The usual dumb algorithms for sorting things require a number of operations proportional to the square of the number of elements to sort ($O(n^2)$). In practice, the used algorithms require a number of operations proportional to $n \times \log n$.

The first one is the **fusion sort**.

The main point is that given two sorted list of numbers, generating the sorted merged list needs a number of operations proportional to the size of this result list. Two index indicate the next elements to take from each list, and one indicates where to store the smallest of the two (see figure 7.3).

This process can be iterated, starting with packets of size 1 (which are *already* sorted ...) and merging them each time two by two (see figure 7.4). After k iterations of that procedure, the packets are of size 2^k, so the number of iterations for this process is $\log_2 n$ where n is the total number of objects to sort.

Each step of this main process cost the sum of the sizes of the resulting packets, which is n. Finally the total number of operations is $\sum_{i=1}^{\log_2 n} n \ = \ n \times \log_2 n$.

7.5 Quick sort

This one is simpler to implement, and widely used in practice. Again it's an application of the *divide and conquer* idea. It consists in choosing one element, and then in splitting the complete set into two half : the elements smaller and the elements larger then the chosen one. Then, the same procedure is applied

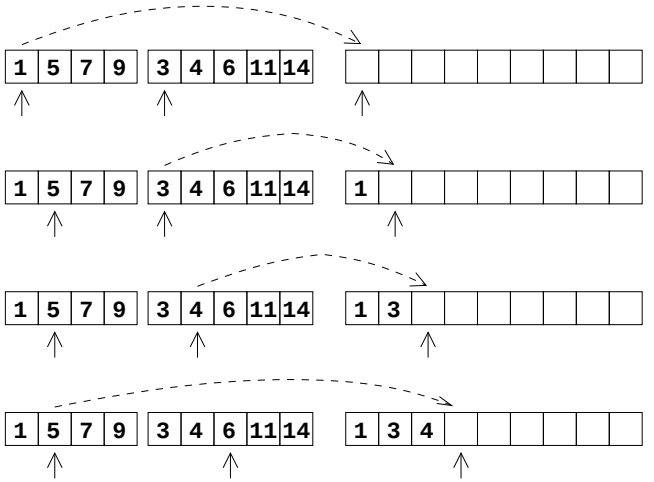


Figure 7.3: Fusionning two groups of sorted elements into a unique sorted group costs a number of operations proportionnal to the total number of elements.

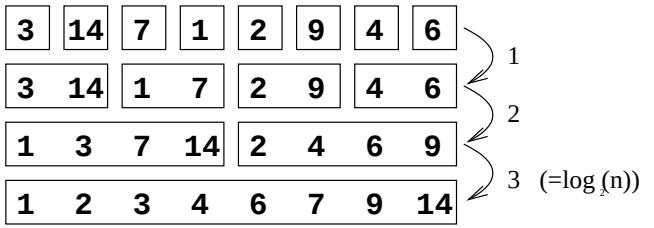


Figure 7.4: The fusion sort consists of grouping at each step pairs of already sorted packets into sorted packets twice bigger.

to each half.

Here each time we take take the first element as the splitting one, and to generate the two half we "fill" the result array starting from the left for the small elements and from the right for the big ones. We put the central one at the end.

If we use each time the first element as the splitting one, the process will require n steps! So the number of operation is between $n \times \log_2 n$ and n^2. A good way to avoid the disaster of n^2 is to pick randomly the splitting element.

7.6 Strategies when two parameters are involved ?

Consider the following operation : having a list of $x_1, \ldots, x_n$ numbers, you have to find which one is the closest to another number y.

This takes n operations if the array is not sorted, and $\log_2 n$ if it is sorted, but sorting would need $n \times \log_2 n$ operations.

If we have to repeat this operation m times, it would take $n \times m$ operations if we do not sort the array first, but only $n \times \log_2 n + m \times \log_2 n$ operations if we sort it first!

Finally in that case, if m is a big number, the cost would be better by sorting the array, and would be finally $O((n+m) \times \log_2 n)$.

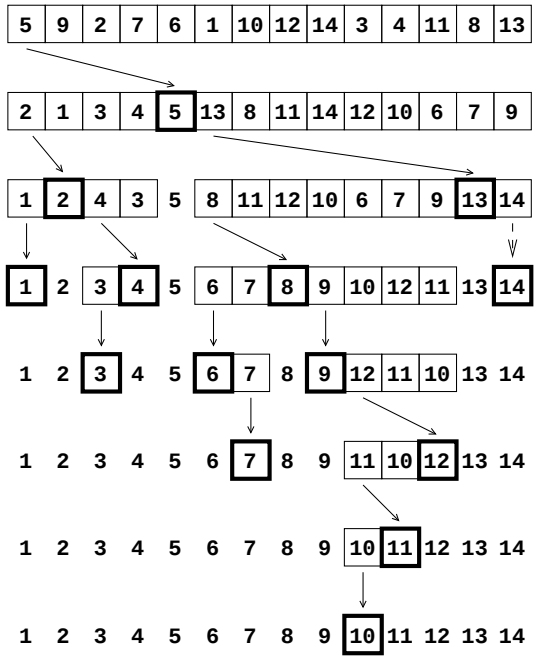


Figure 7.5: The Quick-sort uses at every step the first element (for instance the 5 in the first line) as a separator and organizes the data into a group of smaller elements (2, 1, 3 and 4 in the second line), this splitting value itself, and a group of larger elements (the values 13, 8, ..., 7, and 9). Note that the groups of lesser and larger elements are not themselves srted. They will be in the next steps.

Chapter 8

Creating new types

8.1 Preamble

So far we have used only the built-in types of C++. In many situation this leads to a very non-convenient way of programming. We would like for instance to be able to manipulate arrays with a given size without having to pass both a pointer and an integer each time we want to work with them.

The `class` keyword allow you to define a data structure composed of several built-in type (or other defined types actually).

Each variable of this new type contains several **fields**, each of them with a given type and a given identifier. You can read and write those field by using the identifier of the variable itself, followed by a dot `.` and the identifier of the field. We will see later that we can hide some of the fields to protect the access to them. For now, all our fields can be accessed and are `public`.

8.2 A simple example

```
class Rectangle {
public:
  int width, height;
};

int surface_of_rectangle(Rectangle r) {
  return r.width * r.height;
}

int main(int argc, char **argv) {
  Rectangle r;
  r.width  = 14;
  r.height = 7;
  int surface = surface_of_rectangle(r);
}
```

In this example, we have defined a new class called `Rectangle` which contains two integer data field. In the `main`, we declare such a rectangle and set the values of its two fields and compute its surface.

8.3 Pointers to defined types, and the -> operator

We can also use pointers to the new types. This is very useful to prevent the loss of performances due to multiples copies in memory.

Given a pointer to a given defined type, we can access one of the field by using the identifier of the pointer followed by a `->` symbol and the identifier of the field :

```
int surface_of_rectangle_2(Rectangle *r) {
  return r->width * r->height;
}
```

This will just copy one pointer and not the two field `size` and `elements`.

8.4 Operator definitions, a complex class

We can create a class to deal with complex numbers, and this is a good moment to introduce the fact that **we can also define new operators**. This is possible only because C++ accepts overloaded functions, which allow to have the same operator `+` for example used for different types.

Reminder : if z is a complex number, it can be denoted $z = x + i.y$ where i is a "special number" which verifies $i^2 = -1$. This leads to some simple algebraic operations.

```
class Complex {
public:
  double re, im;
};

Complex operator + (Complex z1, Complex z2) {
  Complex result;
  result.re = z1.re + z2.re;
  result.im = z1.im + z2.im;
  return result;
}

Complex operator * (Complex z1, Complex z2) {
  Complex result;
  result.re = z1.re * z2.re - z1.im * z2.im;
  result.im = z1.im * z2.re + z1.re * z2.im;
  return result;
}
```

The preceding definitions can be used the following way :

```
int main(int argc, char **argv) {
  Complex x, y;
  x.re =  5.0; x.im = 12.0;
  y.re = -1.0; y.im =  4.0;
  Complex z = x + (x*y) + y;
  cout << z.re << " + i." << z.im << '\n';
}
```

Displays

```
-49 + i.24
```

8.5 Passing by value vs. passing by reference

There is almost no reason in such a situation to use pass-by-value parameters. Using references will lead to the same efficiency as pointers and the same syntax as values.

All the operations described so far can be re-written with references.

8.6 Some timing examples

```
class AnArray {
public:
  int values[1000];
};

int max(AnArray a) {
  int m = a.values[0];
  for(int i = 1; i<1000; i++) if(a.values[i] > m) m = a.values[i];
  return m;
}

int main(int argc, char **argv) {
  AnArray a;
  int i, m;
  for(i = 0; i<1000; i++) a.values[i] = i;
  for(i = 0; i<100000; i++) m = max(a);
}
```

Executing `times ./test` returns :

```
real    0m4.080s
user    0m4.010s
sys     0m0.020s
```

The same program with references :

```
class AnArray {
public:
  int values[1000];
};

int maxByRef(AnArray &a) {
  int m = a.values[0];
  for(int i = 1; i<1000; i++) if(a.values[i] > m) m = a.values[i];
  return m;
}

int main(int argc, char **argv) {
  AnArray a;
  int i, m;
  for(i = 0; i<1000; i++) a.values[i] = i;
```

```
  for(i = 0; i<10000; i++) m = maxByRef(a);
}
```

Executing `times ./test` returns :

```
real    0m0.432s
user    0m0.430s
sys     0m0.010s
```

Chapter 9

Object-Oriented programming

9.1 Intro

The "object approach", which is the fundamental idea in the conception of `C++` programs, consists in building the programs as an interaction between objects :

1. For all part of the program that use a given object, it is defined by the **methods** you can use on it ;
2. you can take an existing object and add data inside and methods to manipulate it, this is call inheritance.

The gains of such an approach are :

1. Modularity : each object has a clear semantic (`Employer` or `DrawingDevice`), a clear set of methods (`getSalary(), getAge()`, or `drawLine(), drawCircle()` ;
2. Less bugs : the data are accessed through the methods and you can use them only the way to object's creator wants you to :
3. Re-use : you can extend an existing object, or you can build a new one which could be use in place of the first one, as long as it has all the methods required (for example the `Employer` could be either the CEO or a worker, both of them having the required methods but different data associated to them. `DrawingDevice` could either be a window, a printer, or anything else).

9.2 Vocabulary

- A **class** is the definition of a data structure and the associated operations that can be done on it ;
- an **object** (equivalent to a variable) is an **instanciation** of the class, i.e. an existing set of data build upon the model described by the class ;
- a **data field** is one of the variable internal to the object containing a piece of data ;
- a **method** is a special function associated to a class.

9.3 Protected fields

Some of the data fields of a class can be hidden. By default, they are, and it's why we have used the `public` keyword in preceding examples. You can specify explicitly some fields to be "hidden" with the `private` keywords :

```
class Yeah {
  int a;
public:
  int b;
  double x;
private:
  double z;
};

int main(int argc, char **argv) {
  Yeah y;
  y.a = 5;
  y.b = 3;
  y.x = 2.3;
  y.z = 10.0;
}
```

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:2: ‘int Yeah::a’ is private
/tmp/chose.cc:12: within this context
/tmp/chose.cc:7: ‘double Yeah::z’ is private
/tmp/chose.cc:15: within this context
```

9.4 Methods

The `class` keyword allows you to associate to the data type you create a set of **methods** with privileged access to the inner structure of the object. Those functions must be seen as the *actions* you can do on your object. They are very similar to standard functions, except that they are associated to a class and can be called only for a given object.

```
class Matrix {
  int width, height;
  double *data;
public:
  void init(int w, int h) {
    width = w; height = h;
    data = new double[width * height];
  }

  void destroy() { delete[] data; }

  double getValue(int i, int j) {
    return data[i + width*j];
  }

  void setValue(int i, int j, double x) {
    data[i + width*j] = x;
  }
};
```

9.5 Calling methods

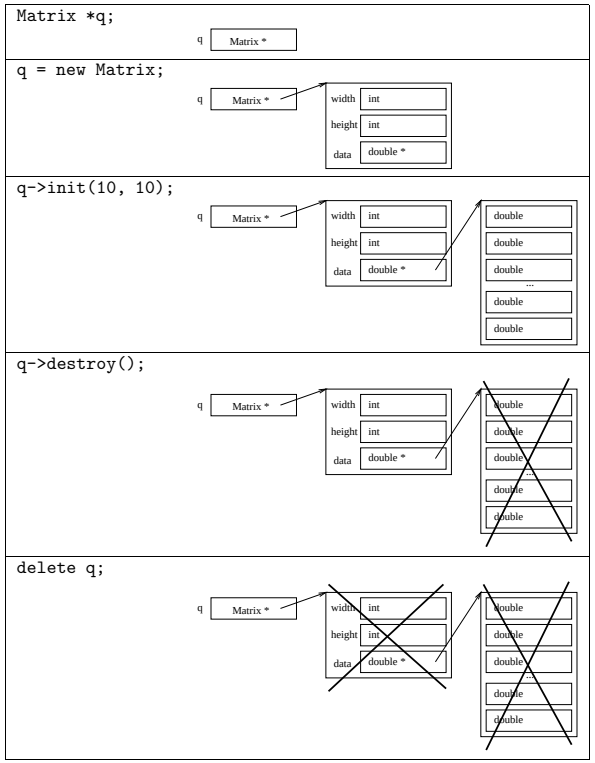
As for fields, the syntax is either the dot-notation `.` or the arrow-notation `->` :

```
int main(int argc, char **argv) {
  Matrix m;
  m.init(20, 20);
  for(int i = 0; i<20; i++) m.setValue(i, i, 1.0);
  m.destroy();

  Matrix *q;
  q = new Matrix;
  q->init(10, 10);
  for(int i = 0; i<10; i++) q->setValue(i, i, 1.0);
```



```
  q->destroy(); // here we deallocate q->data but not q itself
  delete q;     // here we deallocate q itself
}
```

9.6 Some memory figures



9.7 Separating declaration and definition

We have seen that we can separate the declaration (i.e. giving the name of the function, its return type and the number and types of its parameters) and the definition (i.e. the code itself).

For methods it's the same, but we need a syntax to specify the class a function belongs to (the same name can be used for member functions of different classes). The syntax is `<class name>::<function name>`.

The methods identifier can be used alone in the member functions statement.

```
class Small {
  int x;
public:
  void setValue(int a);
};

class Bigger {
  int x, y;
public:
  void setValue(int a);
};

void Small::setValue(int a) { x = a; }
void Bigger::setValue(int a) { x = a; y = a*a; }
```

9.8 Protection of data integrity

This access through methods is very efficient to protect the integrity of data and control the out of bounds errors :

```
class Matrix {
  int width, height;
  double *data;
public:
  void init(int w, int h) {
    width = w; height = h;
    data = new double[width * height];
  }

  void destroy() { delete[] data; }
```



```
  double getValue(int i, int j) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     return data[i + width*j];
  }

  void setValue(int i, int j, double x) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     data[i + width*j] = x;
  }
};
```

9.9 Abstraction of concepts

This notion of matrix, and the associated method can also be used for a special class of matrix with only ONE non-null coefficient. This matrix would allow you to store one value at one location.

```
class MatrixAlmostNull {
  int width, height;
  int x, y;
  double v;
public:
  void init(int w, int h) { width = w; height = h; v = 0.0; }
  void destroy() { }

  double getValue(int i, int j) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     if((i == x) && (j == y)) return v; else return 0.0;
  }

  void setValue(int i, int j, double vv) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     if((v == 0.0) || ((x == i) && (y == j))) {
         x = i;
         y = j;
         v = vv;
     } else abort();
  }
};
```

9.10 Constructors

In the preceding examples, we have used each time one function to initialize the object and another one to destroy it. We know that for any object those two tasks have to be done.

The C++ syntax defines a set of special methods called **constructors**. Those methods have the same name as the class itself, and do not return results. The are called when the variable of that type is defined :

```
#include <iostream>
#include <cmath>

class NormalizedVector {
  double x, y;
public:
  NormalizedVector(double a, double b) {
    double d = sqrt(a*a + b*b);
    x = a/d;
    y = b/d;
  }
  double getX() { return x; }
  double getY() { return y; }
};

int main(int argc, char **argv) {
  NormalizedVector v(23.0, -45.0);
  cout << v.getX() << ' ' << v.getY() << '\n';
  NormalizedVector *w;
  w = new NormalizedVector(0.0, 5.0);
  cout << w->getX() << ' ' << w->getY() << '\n';
  delete w;
}
```

The same class can have many constructors :

```
#include <iostream>
#include <cmath>

class NormalizedVector {
  double x, y;
public:
  NormalizedVector(double theta) {
    x = cos(theta);
```



```
    y = sin(theta);
  }

  NormalizedVector(double a, double b) {
    double d = sqrt(a*a + b*b);
    x = a/d;
    y = b/d;
  }
  double getX() { return x; }
  double getY() { return y; }
};
```

9.11 Default constructor

A default constructor can be called with no parameters, and is used if you define a variable with no initial value.

```
class Something {
public:
  Something() {};
};

class SomethingElse {
public:
  SomethingElse(int x) {};
};

int main(int argc, char **argv) {
  Something x;
  SomethingElse y;
}
```

compilation returns

```
/tmp/chose.cc: In function `int main(int, char **)':
/tmp/chose.cc:13: no matching function for call to
                  `SomethingElse::SomethingElse ()'
/tmp/chose.cc:8: candidates are:
                   SomethingElse::SomethingElse(int)
/tmp/chose.cc:9:
                   SomethingElse::SomethingElse(const
                                        SomethingElse &)
```

9.12 Destructor

The symmetric operation is the destruction of objects. This is required as soon as the object dynamically allocates other objects.

The special method defined to do that is called the **destructor**, and is called as soon as the compiler need to deallocate an instance of the class. There is only one destructor per class, which return no value, and has no parameter. The name of the destructor is the class name prefixed with a ~.

We can now re-write our matrix class :

```
class Matrix {
  int width, height;
  double *data;
public:
  Matrix(int w, int h) {
    width = w; height = h;
    data = new double[width * height];
  }

  ~Matrix() { delete[] data; }

  double getValue(int i, int j) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     return data[i + width*j];
  }

  void setValue(int i, int j, double x) {
     if((i<0) || (i>=width) || (j<0) || (j>=height)) abort();
     data[i + width*j] = x;
  }
};
```

9.13 Tracing precisely what is going on

```
#include <iostream>

class Something {
  char *name;
public:
  Something(char *n) {
    name = n; cout << "Creating " << name << '\n';
  }
  ~Something() { cout << "Destroying " << name << '\n'; }
};

int main(int argc, char **argv) {
  Something x("x"), y("y");
  Something *z = new Something("z");
  Something w("w");
  { Something v("v"); }
  delete z;
}
```

```
Creating x
Creating y
Creating z
Creating w
Creating v
Destroying v
Destroying z
Destroying w
Destroying y
Destroying x
```


9.14 The member operators

We have seen that we can define our own operators. We can also define class operators. Here we redefine the bracket operator, with one integer parameter. By returning a reference to a value, the result of the `[]` operator is a lvalue, and finally we can use those new arrays like standard arrays!

```
#include <iostream>

class OneDArray {
  int size;
  double *data;
public:
  OneDArray(int s) { size = s; data = new double[size]; }
  ~OneDArray() { delete[] data; }
  double &operator [] (int k) {
    if((k < 0) || (k >= size)) abort();
    return data[k];
  }
```

```
};

int main(int argc, char **argv) {
  OneDArray a(10);
  for(int i = 0; i<10; i++) a[i] = 1.0/i;
  for(int i = 0; i<10; i++)
    cout << "a[" << i << "] = " << a[i] << '\n';
  a[14] = 1.0;
}
```

displays :

```
a[0] = inf
a[1] = 1
a[2] = 0.5
a[3] = 0.333333
a[4] = 0.25
a[5] = 0.2
a[6] = 0.166667
a[7] = 0.142857
a[8] = 0.125
a[9] = 0.111111
Aborted
```

A simple vector class to illustrate the `+` operator redefinition. The passing by reference is just used here to increase the performances by avoiding a copy. Note that the precise meaning of the operation `v + w` is here `v.(operator +)(w)`.

The `=` operator is implicitly defined by the compiler and just copies the two field.

```
#include <iostream>

class TwoDVector {
  double x, y;
public:
  TwoVector() { x = 0; y = 0; }
  TwoDVector(double a, double b) { x = a; y = b; }
  TwoDVector operator + (TwoDVector &v) {
    return TwoDVector(x + v.x, y + v.y) ;
  }
  void print() { cout << x << ' ' << y << '\n'; }
};
```



```
int main(int argc, char **argv) {
  TwoDVector v(2, 3);
  TwoDVector w(4, 5);
  TwoDVector z;
  z = v+w;
  z.print();
}
```

displays `6 8`.

9.15 Summary for classes

Properties of a class :

- Corresponds to a data-structure, defined with several **data fields** ;
- each data field has a **type** and an **identifier** ;
- data fields can be **public** or **private** ;
- a instantiation of a class is called an **object** and is the same as a variable ;
- **methods** are functions that can be applied to an object and have privileged access to the data fields ;
- methods are called with either the `.` operator or the `->` operator if we use a pointer to an object ;
- **constructors** are special functions called when creating an instance of the class, they do not return types and have for identifier the same identifier as the class itself ;
- the **destructor** is a special method called when an object is destructed, is has no return value and has for identifier the class name prefixed by a `~` ;
- we can also define **member operators** ;
- we can define method out of the class statement by using the `<class name>::<member name>` syntax.

Chapter 10

Homework

1. **Simple introduction question** (5 points)

   Using a `for` loop, write a function to compute the k-th power of a number :

   ```
   double power(double x, int k)
   ```

2. **Non-trivial recursion** (15 points)

   You can note that $x^{2k} = (x^k)^2$ and $x^{2k+1} = x.(x^k)^2$. Write a function `double sq(double x)` to compute the square of a number, and use it to write a recursive version of the `power` function :

   ```
   double powerRec(double x, int k)
   ```

3. **Evaluate a polynomial** (25 points)

   A polynomial has the form $f(x) = \sum_{i=0}^{n-1} a_i x^i$. Write a function to evaluate a polynomial, given the value of x, the number of coefficients, and their values $a_0, \ldots, a_{n-1}$:

   ```
   double evalPolynomial(double x, double *a, int n)
   ```

   Note that the computation can be also written $f(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_{n-1}))$, reducing both the number of additions and products to $n-1$. Write a second version of the evaluation function :

   ```
   double evalPolynomialEfficient(double x, double *a, int n)
   ```


4. **Allocating and returning arrays** (25 points)

   Given two matrices $A = (a_{1,1}, \ldots, a_{l,m})$ and $B = (b_{1,1}, \ldots, b_{m,n})$, we define the product of A and B as the matrix $C = (c_{1,1}, \ldots, c_{l,n})$ with $\forall i, j \ : c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$. Write a function :

   ```
   double **matrixProduct(double **a, double **b, int l, int m, int n)
   ```

   returning the product of two matrices.

5. **More complex memory management** (30 points)

   Using a `for` loop, write an exponentiation function to compute the k-th power of a matrix $A = (a_{1,1}, \ldots, a_{l,m})$:

   ```
   double **matrixExpon(double **a, int l, int m, int k)
   ```

Chapter 11

Detail of class definitions

11.1 Example

```
#include <iostream>

class SimpleClass {
  char *name;
  int value;
public:
  SimpleClass(char *n, int v) {
    cout << "  " << n << ".SimpleClass("
         << n << ", " << v << ")\n";
    name = n; value = v;
  }
  ~SimpleClass() {
    cout << "  " << name << ".~SimpleClass()\n";
  }
  void changeValue(int v) {
    cout << "  " << name << ".changeValue(" << v << ")\n";
    value = v;
  }
  int readValue() {
    cout << "  " << name << ".readValue()\n";
    return value;
  }
  int copy(SimpleClass &sc) {
    cout << "  " << name << ".copy(" << sc.name << ")\n";
    value = sc.value;
```



```
  }
};

int main(int argc, char **argv) {
  SimpleClass x("x", 12);
  SimpleClass y("y", 14);
  x.copy(y);
  cout << x.readValue() << '\n';
  y.changeValue(10);
  cout << y.readValue() << '\n';
}
```

```
  x.SimpleClass(x, 12)
  y.SimpleClass(y, 14)
  x.copy(y)
  x.readValue()
14
  y.changeValue(10)
  y.readValue()
10
  y.~SimpleClass()
  x.~SimpleClass()
```

11.2 An “integer set” example

We may need an object to store integers. We want to be able to do the two following operations :

```
void add(int i);
bool contains(int i);
```

we will be able then to do something like that :

```
int main() {
  IntegerSet mySet;
  for(int k = 0; k<100; k++) mySet.add(k);
  mySet.add(14);
  mySet.add(4);
  mySet.add(3);
  mySet.add(12323);
  mySet.add(17);
```

```
  cout << mySet.contains(3) << '\n';
  cout << mySet.contains(310) << '\n';
}
```

The first version would need to set the maximum size when the set if built, like this :

```
class IntegerSet {
  int *data;
  int sizeMax, currentSize;
public:
  IntegerSet(int sm) {
    sizeMax = sm;
    data = new int[sizeMax];
    currentSize = 0;
  }

  ~IntegerSet() { delete[] data; }

  void add(int i) {
    if(currentSize < sizeMax) data[currentSize++] = i;
    else {
      cerr << "ouch!\n";
      abort();
    }
  }

  bool contains(int i) {
    for(int k = 0; k < currentSize; k++)
      if(i == data[k]) return true;
    return false;
  }
};
```

This is not very convenient : the size has to be fixed at the beginning.

A new version would increase the size of the array when it's full :

```
class IntegerSet {
  int *data;
  int sizeMax, currentSize;
public:
  IntegerSet() {
    sizeMax = 10; data = new int[sizeMax];
```



```
    currentSize = 0;
  }

  ~IntegerSet() { delete[] data; }

  void add(int i) {
    if(currentSize == sizeMax) {
      int *tmp = new int[sizeMax*2];
      for(int k = 0; k < sizeMax; k++) tmp[k] = data[k];
      delete[] data;
      sizeMax = sizeMax*2;
      data = tmp;
    }
    data[currentSize++] = i;
  }

  bool contains(int i) {
    for(int k = 0; k < currentSize; k++)
      if(i == data[k]) return true;
    return false;
  }
}
```

In that case, the `contains` is a $O(currentSize)$, which is very bad. We could solve this by using a sorted set, so that `contains` could be a $log_2\ currentSize$. This can be achieved by two means :

1. keeping all the time a sorted version of the set ;
2. sort the set when we do a `contains` and keep a flag to tells if the set is sorted or not.

Putting aside the memory management, what is in those two cases the cost of `add(int i)` ?

11.3 The `const` keyword

The `const` keyword can be used to specify which parameters will not be modified (when they are passed by reference) and also to specify if the object itself will be modified.

11.4 The `this` pointer

We will need sometime to be able to have a pointer to the object the method is applied to. A special pointer exists, of type the class itself, the identifier is `this`.

```
#include <iostream>

class ADouble {
  double value;
public:
  ADouble(double v) { value = v; }
  void multiply(ADouble d) { value = value * d.value; }
  void squared() { multiply(*this); }
  double getValue() const { return value; }
};

int main(int argc, char **argv) {
  ADouble x(-12.0);
  x.squared();
  cout << x.getValue() << '\n';
}
```

11.5 The = operator vs. the copy constructor

The copy constructor is called either when you initialize a variable (with the = operator!), or when you pass parameters by value. Precisely, given a class `Something`, the lines `Something x = y;` and `Something x(y);` are equivalent.

The = is called for all other = assignments.

```
#include <iostream>

class AInteger {
  int value;
public:
  AInteger() { value = 0; }
  AInteger(int i) { value = i; }

  AInteger(const AInteger &ai) {
    value = ai.value;
    cout << "   copy\n";
  }

  // As we have seen, this operator is also an expression.
  // To be consistent, we have to return a reference to the
  // result of the assignment, which is the object itself
  AInteger &operator = (const AInteger &ai) {
    value = ai.value;
    cout << "   =\n";
    return *this;
  }
};

void nothing(AInteger a, AInteger b, AInteger c) { }

int main(int argc, char **argv) {
  AInteger i(14), j;
  cout << "AInteger k = i;\n";
  AInteger k = i;
  cout << "k = i;\n";
  j = i;
  cout << "nothing(i);\n";
  nothing(i, j, k);
}
```

```
AInteger k = i;
   copy
k = i;
   =
nothing(i);
   copy
   copy
   copy
```


11.6 Default copy constructor and default = operator

By default the copy constructor and the assignment operator exist, but just copy the field one by one. This is great when no pointers are into the story, but when we have some, this is really a bad idea :

```
class Array {
  int *data;
```

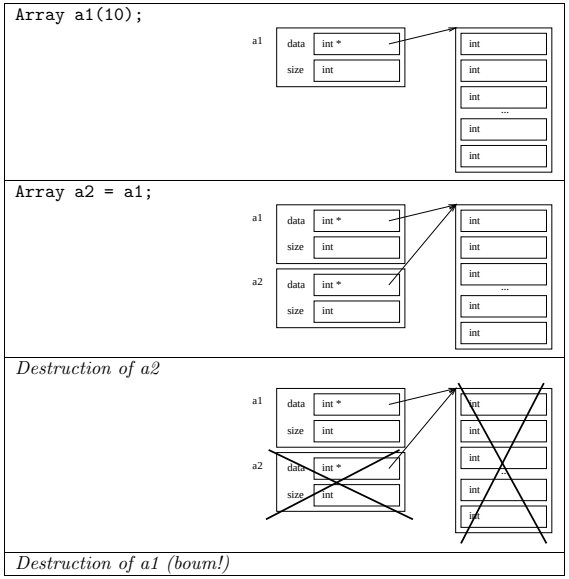
```
  int size;
public:
  Array(int s) { size = s; data = new int[size]; }
  ~Array() { delete[] data; }
};

int main(int argc, char **argv) {
  Array a1(10);
  Array a2 = a1;
}
```

produces a

```
Segmentation fault
```

Because the `a2 = a1` copied the `data` field, and thus the `delete` of `~Array` for both the destruction of `a1` and the destruction of `a2` were done on the same pointer!

11.7 Some memory figures



11.8 A matrix class

Considering all we have seen so far, we can now build a consistent matrix class :

```
class Matrix {
  int width, height;
  double *data;
public:
  Matrix();
  Matrix(int w, int h);
  Matrix(const Matrix &m);
  ~Matrix();
  bool operator == (const Matrix &m) const;
  Matrix &operator = (const Matrix &m);
```

```
  Matrix operator + (const Matrix &m) const;
  Matrix operator * (const Matrix &m) const;
  double &operator () (int i, int j);
  void print() const;
};
```

```
Matrix::Matrix() { width = 0; height = 0; data = 0; }

Matrix::Matrix(int w, int h) {
  cout << "  Matrix::Matrix(int w, int h)\n";
  width = w; height = h;
  data = new double[width * height];
}

Matrix::Matrix(const Matrix &m) {
  cout << "  Matrix::Matrix(const Matrix &m)\n";
  width = m.width; height = m.height;
  data = new double[width * height];
  for(int k = 0; k<width*height; k++) data[k] = m.data[k];
}

Matrix::~Matrix() {
  cout << "  Matrix::~Matrix()\n";
  delete[] data;
}
```

```
Matrix &Matrix::operator = (const Matrix &m) {
  cout << "  Matrix &operator = (const Matrix &m)\n";
  if(&m != this) {
    delete[] data;
    width = m.width; height = m.height;
    data = new double[width * height];
    for(int k = 0; k<width*height; k++) data[k] = m.data[k];
    return *this;
  }
}
```

```
bool Matrix::operator == (const Matrix &m) const {
  cout << "  bool operator == (const Matrix &m) const\n";
  if(width != m.width || height != m.height) return false;
  for(int k = 0; k<width*height; k++) if(data[k] != m.data[k]) return false;
  return true;
}
```



```
Matrix Matrix::operator + (const Matrix &m) const {
  cout << "  Matrix operator + (const Matrix &m) const\n";
  if(width != m.width || height != m.height) {
    cerr << "Size error!\n";
    abort();
  }

  Matrix result(width, height);
  for(int k = 0; k<width*height; k++) result.data[k] = data[k] + m.data[k];

  return result;
}

Matrix Matrix::operator * (const Matrix &m) const {
  cout << "  Matrix operator * (const Matrix &m) const\n";
  if(width != m.height) {
    cerr << "Size error!\n";
    abort();
  }

  Matrix result(m.width, height);
  for(int i = 0; i<m.width; i++)
    for(int j = 0; j<height; j++) {
      double s = 0;
      for(int k = 0; k<width; k++) s += data[k + j*width] * m.data[i + m.width*k];
      result.data[i + m.width*j] = s;
    }

  return result;
}
```

```
double &Matrix::operator () (int i, int j) {
  cout << "  double & operator () (int i, int j)\n";
  if(i<0 || i>=width || j<0 || j >= height) {
    cerr << "Out of bounds!\n";
    abort();
  }
  return data[i + width*j];
}

void Matrix::print() const {
  cout << "  void print() const\n";
  for(int j = 0; j<height; j++) {
    for(int i = 0; i<width; i++) cout << " " << data[i + width * j];
    cout << "\n";
```

```
  }
}
```

```
int main(int argc, char **argv) {
  cout << "DOING Matrix m(3, 2), n(5, 3);\n";
  Matrix m(3, 2), n(5, 3);
  cout << "DOING Matrix x = m*n;\n";
  Matrix x = m*n;
  cout << "DOING m.print();\n";
  m.print();
  cout << "DOING m = n;\n";
  n = m;
  cout << "DOING n.print();\n";
  n.print();
  cout << "DOING x.print();\n";
  x.print();
}
```

```
DOING Matrix m(3, 2), n(5, 3);
  Matrix::Matrix(int w, int h)
  Matrix::Matrix(int w, int h)
DOING Matrix x = m*n;
  Matrix operator * (const Matrix &m) const
  Matrix::Matrix(int w, int h)
  Matrix::Matrix(const Matrix &m)
  Matrix::~Matrix()
DOING m.print();
  void print() const
 0 0 0
 0 0 0
DOING m = n;
  Matrix &operator = (const Matrix &m)
DOING n.print();
  void print() const
 0 0 0
 0 0 0
DOING x.print();
  void print() const
 0 0 0 0 0
 0 0 0 0 0
  Matrix::~Matrix()
  Matrix::~Matrix()
  Matrix::~Matrix()
```

Chapter 12

More details about class definitions

12.1 Back to operators

Has we have seen, **operators** can be either defined as functions, or as methods. When they are defined as member of a class, the left operand is the object itself, thus **they take one less parameter than expected**.

```
class Complex {
public:
  double re, im;
  Complex() { re = 0.0; im = 0.0; }
  Complex(double r, double i) { re = r; im = i; }
  Complex operator * (const Complex &z) const {
    return Complex(re*z.re - im*z.im, re*z.im + im*z.re);
  }
};

// We can do that because the fields are public
Complex operator + (const Complex &z1, const Complex &z2) {
    return Complex(z1.re + z2.re, z1.im + z2.im);
}

int main(int argc, char **argv) {
  Complex z(2.0, 3.0), w(3.0, -4.0);
  Complex x;
```



```
  x = z + w; // equivalent to x = (operator +) (z, w)
  x = z * w; // equivalent to x = z.(operator *) (w)
}
```

We will see later how we can defined functions with privileges to access the private fields.

12.2 Implicit conversion

An very powerful property of constructor is the implicit usage the compiler can do to convert one type to another. For instance :

```
#include <iostream>

class Complex {
  double re, im;
public:
  Complex() {
    cout << "Complex::Complex()\n";
    re = 0.0; im = 0.0;
  }
  Complex(double x) {
    cout << "Complex::Complex(double)\n";
    re = x; im = 0.0;
  }
  Complex(double r, double i) {
    cout << "Complex::Complex(double, double)\n";
    re = r; im = i;
  }
  Complex operator + (const Complex &z) const {
    cout << "Complex::operator + (const Complex &z) const\n";
    return Complex(re + z.re, im + z.im);
  }
  Complex operator * (const Complex &z) const {
    cout << "Complex::operator * (const Complex &z) const\n";
    return Complex(re*z.re - im*z.im, re*z.im + im*z.re);
  }
};
```

```
int main(int argc, char **argv) {
  Complex z = 3.0;
  Complex y;
```

```
  y = 5.0;
}
```

we obtain :

```
Complex::Complex(double)
Complex::Complex()
Complex::Complex(double)
```

The compiler is also able to look for all methods (resp. operators) available for `Complex`, and to check if the argument can be converted to fit as a parameter (resp. right operand) :

```
int main(int argc, char **argv) {
  Complex z = 3.0;
  Complex y;
  y = z + 5.0;
}
```

```
Complex::Complex(double)
Complex::Complex()
Complex::Complex(double)
Complex::operator + (const Complex &z) const
Complex::Complex(double, double)
```

But this it is not able to do the same if it has to convert the object itself (i.e. the left operand for an operator) :

```
int main(int argc, char **argv) {
  Complex z = 3.0;
  Complex y;
  y = 5.0 + z;
}
```

At compilation time we get :

```
/tmp/chose.cc:30: no match for ‘double + Complex &’
```

This can be fixed using non-member operators.

12.3 Private methods

methods can, as data fields, be private. This allows the designer of a class to hide some non-secure functions from the class user.

```
#include <cmath>

class NormalizedVector {
  double x, y;
  void normalize() {
    double n = sqrt(x*x + y*y);
    x /= n; y /= n;
  }
public:
  NormalizedVector(double xx, double yy) {
    x = xx;
    y = yy;
    normalize();
   }
};

int main(int argc, char **argv) {
  NormalizedVector v(3.4, -2.3);
}
```

12.4 Hiding the copy constructor

To detect superfluous copies, we can define the copy constructor and the `=` operator as private :

```
class BigThing {
  int value[1000];
  BigThing(const BigThing &bt) {
    for(int k = 0; k<1000; k++) value[k] = bt.value[k];
  }
  BigThing &operator = (const BigThing &bt) {
    for(int k = 0; k<1000; k++) value[k] = bt.value[k];
  }
public:
  BigThing() {
    for(int k = 0; k<1000; k++) value[k] = 0;
  }
```

```
  int set(int k, int v) {
    if((k<0) || (k>=1000)) abort();
    value[k] = v;
  }

  int get(int k) {
    if((k<0) || (k>=1000)) return 0;
    else                   return value[k];
  }
};
```

```
int main(int argc, char **argv) {
  BigThing x;
  BigThing y = x;
}
```

we obtain (at compilation) :

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:3: ‘BigThing::BigThing(const BigThing &)’ is private
/tmp/chose.cc:27: within this context
```

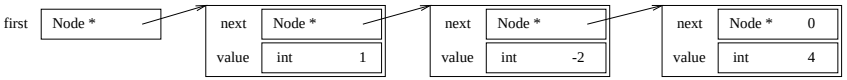
12.5 A linked list class

A very standard structure is a **linked list**, which allows to store a succession of objects in such a way that adding new elements has a constant cost.

We first define a type of this kind :

```
class Node {
public:
  Node *next;
  int value;
  Node(Node *n, int v) { next = n; value = v; }
  ...
```

Such a class allows us to link several values. The convention is that when the `next` field is equal to zero, this is the end of the list. For instance, we could explicitly create a list of three values with the following declaration :




```
  Node *first = new Node(new Node(new Node(0, 4), -2), 1);
```

Would lead to the following figure of the memory :

To be more precise, this is stricly equivalent to doing :

```
  Node *a = new Node(0, 4);
  Node *b = new Node(a, -2);
  Node *c = new Node(b, 1);
  Node *first = c;
```

Except that in that case we create 3 variables which are not required.

12.5.1 The `Node` class

We can be more precise in the definition of this class. We want to be able to create a node, to delete a node and all the linked one, recursively. We also have in mind to copy list, so we need to be able to duplicate a node and all the next ones, and we want to be able to test if two list are equal :

```
#include <iostream>

class Node {
public:
  Node *next;
  int value;
  Node(Node *n, int v);
  void deleteAll();
  bool contains(int v);
  int size();
  Node *cloneAll();
  bool equalAll(Node *n);
};
```

```
Node::Node(Node *n, int v) {
  next = n;
  value = v;
}

void Node::deleteAll() {
  if(next) next->deleteAll();
  delete this;
}

bool Node::contains(int v) {
  return (value == v) || (next && (next->contains(v)));
}

int Node::size() {
  if(next) return 1+next->size();
  else return 1;
}
```

```
Node *Node::cloneAll() {
  if(next) return new Node(next->cloneAll(), value);
  else     return new Node(0, value);
}

bool Node::equalAll(Node *n) {
  if(n) {
    if(value != n->value) return false;
    if(next) return next->equalAll(n->next);
    else return n->next == 0;
  } return false;
}
```

12.5.2 The `LinkedList` class

We can now define the list itself. It hides a `Node` pointer and deals with complex memory management related to constructors, copy, comparisons, etc.

```
class LinkedList {
  Node *first;
public:
  LinkedList();
  LinkedList(const LinkedList &l);
  ~LinkedList();
```



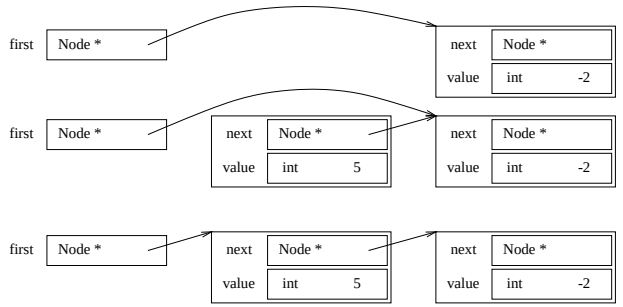
```
  void add(int v);
  LinkedList &operator = (const LinkedList &l);
  bool operator == (const LinkedList &l) const;
  bool contains(int v) const;
  int size() const;
  void print() const;
};
```

```
LinkedList::LinkedList() {
  first = 0;
}

LinkedList::LinkedList(const LinkedList &l) {
  if(l.first) { first = l.first->cloneAll(); }
  else first = 0;
}

LinkedList::~LinkedList() {
  if(first) first->deleteAll();
}

LinkedList &LinkedList::operator = (const LinkedList &l) {
  if(&l != this) {
    if(first) first->deleteAll();
    if(l.first) first = l.first->cloneAll();
    else first = 0;
  }
  return *this;
}
```

```
void LinkedList::add(int v) {
  first = new Node(first, v);
}
```

The `add` function creates a new node and puts it in place of the first one.

```
bool LinkedList::operator == (const LinkedList &l) const {
  if(first) return first->equalAll(l.first);
  else return l.first == 0;
}
```




```
bool LinkedList::contains(int v) const {
  return first && first->contains(v);
}

int LinkedList::size() const {
  if(first) return first->size(); else return 0;
}

void LinkedList::print() const {
  Node *n;
  for(n = first; n != 0; n = n->next) {
    cout << n->value;
    if(n->next) cout << ", ";
    else cout << "\n";
  }
}
```

```
int main(int argc, char **argv) {
  LinkedList l;
  l.add(13);
  cout << l.contains(12) << " " << l.contains(13) << "\n";
  for(int i = 0; i < 10; i++) l.add(i);
  cout << "[" << l.size() << "] ";
  l.print();

  LinkedList m = l;
  cout << (l == m) << "\n";

  cout << "[" << m.size() << "] ";
```



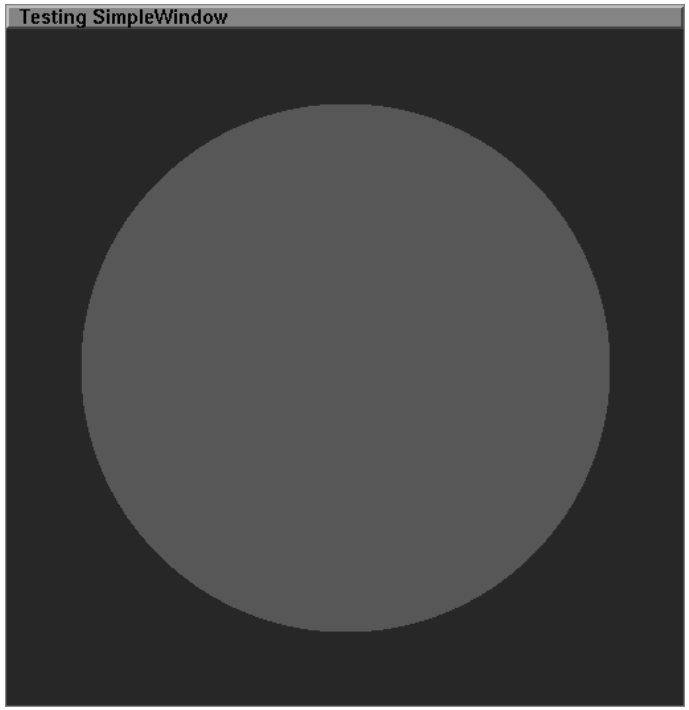
```
  m.print();
  m.add(19);

  cout << (l == m) << "\n";
}
```

12.6 The graphical library

For certain examples of this course, you have to use the `simple_api` library, which provides a `SimpleWindow` class. The methods are the following :

```
class SimpleWindow {
public:
  SimpleWindow(char *name, int w, int h);
  ~SimpleWindow();

  int getWidth();
  int getHeight();

  void color(float red, float green, float blue);
  void drawPoint(int x, int y);
  void drawLine(int x1, int y1, int x2, int y2);
  void drawCircle(int x, int y, int r);
  void drawText(char *s, int x, int y);
  void fillRectangle(int x, int y, int w, int h);
  void show();
  void fill();
};
```

```
int main() {
  SimpleWindow window("Testing SimpleWindow", 512, 512);
  for(int x = 0; x<512; x++) for(int y = 0; y<512; y++) {
    if((x-256)*(x-256) + (y-256)*(y-256) < 200*200) window.color(1.0, 0, 0);
    else window.color(0, 0, 1.0);
    window.drawPoint(x, y);
  }
  window.show();
  cin.get();
}
```


Testing SimpleWindow

Chapter 13

More about methods

13.1 Rvalues, lvalues, references, and `const` qualifier

We have seen that a lvalue is an expression corresponding to value and its location in memory, which means that it can be modified. A rvalue is just a value and one can not modify it.

Thus, passing a reference to a rvalue is meaningless. Nevertheless, for performances, we can pass an intermediate result as a `const` reference.

```
double nothing1(double x) {}
double nothing2(double &x) {}
double nothing3(const double &x) {}

int main(int argc, char **argv) {
  nothing1(3+4);
  nothing2(3+4);
  nothing3(3+4);
}
```

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:7: initializing non-const ‘double &’ with ‘int’ will use a temporary
/tmp/chose.cc:2: in passing argument 1 of ‘nothing2(double &)’
```


13.2 Methods can be called through standard functions

An elegant way to offer nice functions an operators and still use the data-field protection principles is to design a set of member functions with privileged access to the data field and a set of standard functions and operators which call the methods.

```
#include <iostream>

class ForSureNonNull {
  double value;
public:
  ForSureNonNull(double v) {
    if(v == 0) { cerr << "Are you crazy ?!\n"; abort(); }
    value = v;
  }

  double getValue() const {
    return value;
  }
};

double sum(const ForSureNonNull &n1, const ForSureNonNull &n2) {
  return n1.getValue() + n2.getValue();
}

int main(int argc, char **argv) {
  ForSureNonNull x(15);
  double k = sum(x, x);
}
```

13.3 Overloading the << operator

The usage of `cout` is very convenient. The operators re-definition allows us to define our own `<<` operator.

As we have seen, `cout` is of type `ostream`, and an expression such as :

```
cout << a << b << c;
```

Will be evaluated from left to right as :

```
( (cout << a) << b) << c;
```

so, the left operand of << will always be an ostream and the right operand will be whatever we want :

```
ostream &operator << (ostream &s, const ForSureNonNull &x) {
  return (s << x.getValue());
}
```

13.4 Overloading the >> operator

The left operand of >> will always be an istream and the right operand will be whatever we want :

```
#include <iostream>

class Crazy {
public:
  double a, b;
};

istream & operator >> (istream &i, Crazy &c) {
  return i >> (c.a) >> (c.b);
}

int main(int argc, char **argv) {
  Crazy x;
  cin >> x;
}
```

The ostream can not be copied, and will always exist as a lvalue (by definition printing modifies its state), so you have to alway pass it by reference and return a reference :

```
#include <iostream>

void dumb1(ostream &s) {}
void dumb2(ostream s) {}

int main(int argc, char **argv) {
  dumb1(cout);
```



```
  dumb2(cout);
}
```

```
/usr/lib/gcc-lib/i586-pc-linux-gnu/2.95.1/../../../../include/g++-3/streambuf.h:12
  `ios::ios(const ios &)' is private
```

Here the line 8 tries to pass the stream by value, thus to call a copy constructor, which is private.

13.5 An example about what has been said before

```
#include <iostream>

class Vector3D {
  double x, y, z;
public:
  // We use the default copy constructor and = operator
  Vector3D() { x = 0.0; y = 0.0; z = 0.0; }
  Vector3D(double xx, double yy, double zz) { x = xx; y = yy; z = zz; }
  Vector3D sum(const Vector3D &v) const { return Vector3D(x+v.x, y+v.y, z+v.z); }
  Vector3D product(double k) const { return Vector3D(k*x, k*y, k*z); }
  double scalar(const Vector3D &v) const { return x*v.x + y*v.y + z*v.z; }
  ostream &print(ostream &s) const { return s << '[' << x << ',' << y << ',' << z
};

Vector3D operator + (const Vector3D &v1, const Vector3D &v2) { return v1.sum(v2);
double operator * (const Vector3D &v1, const Vector3D &v2) { return v1.scalar(v2);
Vector3D operator * (double k, const Vector3D &v) { return v.product(k); }
Vector3D operator * (const Vector3D &v, double k) { return v.product(k); }
ostream &operator << (ostream &s, const Vector3D &v) { v.print(s); return s; }

int main(int argc, char **argv) {
  Vector3D v(1, 2, 3), w(-1.0, -1.0, 1.0);
  cout << v << ' ' << w << '\n';
  cout << (v*w) << ' ' << (3*v + 5*w + (v*w)*w) << '\n';
}
```

13.6 A bit more about streams : output formats

We can fix the number of digits with `precision` :

```
#include <iostream>

int main(int argc, char **argv) {
  cout << "Standard precision " << (1.0/3.0) << '\n';
  cout.precision(3);
  cout << "precision(3) " << (1.0/3.0) << '\n';
}
```

outputs

```
Standard precision 0.333333
precision(3) 0.333
```

13.7 A bit more about streams : files

The `cout` is not the only `ostream` available around. For example, you can open any file and use it as an `ostream` :

```
#include <iostream>
#include <fstream>

void letSCount(ostream &s, int k) {
  for(int n = 0; n<k; n++) s << n << '\n';
}

int main(int argc, char **argv) {
  letSCount(cout, 50);
  ofstream myFile("count.txt");
  letSCount(myFile, 20);
}
```

13.8 Inline functions

An interesting mechanism to increase the performances of a program consists in replacing function calls by the function itself. To specify to the compiler to

do that, we can use the `inline` keyword :

```
#include <iostream>

inline double dumb1(double x) { return 17*x; }
double dumb2(double x) { return 17*x; }

int main(int argc, char **argv) {
  double x = 4;
  cout << x << '\n';
  x = dumb1(x);
  cout << x << '\n';
  x = dumb2(x);
  cout << x << '\n';
}
```

13.9 First steps with inheritance

A very powerful mecanism of the OO approach consists in extending existing class through the mecanism of inheritance. Basically, it allows you to create a new class by addin members (both data and functions) to an existing class. And you new class can be used wherever the old one was used.

We call the new class a **subclass** of the old one, which is its **superclass**.

13.10 Adding methods

We have to define a new class, which inherits from the first one. We have to define the constructors, which can call the constructors of the initial class. And we can add functions.

```
#include <iostream>

class First {
  double x;
public:
  First(double y) { x = y; }
  bool positive() { return x >= 0.0; }
  double getValue() { return x; }
};
```

```
class Second : public First {
public:
  Second(double z) : First(z) {};
  bool positiveAndNonNull() { return positive() && ( getValue() != 0.0 ); }
};

bool bothPositive(First x, First y) { return x.positive() && y.positive(); }

int main(int argc, char **argv) {
  Second x(3), y(3);
  bothPositive(x, y);
}
```

13.11 Adding data fields

```
#include <iostream>

class Student {
  char *name;
  int age;
public:
  Student(char *n, int a) { name = n; age = a; }
  char *getName() { return name; }
  int getAge() { return age; }
};

class ForeignStudent : public Student {
  char *country;
public:
  ForeignStudent(char *n, int a, char *c) : Student(n, a) { country = c; }
  char *getCountry() { return country; }
};

bool sameAge(Student s1, Student s2) {
  return s1.getAge() == s2.getAge();
}

int main(int argc, char **argv) {
  Student s1("Jack", 21);
  ForeignStudent s2("Steven", 21, "UK");
  bool same = sameAge(s1, s2);
}
```


13.12 Multiple inheritance

A very powerful way to combine the properties of several class is to use multiple-inheritance. With such mechanism, the obtained class possess all data fields and methods from its superclasses.

```
class Mamal {
  double weight, temperature, ageMax;
public:
  ...
}

class FlyingAnimal {
  double distanceMax, altitudeMax;
public:
  ...
}

class Bat : public Mamal, public FlyingAnimal {
}
```

13.13 Tricky issues with multiple inheritance

The main problem appears when data fields or methods with same names are present in both superclasses.

```
class Truc {
public:
  int chose() {}
};

class Machin {
public:
  int chose() {}
};

class Bidule : public Truc, Machin {
};

int main() {
  Bidule x;
```

```
  x.chose();
}
```

This can not compile:

```
chose.cc: In function ‘int main()’:
chose.cc:16: error: request for member ‘chose’ is ambiguous
chose.cc:8: error: candidates are: int Machin::chose()
chose.cc:3: error:                 int Truc::chose()
```

Chapter 14

Homework

14.1 Costs and big-O (10 points)

Give the exact number of calls to `soCool()` as a function of n, and a big-O estimation, for the following pieces of programs :

1.
```
for(i = -6*n; i < 6*n; i += 3) soCool();
```

2.
```
for(i = 0; i < n*n; i++) for(j = i; j > 0; j--) soCool();
```

3.
```
i = n;
while(i > 0) {
  soCool();
  i = i/2;
}
```

4.
```
i = 0;
do {
  for(j = i-2; j < i+2; j++) soCool();
  i = i+1;
} while(i < n);
```

5.
```
for(i = 0; i < n*n; i++) if(i%n == 0) soCool();
```


14.2 Quick-sort (30 points)

Write a function :

```
void qsort(double *orig, double *result, int n)
```

that takes the `n` doubles from the array pointed by `orig`, sorts them with the quick-sort algorithm, and copies them after sort into the array pointed by `result`. This function is recursive and calls itself two times.

The following `main()` fills an array with random numbers between 0 and 1 and displays them after sort :

```
#include <iostream>

void qsort(double *orig, double *result, int n) {
  // ...
}

// This line tells the compiler to allow to use the Linux
// random-generator as a C++ function
extern "C" double drand48();

int main(int argc, char **argv) {
  int size = 100;

  double *dat = new double[size];
  double *result = new double[size];

  for(int n = 0; n < size; n++) dat[n] = drand48();
  qsort(dat, result, size);
  for(int n = 0; n < size; n++) cout << result[n] << "\n";

  delete[] result;
  delete[] dat;
}
```

14.3 The Mandelbrot set (30 points)

An interesting problem is the study of the initial conditions of a dynamic process that allow it to be stable. A very simple example is the following : consider the

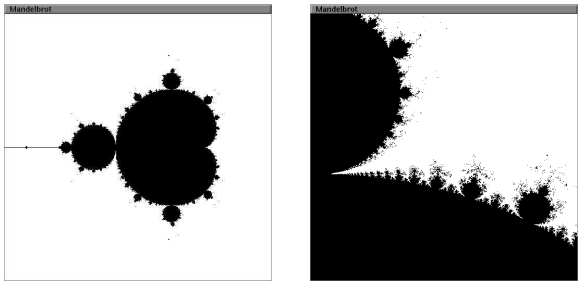


Figure 14.1: Two views of the Mandelbrot set, corresponding to the squares $[-2, 1] \times [-1.5, 1.5]$ (left) and $[-0.13, 0.27] \times [-0.83, -0.53]$ (right).

complex sequence : $z_0 = 0$ and $z_{n+1} = z_n^2 + c$. We can wonder for what values of c this series remains bounded. The set of such points is called the Mandelbrot set (see figure 14.1).

To make a *graphical answer* to this question, we can draw the set of points of the square $[-2, 1] \times [-1.5, 1.5]$ corresponding to values of c such that the 100 first terms of this sequence are in the disc of radius 10. So, using the `libcs116` from the class web site, write a program that :

1. Opens a square window ;
2. loops through all points of the window, and for each of them :
   (a) computes the c value associated to it ;
   (b) checks that the 100 first terms of the sequence are in the disc of radius 10 ;
   (c) displays a white point if this is not true, a black one if this is true.

Chapter 15

Inheritance

Note: Mandelbrot

```
#include "swindow.h"

int main() {
  SimpleWindow window("Mandelbrot", 512, 512);
  int n;

  for(int i = 0; i<window.getWidth(); i++)
    for(int j = 0; j<window.getHeight(); j++) {
      double cr = -0.13 + 0.3 * (double(i)/512.0);
      double ci = -0.83 + 0.3 * (double(j)/512.0);
      //double cr = -2.0 + 3.0 * (double(i)/512.0);
      //double ci = -1.5 + 3.0 * (double(j)/512.0);
      double zr = 0, zi = 0;
      for(n = 0; (n<100) && (zr*zr + zi*zi < 100); n++) {
        double t = zr*zr -zi*zi + cr;
        zi = 2*zr*zi + ci;
        zr = t;
      }
      if(n < 100) window.color(1.0, 1.0, 1.0);
      else window.color(0.0, 0.0, 0.0);
      window.drawPoint(i, j);
    }
  window.show();

  int k;
```



```
  cin >> k;
}
```

15.1 Adding member data field and functions

We have seen that a class is defined by a set of **data fields** and a **methods**. All operations done on a giving object, access the data field either directly or through the methods.

The main idea of **inheritance** is to create new class by extending existing ones. This is done by adding methods and member data fields.

Doing this, we ensure that all operations that could be done on the initial class can still be done on the new one.

15.2 Syntax to inherit

To create a **derived class** (or subclass), the syntax is similar to the declaration of a new class, but we have to specify what is the initial class it inherits from :

```
#include <iostream>

class Vehicle {
public:
  double speed, mass;
  double kineticEnergy() {
    return 0.5 * mass * speed * speed;
  }
};

class Plane : public Vehicle {
public:
  double altitude;
  double totalEnergy() {
    return kineticEnergy() + mass * altitude;
  }
};

bool nonNullEnergy(Vehicle v) {
  return v.kineticEnergy() != 0.0;
}
```

```
int main(int argc, char **argv) {
  Plane p;
  p.speed = 150.0; p.mass = 1500.0; p.altitude = 1300;
  if(nonNullEnergy(p)) cout << "There is some energy.\n";
}
```

15.3 Syntax to call constructors

In the preceding example, we were using the default constructor and filling the fields one by one. If we want to use the constructor syntax, we have to call the existing constructors in the new class :

```
#include <iostream>

class Vehicle {
  double speed, mass;
public:
  Vehicle(double s, double m) {
    speed = s; mass = m;
  }

  double kineticEnergy() {
    return 0.5 * mass * speed * speed;
  }
};

class Plane : public Vehicle {
  double altitude;
public:
  Plane(double a, double s, double m) : Vehicle(s, m) {
    altitude = a;
  }

  double totalEnergy() {
    return kineticEnergy() + mass * altitude;
  }
};
```

We can use the same syntax to initialize the various fields :

```
#include <iostream>
```



```
class Vehicle {
  double speed, mass;
public:
  Vehicle(double s, double m) : speed(s), mass(m) {};
  double kineticEnergy() {
    return 0.5 * mass * speed * speed;
  }
};

class Plane : public Vehicle {
  double altitude;
public:
  Plane(double a, double s, double m) : Vehicle(s, m), altitude(a) { }
  double totalEnergy() {
    return kineticEnergy() + mass * altitude;
  }
};
```

15.4 An example

Given the `SimpleWindow` class from the `libcs116` library, we can create a new object to draw histograms.

The existing interface is the following :

```
class SimpleWindow {
public:
  SimpleWindow(char *name, int w, int h);
  ~SimpleWindow();

  int getWidth();
  int getHeight();

  void color(float red, float green, float blue);
  void drawPoint(int x, int y);
  void drawLine(int x1, int y1, int x2, int y2);
  void drawCircle(int x, int y, int r);
  void drawText(char *s, int x, int y);
  void fillRectangle(int x, int y, int w, int h);
  void show();
  void fill();
};
```

We want to add the possibility to have n bars, each of them with a given value. But we do not care anymore to specify the size and name of the window, which will be fixed.

```
#include <iostream>
#include "swindow.h"

class Histogram : public SimpleWindow {
  double *barValues;
  int nbars;
public:
  Histogram(int n);
  ~Histogram();
  void setBarValue(int k, double v);
};

Histogram::Histogram(int n) : SimpleWindow("Histogram", 256, 256) {
  nbars = n;
  barValues = new double[nbars];
}

Histogram::~Histogram() { delete[] barValues; }
```

```
void Histogram::setBarValue(int k, double v) {
  int i, j;

  if((k<0) || (k>=nbars) || (v < 0)) abort();
  barValues[k] = v;
  double vmax = barValues[0];
  for(int k = 0; k<nbars; k++) if(barValues[k] > vmax) vmax = barValues[k];
  vmax = vmax*1.2;
  color(1.0, 1.0, 1.0);
  fill();
  color(0.0, 0.0, 0.0);
  for(int k = 0; k<nbars; k++) {
    i = (getWidth()*k) / nbars;
    j = int(getHeight() * (1 - barValues[k]/vmax));
    drawLine(i, j, i + getWidth()/nbars, j);
  }
  show();
}

int main() {
  Histogram hi(25);
  for(int k = 0; k<25; k++) hi.setBarValue(k, 1+ sin((2*M_PI*k)/25));
```



```
  cin.get();
}
```

15.5 Tracing what's going on "inside"

The calls to the constructors / destructors is (again) pretty complex. Let's trace what's going on :

```
#include <iostream>

class A {
public:
  A() { cout << "Constructor for A\n"; }
  ~A() { cout << "Destructor for A\n"; }
  int dummy() { return 42; }
};

class B : public A {
public:
  B() : A() { cout << "Constructor for B\n"; }
  ~B() { cout << "Destructor for B\n"; }
};

class C : public B {
public:
  C() : B() { cout << "Constructor for C\n"; }
  ~C() { cout << "Destructor for C\n"; }
};

int main() {
  C c;
  cout << c.dummy() << '\n';
}
```

```
Constructor for A
Constructor for B
Constructor for C
42
Destructor for C
Destructor for B
Destructor for A
```

15.6 The `protected` keyword

For performance reasons, we can specify that a given field can not be accessed except by methods, but can be accessed by inherited classes. Such a member is called a `pteocted` member.

```
class Student {
protected:
  char *name;
  int age;
public:
  Student(char *n, int a) { name = n; age = a; }
};

class ForeignStudent : public Student {
  int nbYearsInTheUS;
public:
  ForeignStudent(char *n, int a, int nbytu) : Student(n, a),
                                              nbYearsInTheUS(nbytu) {}
  bool moreThanHalfHisLifeInTheUS() { return nbYearsInTheUS*2 > age; }
};

int main(int argc, char **argv) {
  ForeignStudent student("Sergei", 19, 4);
  student.age = 13;
}
```

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:4: ‘int Student::age’ is protected
/tmp/chose.cc:18: within this context
```

15.7 Hiding the superclass

We can set private the properties of the original superclass :

```
class Something {
  int value;
public:
  Something(int v) : value(v) {}
  int getValue() { return value; }
};
```



```
class SomethingElse : private Something {
  int value2;
public:
  SomethingElse(int v) : Something(v), value2(v) {}
};

int main(int argc, char **argv) {
  SomethingElse se(5);
  int k = se.getValue();
}
```

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:5: ‘int Something::getValue()’ is inaccessible
/tmp/chose.cc:16: within this context
```

15.8 Ambiguities between different members with the same name

We can also inherits from different classes. In such a case, ambiguities can appear between different members with the same identifier, from different classes :

```
class AnInteger {
  int value;
public:
  AnInteger(int v) : value(v) {}
  int getValue() { return value; }
};

class ADouble {
  double value;
public:
  ADouble(double v) : value(v) {}
  double getValue() { return value; }
};

class OneOfEach : public AnInteger, public ADouble {
public:
  OneOfEach(int i, double d) : AnInteger(i), ADouble(d) {}
  double sum() { return ADouble::getValue() + AnInteger::getValue(); }
};
```

```
int main() {
  OneOfEach x(2, 3.0);
  double s = x.sum();
  double u = x.getValue();
  double t = x.AnInteger::getValue();
}
```

```
/tmp/chose.cc: In function ‘int main()’:
/tmp/chose.cc:24: request for member ‘getValue’ is ambiguous
/tmp/chose.cc:12: candidates are: double ADouble::getValue()
/tmp/chose.cc:5:                  int AnInteger::getValue()
```

15.9 method overload and calls

The method called is always the one of the type when the call is done :

```
#include <iostream>

class A {
public:
  void dummy() { cout << "A::dummy\n"; }
};

class B : public A {
public:
  void dummy() { cout << "B::dummy\n"; }
};

void callAsA(A x) { x.dummy(); }

int main(int argc, char **argv) {
  B b;
  b.dummy();
  callAsA(b);
}
```

displays

```
B::dummy
A::dummy
```

Same when the function is called from another one :

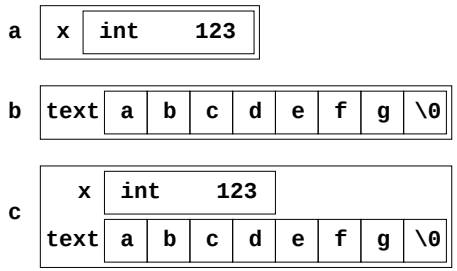

```
#include <iostream>

class A {
public:
  void dummy() { cout << "A::dummy\n"; }
  void something() { dummy(); }
};

class B : public A {
public:
  void dummy() { cout << "B::dummy\n"; }
};

void callAsA(A x) { x.dummy(); }

int main(int argc, char **argv) {
  B b;
  b.something();
}
```

displays

```
A::dummy
```

15.10 What's going on in the memory ?

```
#include <iostream>

class A {
  int x;
public:
  A(int y) { x = y; }
};

class B {
  char text[8];
public:
  B(char *s) { char *t = text; while(*t++ = *s++); }
};

class C : public A, public B{
public:
```




```
  C(int v, char *s) : A(v), B(s) {};
};

void printMemory(char *x, int s) {
  cout << "size = " << s << '\n';
  for(int k = 0; k<s; k++) {
    cout << int(x[k]);
    if(k < s-1) cout << ' '; else cout << '\n';
  }
  cout << '\n';
}
```

```
int main() {
  A a(123); printMemory((char *) &a, sizeof(a));
  B b("abcdefg"); printMemory((char *) &b, sizeof(b));
  C c(123, "abcdefg"); printMemory((char *) &c, sizeof(c));
}
```

```
size = 4
123 0 0 0

size = 8
97 98 99 100 101 102 103 0

size = 12
123 0 0 0 97 98 99 100 101 102 103 0
```


15.11 Memory addresses can change!

The reference of the same object can change when we convert it to one of its superclass :

```
void refAsA(A &x) { cout << &x << '\n'; }
void refAsB(B &x) { cout << &x << '\n'; }
void refAsC(C &x) { cout << &x << '\n'; }

int main() {
  C y(123, "abcdefg");
  refAsA(y);
  refAsB(y);
  refAsC(y);
}
```

```
0xbffffb88
0xbffffb8c
0xbffffb88
```

Chapter 16

Exercises

16.1 Find the bug!

```
int main(int argc, char **argv) {
  int *a = new int[100];
  for(i = 1; i<=100; i++) a[i] = 5;
}
```

Out of bounds : the index should goes from 0 to 99 not to 100.

16.2 Find the bug!

```
double *smoothing(double *x, int size, int width) {
  double *result = new double[size];
```



```
  for(int i = 0; i<size; i++) {
    double s = 0;
    for(int j = 0; j<width; j++) s += x[(i+j)%size];
    result[i] = s/width;
  }
  return result;
}

double *iterativeSmoothing(double *x, int size, int width, int nb) {
  double *result;
  result = x;
  for(int k = 0; k<nb; k++) result = smoothing(result, size, width);
  return result;
}

int main(int argc, char **argv) {
  double a[] = {1, 2, 3, 4, 5, 6, 7};
  double *s = iterativeSmoothing(a, sizeof(a)/sizeof(double), 3, 100);
  delete[] s;
}
```

There is a huge memory leak in the iterative form of the smoothing!

The function could be re-written that way :

```
double *iterativeSmoothing(double *x, int size, int width, int nb) {
  double *result = new double[size];
  for(int k = 0; k<size; k++) result[k] = x[k];
  for(int k = 0; k<nb; k++) {
    double *tmp = smoothing(result, size, width);
    delete[] result;
    result = tmp;
  }
  return result;
}
```

16.3 Find the bug!

```
class A {
  A(const A &x) {}
public:
  A();
  void dummy(A x) {}
```

```
};

int main(int argc, char **argv) {
  A y;
  y.dummy(y);
}
```

The copy constructor `A(const A &x)` is private and thus can not be used when the call to `dummy` requires a copy of `y`.

16.4 Find the bug!

```
class A {
  int *something;
public:
  A() { something = new int(42); }
  ~A() { delete something; }
};

int main(int argc, char **argv) {
  A x;
  A y = x;
}
```


The default copy constructor called by `A y = x;` just copies each field separately. So `x.something` and `y.something` points to the same object, and the same dynamically created `int` will be deleted by the two destructors.

16.5 Find the bug!

```
class First {
  First(int k) {}
public:
  First() {}
};

class Second : public First {
public:
  Second(int k) : First(k) {}
};

int main(int argc, char **argv) {
  Second x(3);
}
```

The constructor `First(int k)` is private and thus can not be called by the constructor in class `Second`.

16.6 What is printed ?

```
#include <iostream>

int main(int argc, char **argv) {
  int x = 3;
  if(x = 4) cout << x << '\n';
  else cout << "x is not equal to 4";
}
```

The `if(x = 4)` does not test if `x` is equal to 4 (which could be done by `if(x == 4)`), but assign 4 to `x` and then convert 4 to a `bool`, which is `true`. Thus, the program prints 4 on the screen.

16.7 What is printed ?

```
#include <iostream>

int main(int argc, char **argv) {
  int x = 24;
  do {
    while(x%5 > 0) x--;
    cout << x << '\n';
    x--;
  } while(x > 0);
}
```

```
20
15
10
5
0
```

16.8 What is printed ?

```
#include <iostream>

class A {
public:
  A() { cout << "#1\n"; }
  A(const A &a) { cout << "#2\n"; }
  A(double x) { cout << "#3\n"; }
  ~A() { cout << "#4\n"; }
};
```

```
int main(int argc, char **argv) {
  A x = 3.0;
  A y;
  y = x;
}
```

```
#3
#1
#4
#4
```

16.9 What is printed ?

```
#include <iostream>

class AnInteger {
  int k;
public:
  AnInteger() { cout << "#1\n"; }
  AnInteger(const AnInteger &i) { k = i.k; cout << "#2\n"; }
  AnInteger(int n) { k = n; cout << "#3\n"; }
  AnInteger operator + (const AnInteger &i) const {
    cout << "#4\n";
    return AnInteger(k + i.k);
  }
};

int main(int argc, char **argv) {
  AnInteger x = 3;
```



```
  AnInteger y = x + 3 + 4;
}
```

```
#3
#3
#3
#4
#3
#4
#3
```

Chapter 17

Exercices

17.1 Find the bug!

```
#include <iostream>

double *something(int n) {
  double a[n];
  double *x = a;
  return x;
}

int main(int argc, char ** argv) {
  double *z = something(10000);
  double *w = something(10000);
  delete z;
  delete w;
}
```

17.2 Find the bug!

```
#include <iostream>

int main(int argc, char **argv) {
 int k, n;
 cin >> k;
 while(n < k) cout << n++ << '\n';
```



```
}
```

17.3 Find the bug!

```
#include <iostream>

int main(int argc, char **argv) {
  int a[100];
  int k, n = 0;
  for(int i = 0; i<100; i++) a[i] = i;
  cin >> k;
  do {
    n += k;
    cout << a[n] << '\n';
  } while(n < 100);
}
```

17.4 Find the bug!

```
int kindOfLog2(int n) {
  if(n < 0) return 1;
  else return 1 + kindOfLog2(n/2);
}

int main(int argc, char ** argv) {
  int k = kindOfLog2(987);
}
```

17.5 Find the bug!

```
#include <iostream>

int main(int argc, char **argv) {
  double s;
  for(double x = 0; x != 1; x += 0.01) s += 1/(1+x);
  cout << s << '\n';
}
```

17.6 When does it bug ?

```
int estimation(int k, int l) {
  int n = 0;
  for(int i = 0; i<k; i++)
    for(int j = 0; j<k; j++)
      if(i*i + j*j <= k*k) n++;
  return (4*n*l)/(k*k);
}
```

17.7 Find the bug!

```
class BusinessClass {
public:
  void dummy() { }
}

class EconomyClass {
public:
  void dummy() { }
}

class ReallyDumbClass : public BusinessClass, public EconomyClass {
public:
  void moreDumb() { dummy(); }
}
```

17.8 Find the bug!

```
class Polynomial {
  double *coeff;
  int size;
public:
  Polynomial(double *c, int s) { coeff = c; size = s; }
  Polynomial() { coeff = 0; }
  ~Polynomial() { delete coeff; }
};

int main(int argc, char **argv) {
  double a[] = { 1, 2, 3, 4 };
```



```
  Polynomial p(a, sizeof(a)/sizeof(double));
}
```

17.9 What is printed ?

```
#include <iostream>

class Vector {
  double *coord;
  double dim;
public:
  Vector(double *c, int d) {
    coord = new double[d];
    dim = d;
    for(int k = 0; k<dim; k++) coord[k] = c[k];
  }
  ~Vector() { delete coord; }


  bool operator == (const Vector &v) const {
    return dim == v.dim && coord == v.coord;
  }
};

int main(int argc, char **argv) {
  double a[] = { 0.1, 2.0, 3.5 };
  Vector u(a, sizeof(a)/sizeof(double));
  Vector v(a, sizeof(a)/sizeof(double));
  if(u == v) cout << "u is equal to v!!!\n";
  else       cout << "u is not equal to v!!!\n";
}
```

17.10 What is printed ?

```
#include <iostream>

class A {
  int x;
public:
  A(int y) { x = y; }
  void dummy() { cout << "x = " << x << '\n'; }
```

```
};

class B : public A {
  int z;
public:
  B(int k) : A(k), z(2*k) {}
  void dummy() { cout << "Hello!\n"; }
};

void f(B r, A q) {
  r.dummy();
  q.dummy();
}

int main(int argc, char **argv) {
  B x(3);
  x.dummy();
  x.A::dummy();
  f(x, x);
}
```

```
Hello!
x = 3
Hello!
x = 3
```

17.11 Non trivial inheritance

We have seen a linked list class :

```
class LinkedList {
  Node *first;
public:
  LinkedList();
  LinkedList(const LinkedList &l);
  ~LinkedList();
  void add(int v);
  LinkedList &operator = (const LinkedList &l);
  bool operator == (const LinkedList &l) const;
  bool contains(int v) const;
  int size() const;
  void print() const;
};
```


The `size()` function was very non-efficient. If we know that we need now to call it frequently, it would be wiser to keep the size in a new field.

```
class LinkedList2 : public LinkedList {
  int keepSize;
public:
  LinkedList2();
  LinkedList2(const LinkedList &l);
  LinkedList2(const LinkedList2 &l);
  LinkedList2 &operator = (const LinkedList2 &l);
  void add(int v);
  int size() const;
};

LinkedList2::LinkedList2() : LinkedList(), keepSize(0) {}

LinkedList2::LinkedList2(const LinkedList &l) : LinkedList(l),
                                                keepSize(l.size()) {}

LinkedList2::LinkedList2(const LinkedList2 &l) : LinkedList(l),
                                                 keepSize(l.keepSize) {}

LinkedList2 &LinkedList2::operator = (const LinkedList2 &l) {
  LinkedList::operator =(l);
  keepSize = l.keepSize;
}

void LinkedList2::add(int v) { keepSize++; LinkedList::add(v); }
int LinkedList2::size() const { return keepSize; }
```

Chapter 18

Homework

18.1 Various questions (20 points)

Write four lines in English for each question.

1. How can you control the access to certain data fields ?
2. How can you specify to the compiler that an existing type can be implicitly converted to a type you define yourself ?
3. What happens when a type you define yourself is passed by value to a function ?
4. Why is it sometime useful to use parameters passed by references instead of passing them by value ?
5. What would be the data fields for a `Node` class used for a list containing pairs of doubles ?

18.2 A polynomial class (80 points)

This class uses **dynamically allocated arrays for the coefficients**. Write all the methods given below and be careful with memory management. The coefficient of the highest degree must be **always** different than 0, so that you **never** store useless coefficients in the representation of the polynomial. By convention the null polynomial will have a degree equal to -1. Note : the methods are roughly sorted by difficulty.


```
class Poly {
  double *coeff;
  int degre;
public:
  // default constructor
  Poly();
  // built the polynomial from the degre and a list of coefficients
  Poly(int d, double *c);
  // copy constructor
  Poly(const Poly &p);
  // construct ONE polynomial equal to c*X^k
  Poly(double c, int k);
  // To convert a double to a polynomial of degre 0
  Poly(double x);
  // Destructor
  ~Poly();

  Poly &operator = (const Poly &p);
  bool operator == (const Poly &p) const;

  void print();
  Poly derivative() const;
  Poly operator * (const Poly &p) const;
  Poly operator + (const Poly &p) const;
};
```

So that we can execute (for example) the following `main()` :

```
int main() {
  // We initialize P to 5*X^3 + 1
  double x[] = {1, 0, 0, 5};
  Poly p(3, x);
  p.print();

  Poly q = p.derivative();
  p.print();

  // We use here the *, the + and the implicit conversion from double
  Poly r = p * q + (p + 2.0);
  r.print();
}
```

Chapter 19

Mid-term preparation

19.1 Variables, types, scope, default initialization

A variable is a small area of memory which is associated to an **identifier** and a **type**. The **scope** of a variable (or other identifier) is the area of the source code where the variable can be referred to, most of the time the part between the declaration of the variable and the end of the smallest enclosing `{}` block. Note that a variable is **not initialized by default**.

```
#include <iostream>

int main(int argc, char **argv) {
  int a;
  a = a+1;                  // ouch!
  int b = 3;                // good
  if(b == 3) { int b = 5; int c = 4; } // ouch!
  cout << "b=" << b << '\n'; // here b = 3
  cout << "c=" << c << '\n'; // here can't compile : out of scope
}
```

19.2 Variables, pointers, dynamic allocation

A **pointer** is an **address in memory**. Its type depends on the type of the variable it refers to. The `*` operator allow to denote not the pointer's value but

the pointed variable's value. The `new` operator allows to create a variable of a given type and to get its address. The `delete` operator (resp. `delete[]`) indicates to the computer a variable (resp. array) located at a given address is not used anymore. A variable created with `new` is called a **dynamic variable**, while a *normal* variable is called **static**. The `[]` operator allow to access either an element in a static or dynamically allocated array.

```
#include <iostream>

double *definitelyStupid() {
  double a[10];
  return a;     // ouch !!! *NEVER* do that!!!
}

int main(int argc, char **argv) {
  double *a, *b;
  a = definitelyStupid();
  delete[] a;                           // ouch!
  b = new double[10];
  for(int i = 1; i<100; i++) b[i] = i; // ouch!
  double *c;
  c[10] = 9.0                           // ouch!
}
```

19.3 Expressions, operators, implicit conversion, precedence

An expression is a sequence of one or more **operands**, and zero or more **operators**, that when combined, produce a value.

Operators are *most of the time* defined for two operands of same type. The compiler can automatically convert a numerical type into another one with no loss of precision, so that the operator exists.

Arithmetic computations can lead to **arithmetic exceptions**, either because the computation can not be done mathematically, or because the used type can not carry the resulting value. In that case the result is either a wrong value or a non-numerical value.

The precedence of operators is the order used to evaluate them during the evaluation of the complete expression. To be compliant with the usual mathematical notations, the evaluation is not left-to-right.

19.4 `if`, `while`, `for`, `while/do`

To repeat part of programs, or execute them only if a given condition is true, the C++ has four main statements :

```
if(condition) { ... }
for(init; condition; iteration) { ... }
while(condition) { ... }
do { ... } while(condition);
```

The main bugs are usage of `=` instead of `==` in test, and never-ending loops.

```
#include <iostream>

int main(int argc, char **argv) {
  int a = 10, b = 20;
  while(a < b) { a = 0; b = 2; } // ouch!
  if(a = 3) { cout << "We have a three!!!!!\n"; } // ouch!
}
```

19.5 Declaring and defining functions

Typical definition contains the type of the value it returns, an **identifier** for its name, and the **list of parameters** with their types. The **return** keyword allows to return the result of the function. The evaluation is done when the **call operator** `()` is used. One **argument** is provided to each parameter.

A function, like a variable has a scope, which starts after its **declaration**. The definition can be somewhere else :

```
int product(int a, int b);                // declaration

int square(int a) { return product(a, a); }
int product(int a, int b) { return a*b; } // definition

int main(int argc, char **argv) {
  int a = square(5);
}
```


19.6 Parameters by value or by reference

A parameter can be passed either **by value** or **by reference**. In the first case, the value of the argument at the call point is copied into the parameter. In the second case, the parameter and the value are two different identifiers for the same variable in memory. The copy has to be avoided sometime for performance issue (copying a large object like an array can be expensive).

We will usually make a difference between a **lvalue** (location value, on the left of the `=` operator), and a **rvalue** (reading value, or the right of the `=` operator).

```
#include <iostream>

void reset(int &a) { a = 0; }
void bug(int a) { a = 42; }

int main(int argc, char **argv) {
  int x = 3;
  reset(x);
  cout << x << '\n';
  bug(x);
  cout << x << '\n';
}
```

19.7 Functions, recursion

A function can have a recursive structure, and calls itself. The main bug in that case is to forget the stop criterion.

```
int something(int k) {
  if(k%1 == 0) return something(k+1); // ouch!!!
  else return 2;
}
```

19.8 Algorithm costs, Big-O notation

To estimate the efficiency of an algorithm, the programmer has to be able to estimate the **number of operations** if requires to be executed. Usually the number of operations is estimated as a function of a parameter (like the number of data to work on, or the expected precision of a computation, etc.)

For example :

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

```
for(i = 0; i < n; i++) for(j = 0; j<n*n; j++) { ... }
```

The classical way to denote an approximation of a complexity is to use the $O(.)$ notation (called "big-O").

If n is a parameter and $f(n)$ the exact number of operations required for that value of the parameter, then we will denote $f(n) = O(T(n))$ and say that f is a big-O of T if and only if :

$$\exists c,\ N,\ \ \forall n \geq N,\ f(n) \leq c.T(n)$$

it means that f is **asymptotically bounded** by a function proportional to T.

19.9 Sorting algorithms

Sorting numbers is a very basic tasks one has to do often. We have seen three different algorithms.

1. **Pivot sort**
2. **Fusion sort**
3. **Quick sort**

The *normal* cost for a reasonable sort-algorithm is $O(n \times \log(n))$

19.10 `class` keyword

The main concept in C++ is the concept of **class**. Roughly speaking, a class is a type created by the programmer (opposed to the built-in types like `int`, `double`, etc.)

A class is defined by a name (identifier), data fields (each of them with a name and a type) and methods (each of them with a name a return type and a parameter).

An **object** is an instance of the class, i.e. an entity build from the model the class (like a physical car is an instance of the car described on a plan).

19.11 Constructors / destructor, = operator

The creation and destruction of an object involve special member functions called constructors and destructors. The : operator allow to call constructors for various data fields with no call to default constructors. The **default constructor** is a constructor that does not require parameters. The **copy constructor** is a constructor that take as parameter one instance of the class itself by reference.

The copy constructor is called each time an object has to be created equal to an existing one : definition of a variable with an initial value, or argument passed by value.

The `=` operator (assignment) has to be defined also in most of the case as soon as there are pointers in the data fields.

Note that when the `=` operator is used to specify the initial value of a static variable the compiler calls the copy constructor and not the `=` operator!

19.12 A matrix class

Considering all we have seen so far, we can now build a consistent matrix class :

```
class Matrix {
  int width, height;
  double *data;
public:
  Matrix();
  Matrix(int w, int h);
  Matrix(const Matrix &m);
  ~Matrix();
  bool operator == (const Matrix &m) const;
  Matrix &operator = (const Matrix &m);
  Matrix operator + (const Matrix &m) const;
  Matrix operator * (const Matrix &m) const;
  double &operator () (int i, int j);
  void print() const;
};
```

```
Matrix::Matrix() { width = 0; height = 0; data = 0; }

Matrix::Matrix(int w, int h) {
  cout << "  Matrix::Matrix(int w, int h)\n";
  width = w; height = h;
  data = new double[width * height];
}

Matrix::Matrix(const Matrix &m) {
  cout << "  Matrix::Matrix(const Matrix &m)\n";
  width = m.width; height = m.height;
  data = new double[width * height];
  for(int k = 0; k<width*height; k++) data[k] = m.data[k];
}

Matrix::~Matrix() {
  cout << "  Matrix::~Matrix()\n";
  delete[] data;
}
```

```
Matrix &Matrix::operator = (const Matrix &m) {
  cout << "  Matrix &operator = (const Matrix &m)\n";
  if(&m != this) {
    delete[] data;
    width = m.width; height = m.height;
    data = new double[width * height];
    for(int k = 0; k<width*height; k++) data[k] = m.data[k];
    return *this;
  }
}
```

```
bool Matrix::operator == (const Matrix &m) const {
  cout << "  bool operator == (const Matrix &m) const\n";
  if(width != m.width || height != m.height) return false;
  for(int k = 0; k<width*height; k++) if(data[k] != m.data[k]) return false;
  return true;
}
```

```
Matrix Matrix::operator + (const Matrix &m) const {
  cout << "  Matrix operator + (const Matrix &m) const\n";
  if(width != m.width || height != m.height) {
    cerr << "Size error!\n";
    abort();
  }
```



```
  Matrix result(width, height);
  for(int k = 0; k<width*height; k++) result.data[k] = data[k] + m.data[k];

  return result;
}

Matrix Matrix::operator * (const Matrix &m) const {
  cout << "  Matrix operator * (const Matrix &m) const\n";
  if(width != m.height) {
    cerr << "Size error!\n";
    abort();
  }

  Matrix result(m.width, height);
  for(int i = 0; i<m.width; i++)
    for(int j = 0; j<height; j++) {
      double s = 0;
      for(int k = 0; k<width; k++) s += data[k + j*width] * m.data[i + m.width*k];
      result.data[i + m.width*j] = s;
    }

  return result;
}
```

```
double &Matrix::operator () (int i, int j) {
  cout << "  double & operator () (int i, int j)\n";
  if(i<0 || i>=width || j<0 || j >= height) {
    cerr << "Out of bounds!\n";
    abort();
  }
  return data[i + width*j];
}

void Matrix::print() const {
  cout << "  void print() const\n";
  for(int j = 0; j<height; j++) {
    for(int i = 0; i<width; i++) cout << " " << data[i + width * j];
    cout << "\n";
  }
}
```

```
int main(int argc, char **argv) {
  cout << "DOING Matrix m(3, 2), n(5, 3);\n";
  Matrix m(3, 2), n(5, 3);
```

```
  cout << "DOING Matrix x = m*n;\n";
  Matrix x = m*n;
  cout << "DOING m.print();\n";
  m.print();
  cout << "DOING m = n;\n";
  n = m;
  cout << "DOING n.print();\n";
  n.print();
  cout << "DOING x.print();\n";
  x.print();
}
```

```
DOING Matrix m(3, 2), n(5, 3);
  Matrix::Matrix(int w, int h)
  Matrix::Matrix(int w, int h)
DOING Matrix x = m*n;
  Matrix operator * (const Matrix &m) const
  Matrix::Matrix(int w, int h)
  Matrix::Matrix(const Matrix &m)
  Matrix::~Matrix()
DOING m.print();
  void print() const
 0 0 0
 0 0 0
DOING m = n;
  Matrix &operator = (const Matrix &m)
DOING n.print();
  void print() const
 0 0 0
 0 0 0
DOING x.print();
  void print() const
 0 0 0 0 0
 0 0 0 0 0
  Matrix::~Matrix()
  Matrix::~Matrix()
  Matrix::~Matrix()
```

19.13 Inheritance

A very powerful mechanism of the OO approach consists in extending existing class through the mechanism of inheritance. Basically, it allows you to create a new class by adding members (both data and functions) to an existing class.

And you new class can be used wherever the old one was used.

We call the new class a **subclass** of the old one, which is its **superclass**.

We have to define a new class, which inherits from the first one. We have to define the constructors, which can call the constructors of the initial class. And we can add functions.

```
#include <iostream>

class Student {
  char *name;
  int age;
public:
  Student(char *n, int a) name(n), age(a) { }
  char *getName() { return name; }
  int getAge() { return age; }
};

class ForeignStudent : public Student {
  char *country;
public:
  ForeignStudent(char *n, int a, char *c) : Student(n, a),
                                            country(c) { }
  char *getCountry() { return country; }
};

bool sameAge(Student s1, Student s2) {
  return s1.getAge() == s2.getAge();
}

int main(int argc, char **argv) {
  Student s1("Jack", 21);
  ForeignStudent s2("Steven", 21, "UK");
  bool same = sameAge(s1, s2);
}
```

Chapter 20

Homework

20.1 Introduction

The goal of this homework is to write a class to draw grids of complex numbers in the complex plane. We want to be able to define a mesh of size $n \times n$, to associate to each node a complex number, and then to draw it in a window.

This will allow to represent deformations of the plane associated to complex mappings by drawing the mesh obtained by applying a mapping to a initial "flat" mesh (see figure 20.1).

20.2 A window to draw lines in the complex plane (40 points)

Inherits from the `SimplexWindow` class and create a new class `ComplexWindow` with the following methods (you have to add also some member data fields) :

```
ComplexWindow(int ws, ComplexNumber c, double s);
void clear();
void drawSegment(ComplexNumber a, ComplexNumber b);
```

Where the constructor parameter `ws` is the size of the window (both width and height), `c` is the complex number at the center of the window, and `s` is the width and height in the complex plane.

The `clear` function set the window in white, and the `drawSegment` function draw a black segment between two complex numbers.

20.3 A window to draw a mesh in the complex plane (60 points)

Inherits from `ComplexWindow` and create a class `MeshWindow` with the following methods :

```
MeshWindow(int ws, ComplexNumber center, double scale, int gs);
~MeshWindow();
void setPoint(int i, int j, ComplexNumber z);
ComplexNumber getPoint(int i, int j);
void draw();
```

The three first parameters of the constructor have the same meaning as in `ComplexWindow`, the fourth one indicates how many lines the mesh will have vertically and horizontally.

This class will keep in memory an bi-dimensional array of complex numbers (you can use a simple array and access it with an index of the form $i + width \times j$) to store the complex value for each node.

The `setPoint` and `getPoint` allow to set and read the complex value associated to a given node. The `draw` function clear the window, draw the mesh and display the final image.

Finally we can represent the deformation associated to a given complex mapping by drawing the mesh which node are of the form :

$$\{z = \ k\epsilon + ik'\epsilon \ : \ k, k' \in N, \ |\, re(z)\,| \leq \frac{1}{\sqrt{2}}, \ |\, im(z)\,| \leq \frac{1}{\sqrt{2}} \ \}$$

The $\frac{1}{\sqrt{2}}$ bounds ensure that all the nodes are in the disc of center O and radius 1 (this is nice to prevent the mesh to go too far when we apply exponents).

The `main` function could have the following form to draw the deformed mesh associated to the mapping $z \mapsto z^2$:

```
int main () {
  int size = 50;
```

```
  MeshWindow win(600, 0, 2.1, size);
  for(int i = 0; i<size; i++) for(int j = 0; j<size; j++) {
    ComplexNumber z((i/double(size-1) - 0.5)*sqrt(2),
                    (j/double(size-1) - 0.5)*sqrt(2));
    win.setPoint(i, j, z*z);
  }
  win.draw();
  cin.get();
}
```

Results

Deformations associated to z, $z \times (0.9 + 0.1\,i)$, z^2, z^3, $z + 0.4 \times z^3$ and $\frac{0.2}{z}$ respectively.

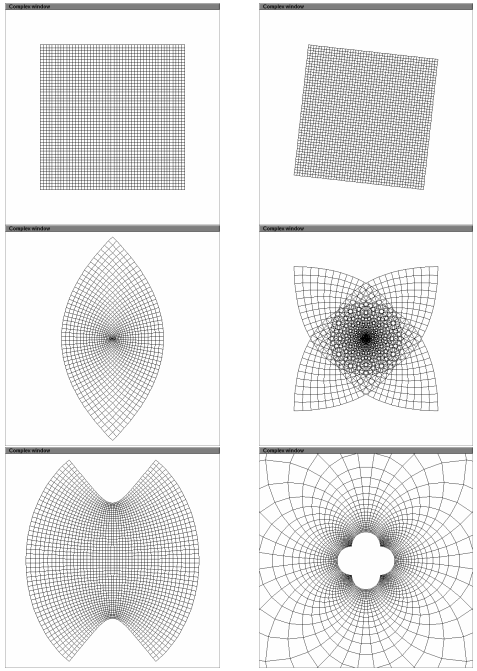


Figure 20.1: Deformation of a regular mesh with analytical complex transformations.

Chapter 21

Virtual methods

21.1 One major weakness of inheritance

As we have seen, the method specified by a given identifier depends on the type of the object it belongs to at the call point :

```
#include <iostream>

class FirstClass {
public:
  void something() { cout << "FirstClass::something()\n"; }
};

class SecondClass : public FirstClass {
public:
  void something() { cout << "SecondClass::something()\n"; }
};

void niceFunction(FirstClass y) { y.something(); }

int main(int argc, char **argv) {
  SecondClass x;
  x.something();   // Here x is of type SecondClass
  niceFunction(x); // In the function it is seen as FirstClass
}
```

prints


```
SecondClass::something()
FirstClass::something()
```

This does the same, even when we pass the object by reference, which can be pretty annoying. Imagine we setup the `<<` operator with a method like this :

```
#include <iostream>

class Employer {
  char *name;
public:
  Employer(char *n) : name(n) {}
  void print(ostream &os) const { os << name; }
};

ostream &operator << (ostream &os, const Employer &e) {
  e.print(os); return os;
}

class RichGuy : public Employer {
  int stocks;
public:
  RichGuy(char *n, int s) : Employer(n), stocks(s) {}
  void print(ostream &os) const {
    Employer::print(os);
    os << " (this guy has " << stocks << " stocks!!)";
  }
};

int main(int argc, char **argv) {
  RichGuy bill("Bill Gates", 1576354987);
  cout << bill << "\n";
}
```

This prints : `Bill Gates`

21.2 Using virtual methods

virtual methods are designed to allow to do such thing. When such a function is called, the computer traces the *real* type of the object and calls the *real* method. To define a method to be virtual, just add `virtual` in front of its declaration. If we just change one line in class `Employer` :

```
// virtual tells the compiler that any call to this function
// has to trace the real type of the object
virtual void print(ostream &os) const { os << name; }
```

the program now prints

```
Bill Gates (this guy has 1576354987 stocks!!)
```

21.3 Precisions about virtual methods

The computer is able to trace the *real type* of an object if it has a pointer to it or a reference to it. So, beware of **arguments passed by value** :

```
#include <iostream>

class AA {
public:
  virtual void something() { cout << "AA:something()\n"; }
};

class BB : public AA {
public:
  void something() { cout << "BB:something()\n"; }
};

void byvalue(AA x) { x.something(); }
void byref(AA &x) { x.something(); }
void bypointer(AA *x) { x->something(); }

int main(int argc, char **argv) {
  BB b;
  byvalue(b);
  byref(b);
  bypointer(&b);
}
```

prints

```
AA:something()
BB:something()
BB:something()
```


21.4 Pure virtual methods

In many situations, we want to be able to define some of the member functions, which will call other ones we do not want to define yet.

We can imagine for instance a function class that would have a member function

```
double derivative(double x, double epsilon)
```

to compute an empirical derivative.

This function would call another method `double eval(double x)`. Even if we do not have this later function yet, we are able to write the first one :

```
double derivative(double x, double e) {
  return (eval(x+e) - eval(x-e))/(2*e);
}
```

The C++ allows to define classes without writing all method, having in mind to write them in the subclasses only. Of course, this is meaningful only because we have the concept of virtual methods.

Such functions are called **pure virtual methods**. To define such a function, just use as a definition `= 0`. Example :

```
#include <iostream>

class Function {
public:
  // This is pure virtual
  virtual double eval(double x) = 0;

  // This calls the pure one
  double derivative(double x, double e) {
    return (eval(x+e) - eval(x-e))/(2*e);
  }
};

class Oscillating : public Function {
  double a, b, c;
public:
  Oscillating(double aa, double bb, double cc) : a(aa), b(bb), c(cc) {}
  double eval(double x) { return a*sin(b*x+c); }
```

```
};

int main(int argc, char *argv) {
  Oscillating f(1, 1, 0);
  cout << f.derivative(0, 0.1) << "\n";
  cout << f.derivative(0, 0.01) << "\n";
  cout << f.derivative(0, 0.001) << "\n";
}
```

```
0.998334
0.999983
1
```

Trying to create an object of a class with virtual methods is meaningless and the compiler is able to trace such attempts :

```
int main(int argc, char *argv) {
  Function f;
  // let's mess the compiler!
  cout << f.eval(3.0) << "\n";
}
```

returns a compilation error :

```
/tmp/chose.cc: In function 'int main(int, char *)':
/tmp/chose.cc:22: cannot declare variable 'f' to be of type 'Function'
/tmp/chose.cc:22:   since the following virtual functions are abstract:
/tmp/chose.cc:6:        double Function::eval(double)
```

21.5 Pure virtual classes

In certain case, we need to design pure abstract classes, which are classes with only pure virtual methods and no data fields.

This is a very powerful way to write down the specifications associated to an abstract object. It also allows to write programs that use such an object without having written the real class behind it.

```
class GraphicalOutput {
public:
```



```
  virtual void drawLine(double x0, double y0, double x1, double y1) = 0;
  virtual void drawText(double x, double y, char *text) = 0;
  virtual void clear() = 0;
};

class InternetFilter {
public:
  virtual bool acceptURL(char *url) = 0;
};

class IntegerSet {
public:
  virtual void add(int k) = 0;
  virtual bool in(int k) = 0;
  virtual bool empty() = 0;
};

class Function {
public:
  virtual double eval(double x) = 0;
  virtual Function *derivative() = 0;
};
```

21.6 Pointers to virtual classes

We have seen that a pointer to a given type can point to any subclass. That is also true for classes with virtual methods. The compiler does not accept to instanciate a class with virtual methods, but it allows to point to an instance of one of the subclasses with a pointer of type "pointer to the super class".

This is consistant : as soon as one of the method is called, the CPU identifies the real type of the object, and jumps to the corresponding method in the subclass.

```
class ReallyVirtual {
public:
  virtual double operation(double x, double y) = 0;
  double twiceTheOperation(double x, double y) {
    return 2.0 * operation(x, y);
  }
};

class NotVirtualAnymore : public ReallyVirtual {
  double k;
```

```
public:
  NotVirtualAnymore(double l) : k(l) {}
  double operation(double x, double y) { return x+k*y; }
};

int main(int argc, char **argv) {
  ReallyVirtual *f = new NotVirtualAnymore(3.5);
  double x = f->twiceTheOperation(4.3, -8.9);
  delete f;
}
```

Playing with the virtual methods, we could even do more fancy things :

```
#include <iostream>

class Function {
public:
  // This is pure virtual
  virtual double eval(double x) = 0;
  // This calls the pure one
  double derivative(double x, double e) {
    cout << "Function::derivative\n";
    return (eval(x+e) - eval(x-e))/(2*e);
  }
  // Let's define a derivative by default
  virtual double derivative(double x) { return derivative(x, 0.001); }
};

class Oscillating : public Function {
  double a, b, c;
public:
  Oscillating(double aa, double bb, double cc) : a(aa), b(bb), c(cc) {}
  double eval(double x) { return a*sin(b*x+c); }
};

class Quadratic : public Function {
  double a, b, c;
public:
  Quadratic(double aa, double bb, double cc) : a(aa), b(bb), c(cc) {}
  double eval(double x) { return c + (b + a*x)*x; }
  double derivative(double x) { return b + 2*a*x; }
};
```

With such a definition, the class `Oscillating` do not overload the `derivative(double)`, and thus when this method is called on one instance of that class, it will finnally

uses the member function of `Function`, which finally uses the empirical computation of the derivative. In the class `Quadratic`, this function is overloaded, and when it is called, it will just use the analytic version it defines.

```
int main(int argc, char *argv) {
  Oscillating f(2, 3, 4);
  cout << f.derivative(2) << "\n";

  Quadratic q(5, 4, 5);
  cout << q.derivative(2) << "\n";
}
```

prints :

```
Function::derivative
-5.03442
24
```

Also, multiple-inheritance would allow to consider a given object as something else easily :

```
class Polynomial {
protected:
  // We suspect we'll need those field in subclasses
  double *coeff; int degree;
public:
  Polynomial(double *c, int d) : coeff(new double[d+1]), degree(d) {
    for(int k = 0; k<=d; k++) coeff[k] = c[k];
  }
  double value(double x) {
    double r = 0;
    for(int k = degree; k>=0; k--) r = coeff[k] + x*r;
    return r;
  }
};

class Function {
public:
  // This is pure virtual
  virtual double eval(double x) = 0;
  // This calls the pure one
  double derivative(double x, double e) {
    return (eval(x+e) - eval(x-e))/(2*e);
```

```
  }
  // Let's define a derivative by default
  virtual double derivative(double x) {
    return derivative(x, 0.001);
  }
};
```

```
class FunctionPoly : public Function, public Polynomial {
public:
  FunctionPoly(double *c, int d) : Polynomial(c, d) {}
  double eval(double x) { return value(x); }
};

// Let's implement analytic derivative now
class FunctionPolyAD : public FunctionPoly {
public:
  FunctionPolyAD(double *c, int d) : FunctionPoly(c, d) {}
  double derivative(double x) {
    double r = 0;
    for(int k = degree; k>=1; k--) r = k*coeff[k] + x*r;
    return r;
  }
};
```

21.7 Non-trivial example

We want to draw graphical objects which can be either primitives (circle and rectangles) or couples of two graphical objects aligned vertically or horizontally.

```
#include <iostream>
#include "swindow.h"

class GraphicalObject {
public:
  virtual int width() = 0;
  virtual int height() = 0;
  virtual void draw(SimpleWindow &win, int x0, int y0) = 0;
};
```

Each graphical object has a size (width and height) and can be drawn at a given location of a window.

First the two kind of primitives :


```
class Circle : public GraphicalObject {
  int r;
public:
  Circle(int rr) : r(rr) {}
  int width() { return 2*r; }
  int height() { return 2*r; }
  void draw(SimpleWindow &win, int x0, int y0) {
    win.color(0.0, 0.0, 0.0);
    win.drawCircle(x0+r, y0+r, r);
  }
};

class Rectangle : public GraphicalObject {
  int w, h;
public:
  Rectangle(int ww, int hh) : w(ww), h(hh) {}
  int width() { return w; }
  int height() { return h; }
  void draw(SimpleWindow &win, int x0, int y0) {
    win.color(0.0, 0.0, 0.0);
    win.drawLine(x0, y0, x0+w, y0);
    win.drawLine(x0+w, y0, x0+w, y0+h);
    win.drawLine(x0+w, y0+h, x0, y0+h);
    win.drawLine(x0, y0+h, x0, y0);
  }
};
```

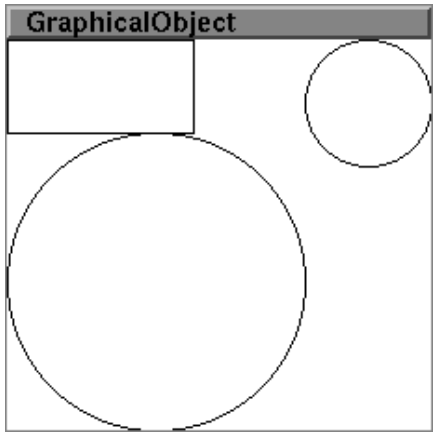
Then, couples. The two objects of a couple can either be aligned vertically or horizontally.

```
class Couple : public GraphicalObject {
  bool vertical;
  GraphicalObject *o1, *o2;
public:
  Couple(bool v,
         GraphicalObject *oo1,
         GraphicalObject *oo2) : vertical(v), o1(oo1), o2(oo2) {}
  ~Couple() { delete o1; delete o2; }

  int max(int a, int b) { if (a>=b) return a; else return b; }

  int width() {
    if(vertical) return max(o1->width(), o2->width());
    else return o1->width() + o2->width();
  }
```

```
  int height() {
    if(vertical) return o1->height() + o2->height();
    else return max(o1->height(), o2->height());
  }

  void draw(SimpleWindow &win, int x0, int y0) {
    o1->draw(win, x0, y0);
    if(vertical) o2->draw(win, x0, y0+o1->height());
    else         o2->draw(win, x0 + o1->width(), y0);
  }
};
```

Here is the result :

```
int main() {
  GraphicalObject *g1 = new Rectangle(100, 50);
  GraphicalObject *g2 = new Circle(80);
  GraphicalObject *g3 = new Couple(true, g1, g2);
  GraphicalObject *g4 = new Circle(34);
  GraphicalObject *g5 = new Couple(false, g3, g4);
  SimpleWindow window("GraphicalObject", g5->width(), g5->height());
  window.color(1.0, 1.0, 1.0);
  window.fill();
  g5->draw(window, 0, 0);
  window.show();
  cin.get();
  delete g5;
}
```

Chapter 22

Boxes and arrows

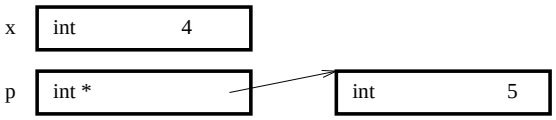
```
int x = 4;
int *p = new int;
*p = 5;
```

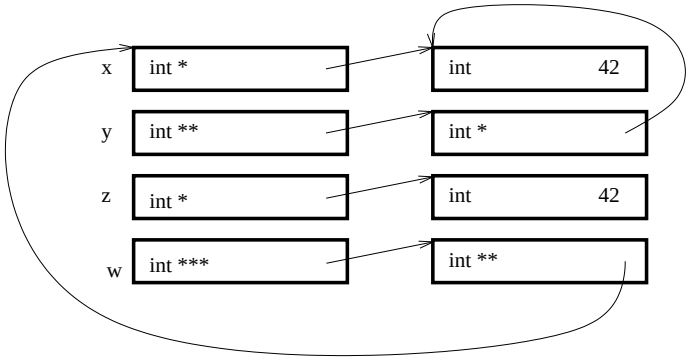
Boxes and arrows !

```
int *x = new int(42);
int **y = new (int *)(x);
int *z = new int(*x);
int ***w = new (int **)(&x)
```

Boxes and arrows !

```
int **x = new (int *)[3];
x[0] = new int[3];
```

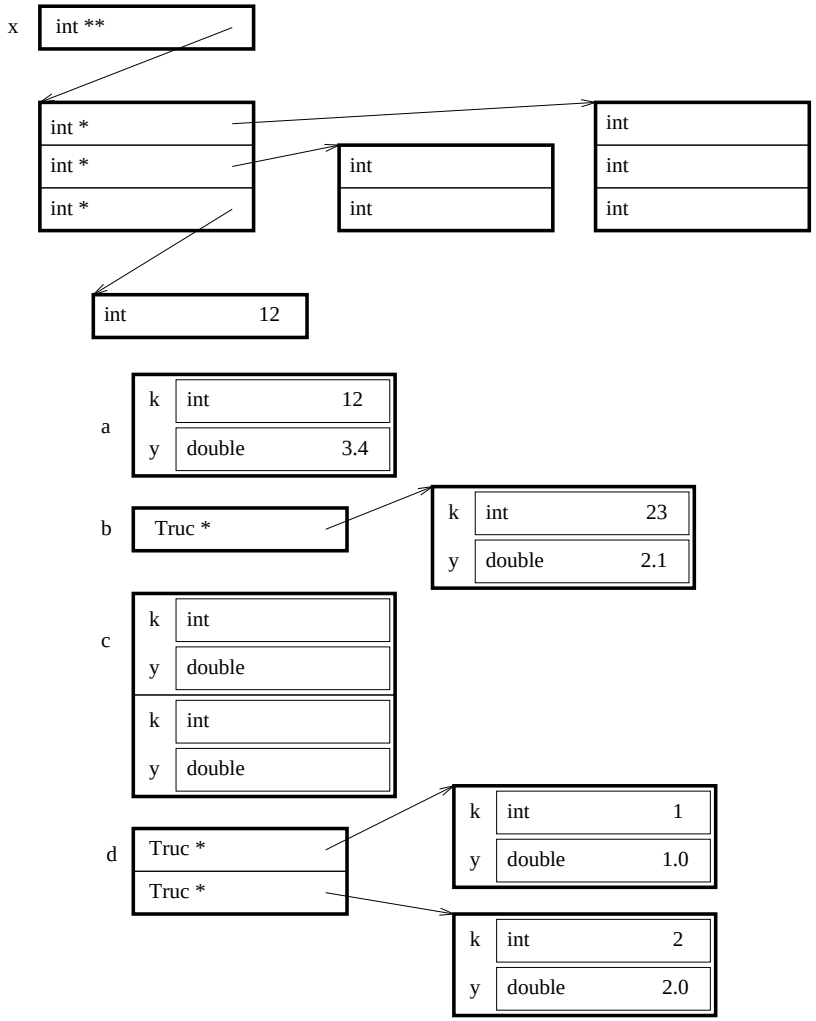





```
x[1] = new int[2];
x[2] = new int(12);
```

Boxes and arrows !

```
class Truc {
  int k;
  double y;
public:
  Truc(int l, double z) : k(l), y(z) {}
}

...

Truc a(12, 3.4);
Truc *b = new Truc(23, 2.1);
Truc c[2];
Truc **d = new (Truc *)[2];
d[0] = new Truc(1, 1.0);
d[1] = new Truc(2, 2.0);
```



Example of virtual classes : mathematical functions

```
#include <iostream>
#include <cmath>

class Function {
public:
  virtual double eval(double x) = 0;
  virtual double derivative(double x) = 0;
};
```

```
class FIdentity : public Function {
public:
  FIdentity() {}
  double eval(double x) { return x; }
  double derivative(double x) { return 1; }
};

class FConst : public Function {
  double k;
public:
  FConst(double l) : k(l) {}
  double eval(double x) { return k; }
  double derivative(double x) { return 0; }
};
```

```
class FSum : public Function {
  Function *f1, *f2;
public:
  FSum(Function *ff1, Function *ff2) : f1(ff1), f2(ff2) {}
  ~FSum() { delete f1; delete f2; }
  double eval(double x) { return f1->eval(x) + f2->eval(x); }
  double derivative(double x) {
    return f1->derivative(x) + f2->derivative(x);
  }
};

class FProd : public Function {
  Function *f1, *f2;
public:
  FProd(Function *ff1, Function *ff2) : f1(ff1), f2(ff2) {}
  ~FProd() { delete f1; delete f2; }
  double eval(double x) { return f1->eval(x) * f2->eval(x); }
```

```
  double derivative(double x) {
    return f1->derivative(x)*f2->eval(x) +
           f1->eval(x)*f2->derivative(x);
  }
};

class FExp : public Function {
  Function *f;
public:
  FExp(Function *ff) : f(ff) {}
  ~FExp() { delete f; }
  double eval(double x) { return exp(f->eval(x)); }
  double derivative(double x) {
    return f->derivative(x)*exp(f->eval(x));
  }
};
```

```
int main(int argc, char **argv) {
  // f(x) = exp(x)
  Function *f = new FExp(new FIdentity());
  cout << f->eval(1.0) << "\n";

  // g(x) = exp(x*x + 2)
  Function *g = new FExp(new FSum(new FProd(new FIdentity(),
                                            new FIdentity()),
                                  new FConst(2.0)));
  cout << g->eval(0.9) << "\n";
}
```

Chapter 23

References and virtual classes

23.1 References to virtual classes

We have seen that even if we can not instantiate a virtual class, we can still have a pointer to its type, which in practice points to an instance of one of its non-virtual subclasses.

Similarly, we can have references to virtual class.

23.2 References, `const` qualifier, and temporary objects

If we pass an object by reference, without the `const` qualifier, the compiler refuse to use temporary objects, which can not be considered as *lvalue*. A parameter passed by reference has to be modifiable.

This is consistent : only *lvalue* can be modified, and a reference can be modified. So if we do not specify it `const`, which would mean we do not expect to be able to modify it, it has to be a *lvalue*.

23.3 Exercises

23.3.1 What does it print ?

```
#include <iostream>

class FirstClass {
public:
  void print1() { cout << "FirstClass::print1()\n"; }
  virtual void print2() { cout << "FirstClass::print2()\n"; }
  void print3() { print1(); print2(); }
};

class SecondClass : public FirstClass {
public:
  void print1() { cout << "SecondClass::print1()\n"; }
  virtual void print2() { cout << "SecondClass::print2()\n"; }
};

class ThirdClass : public SecondClass {
public:
  void print1() { cout << "ThirdClass::print1()\n"; }
  virtual void print2() { cout << "ThirdClass::print2()\n"; }
};

int main(int argc, char **argv) {
  FirstClass x;
  x.print1(); x.print2(); x.print3();
  SecondClass y;
  y.print1(); y.print2(); y.print3();
  ThirdClass z;
  z.print1(); z.print2(); z.print3();
}
```

```
FirstClass::print1()
FirstClass::print2()
FirstClass::print1()
FirstClass::print2()
SecondClass::print1()
SecondClass::print2()
FirstClass::print1()
SecondClass::print2()
ThirdClass::print1()
ThirdClass::print2()
```

```
FirstClass::print1()
ThirdClass::print2()
```

23.3.2 What does it do ?

```
#include <iostream>

class IntegerMapping {
public:
  virtual int maps(int x) const = 0;
};

class Translation : public IntegerMapping {
  int k;
public:
  Translation(int l) : k(l) {}
  int maps(int x) const { return x+k; }
};

class Negate : public IntegerMapping {
public:
  Negate() {}
  int maps(int x) const { return -x; }
};

class Compose : public IntegerMapping {
  IntegerMapping *m1, *m2;
public:
  Compose(IntegerMapping &n1, IntegerMapping &n2) : m1(&n1), m2(&n2) {}
  int maps(int x) const { return m1->maps(m2->maps(x)); }
};

int weird(IntegerMapping *m, int a, int b) { return m->maps(a) * m->maps(b); }

int main(int argc, char **argv) {
  Translation t(5); Negate n;
  Compose c(&t, &n);
  cout << weird(&c, 15, 16) << "\n";
}
```

23.3.3 What does it do ?

We keep the definition of the preceding classes and we add :


```
class VectorMaping {
public:
  virtual int *maps(int *c, int s) const = 0;
};

class Shift : public VectorMaping {
  int k;
public:
  Shift(int l) : k(l) {}
  int *maps(int *c, int s) const {
    int *result = new int[s];
    for(int j = 0; j<s; j++) result[j] = c[(j+k)%s];
    return result;
  }
};

class MetaMaping : public VectorMaping {
  const IntegerMapping *im;
public:
  MetaMaping(const IntegerMapping &m) : im(&m) {}
  int *maps(int *c, int s) const {
    int *result = new int[s];
    for(int k = 0; k<s; k++) result[k] = im->maps(c[k]);
    return result;
  }
};
```

```
void print(const VectorMaping &vm, int *c, int s) {
  int *t = vm.maps(c, s);
  for(int j = 0; j<s; j++) {
    cout << t[j];
    if(j < s-1) cout << " "; else cout << "\n";
  }
  delete[] t;
}

int main(int argc, char **argv) {
  int v[] = { 1, 2, 3, 4 };
  print(Shift(3), v, sizeof(v)/sizeof(int));
  print(MetaMaping(Negate()), v, sizeof(v)/sizeof(int));
}
```

Chapter 24

Homework

24.1 Z-buffer

24.2 Introduction

The main problem to draw 3D objects on a computer screen is to deal with occlusion, i.e. to determinate at each point of the image what object is visible.

We consider the following problem : we define a list of objects localized in the cube $[0, w] \times [0, h] \times [0, \infty[$, we want to draw the projection of those objects on the plane $[0, w] \times [0, h]$. Practically we want to estimate at each pixel of a $w \times h$ window which of the objects is visible.

A very efficient algorithm consist in associating to each pixel of coordinate (x, y) of the window a real value representing the z-coordinate of the element of surface visible at this location so far.

Thus, the algorithm initially fills this **z-buffer** with the value ∞ at each point. Then, each time an object is drawn (sphere or triangle) this buffer is used to estimate for each pixel if the new object is hidden by what have been drawn so far (in that case, nothing is drawn on the window, and the z-buffer remains unchanged), or if it hides what have be drawn (in that case, the pixel color is changed, and the z-buffer has to be modified to the new value of the z-coordinate of the element of surface).

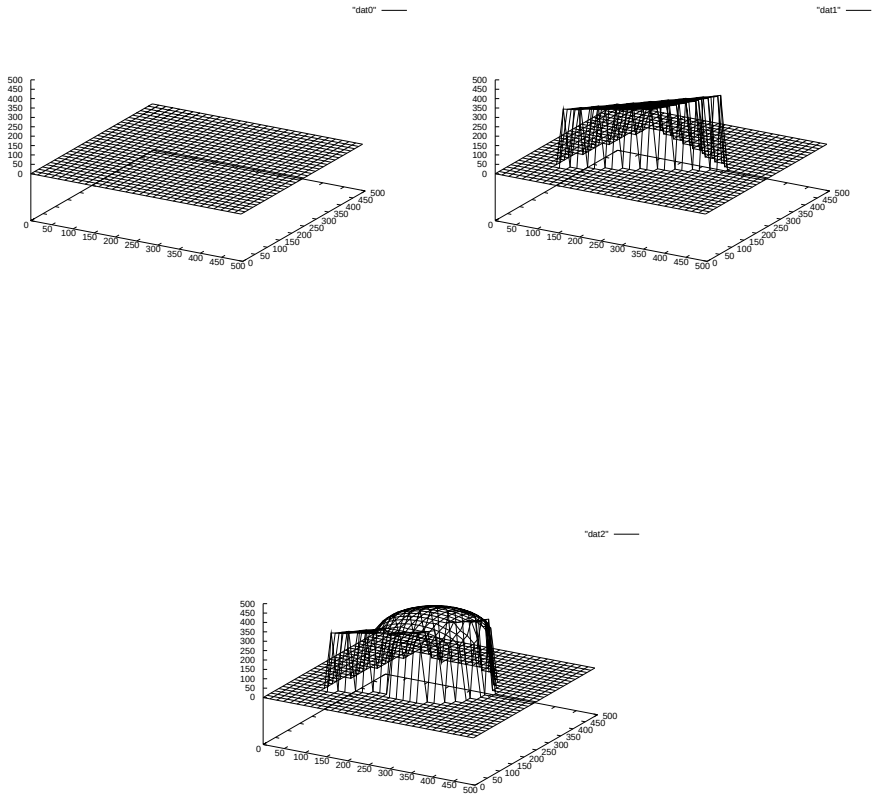
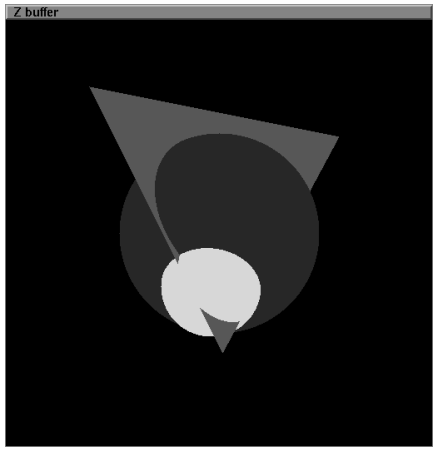


Fig 1 : Z-buffer after initialization, after drawing a triangle, and after drawing a triangle and a ball



24.3 Some math

Let's denote Δ a line of equation $\{(x, y, \lambda) \ : \lambda \in R\}$.

24.3.1 Intersection with a ball

Given a ball B of radius r whose center is at (x_0, y_0, z_0), formulate the constraint on x, y, x_0, y_0, z_0, r so that Δ meets B, and give the formula for the z coordinates of the "first" intersection points (i.e. the one with the smallest z).

24.3.2 Intersection with a triangle

Given a triangle T whose vertices are $(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)$, give the constraints on the variables so that Δ meets T, and give the formula for the z coordinate of the intersection point.

The *easy* way to do that is first to compute the coordinate of the intersection point between Δ and the plan containing the triangle, and then to ensure this point is in the triangle. To ensure the point is in the triangle, you have to check three linear inequations.

Fig 2 : Example of display

24.4 Class to write

By extending the `SimpleWindow` class, you will create a new window class with all required data fields, constructors, destructor, so that it will in particular have the following methods :

```
void clear();
void drawBall(float x0, float y0, float z0, float r);
void drawTriangle(float x1, float y1, float z1,
                  float x2, float y2, float z2,
                  float x3, float y3, float z3);
```

An example of a `main` using such a class would be the following :

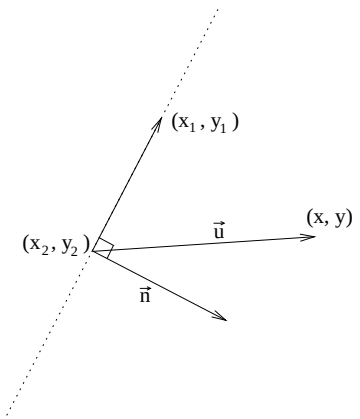
```
int main() {
  ZWindow window(512, 512);

  window.color(1.0, 0.0, 0.0);
  window.drawTriangle(100.0, 80.0, 1000.0,
                      400.0, 140.0, 1200.0,
                      260.0, 400.0, 900.0);
  window.color(0.0, 0.0, 1.0); window.drawBall(256, 256, 1050, 120);
  window.color(1.0, 1.0, 0.0); window.drawBall(246, 320, 970, 60);
  window.show();

  cin.get();
}
```

24.5 Some maths

Belonging to a half-plane

Given two points (x_1, y_1) and (x_2, y_2), it defines a line, which separates the plane into two half-planes. We can estimate to which of the half-planes a point (x, y) belongs to by looking at the sign of the scalar product between the vector $\vec{u} = (x - x_1, y - y_1)$ and the vector $\vec{n} = (y_1 - y_2, x_2 - x_1)$.

$$< \vec{u}, \vec{n} > \ = \ (x - x_1)(y_1 - y_2) \ + \ (y - y_1)(x_2 - x_1)$$



The vector $\vec{n}$ is orthogonal to the line. And the scalar product can be seen as the component of $\vec{u}$ in this orthogonal direction.

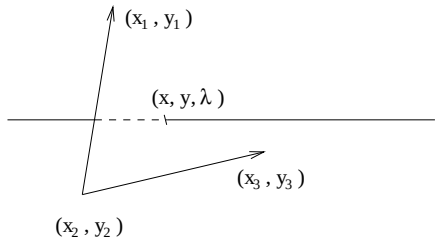
To determine if a point (x, y) is in a triangle whose vertices are $(x_1, y_1), (x_2, y_2), (x_3, y3)$, one has to check three linear inequations. Each of those inequations tests that the point is in a given half-plan.

24.5.1 Intersection between a line and a plane

Given three vectors $(\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2), (\alpha_3, \beta_3, \gamma_3)$, they are in the same plane if and only if the following determinant is null :

$$\begin{vmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \end{vmatrix} = \alpha_1\beta_2\gamma_3 + \alpha_2\beta_3\gamma_1 + \alpha_3\beta_1\gamma_2 - \alpha_3\beta_2\gamma_1 - \alpha_1\beta_3\gamma_2 - \alpha_2\beta_1\gamma_3$$

So, given a line of equation $(x, y, \lambda), \lambda \in R$, one can find its intersection with a plane containing the points (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3) by computing for which value of λ the three vectors $(x_2 - x_1, y_2 - y_1, z_2 - z_1)$, $(x_3 - x_1, y_3 - y_1, z_3 - z_1)$, $(x - x_1, y - y_1, z - z_1)$ are in the same plane.



Chapter 25

Design patterns : sets and iterators

The term "design pattern" describes algorithmic structures that appear very often in many different situations. The idea is to propose an implementation that deal with very abstract objects so that it could be re-use in different cases.

25.1 Example : integer sets and iterators

```
class IntIterator {
public:
  virtual bool hasNext() = 0;
  virtual int next() = 0;
};

class IntSet {
public:
  virtual void add(int k) = 0;
  virtual IntIterator *iterator() const = 0;
};
```

25.2 Exercices

1. Write a function that return the size of a `IntSet` ;

2. propose an implementation of `IntSet` (and of an `IntIterator`) with an array ;
3. propose an implementation of `IntSet` (and of an `IntIterator`) with a linked array.

```
int size(const IntSet &set) {
  int s = 0;
  IntIterator *i = set.iterator();
  while(i->hasNext()) { i->next(); s++; }
  delete i;
  return s;
}
```

```
class IntSetArrayIterator : public IntIterator {
  int *values;
  int current, size;
public:
  IntSetArrayIterator(int *v, int s) : values(v), current(0), size(s) {}
  bool hasNext() { return current < size; }
  int next() { return values[current++]; }
};

class IntSetArray : public IntSet {
  int *values;
  int size, sizemax;
public:
  IntSetArray(int sm) : values(new int[sm]), size(0), sizemax(sm) {}
  ~IntSetArray() { delete values; }
  void add(int k) { if(size >= sizemax) abort(); values[size++] = k; }
  IntIterator *iterator() const { return new IntSetArrayIterator(values, size); }
};
```

```
class Node {
public:
  int value;
  Node *next;
  Node(int v, Node *n) : value(v), next(n) {}
};

class IntSetListIterator : public IntIterator {
  Node *current;
public:
  IntSetListIterator(Node *n) : current(n) {}
```

```
  bool hasNext() { return current; }
  int next() { int r = current->value; current = current->next; return r; }
};

class IntSetList : public IntSet {
  Node *first;
public:
  IntSetList() : first(0) {}
  ~IntSetList() { for(Node *n = first; n; n=n->next) delete n; }
  void add(int k) { first = new Node(k, first); }
  IntIterator *iterator() const { return new IntSetListIterator(first); }
};
```

25.3 Economy of CPU usage : smart copies

In many cases, we can reduce dramatically the number of array copies by using a reference counter scheme. A simple example is a vector class. We want to be able to manipulate vectors, and to do copies only when they are really necessary.

The interface should look like that :

```
  Vector();
  Vector(double *d, int s);
  Vector(const Vector &v);
  ~Vector();
  Vector & operator = (const Vector &v);
  inline double get(int k) const;
  inline void set(int k, double v);
  inline int size() const;
```

We have seen that by using pointer we are able to manipulate arrays, without actually copying them. The main problem is that two different pointers holding the same value are referring to the same object, and thus, modifying one modify the other one.

Thus, we can try to build a vector type with an hidden pointer, so that several copies of the same vector (**as long as it is unmodified**) are in real references to a unique array in memory.

We will introduce a new *hidden* type `InternalVector`, which knows all the time how many `Vector` are referencing it. The `Vector` type will be simply a reference to such a `HiddenVector`.


```
class Vector {
  InternalVector *internal;
public:
  Vector();
  Vector(double *d, int s);
  Vector(const Vector &v);
  ~Vector();
  Vector & operator = (const Vector &v);
  inline double get(int k) const;
  inline void set(int k, double v);
  inline int size() const;
};
```

The `InternalVector` types represent a standard array of `double` but has a field indicatinb how many `Vector` are referencing it. It allows three main operations :

- `release()` indicates that one of the `Vector` that was looking at it is not anymore. This operation will deallocate the `InternalVector` if nobody is looking at it anymore ;
- `grab()` indicates that one more `Vector` is looking at this `InternalVector` ;
- `own()` return a reference to an `InternalVector` containing the same data at `this` but only one reference to it. This will lead to a copy of the object if it has more than one *observer*.

```
class InternalVector {
public:
  double *data;
  int size;
  int nbref;

  InternalVector(double *d, int s) : data(new double[s]),
                                     size(s), nbref(0) {
    cout << "  + Expensive allocation and copy\n";
    for(int k = 0; k<s; k++) data[k] = d[k];
  }

  ~InternalVector() {
    cout << "  + Destruction\n";
    delete[] data;
  }

  void release() {
    if(--nbref == 0) delete this;
```

```
  }

  InternalVector *grab() {
    nbref++;
    return this;
  }

  InternalVector *own() {
    if(nbref == 1) return this;
    else {
      nbref--;
      InternalVector *result = new InternalVector(data, size);
      result->nbref++;
      return result;
    }
  }
};
```

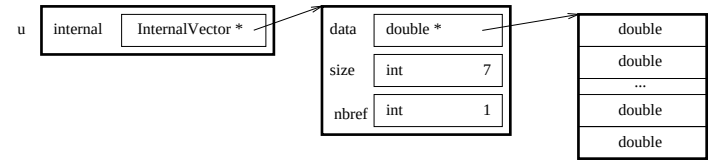
```
Vector::Vector() {
  cout << "  * Creating empty Vector\n";
  internal = 0;
}

Vector::Vector(double *d, int s) {
  cout << "  * Creating Vector\n";
  internal = new InternalVector(d, s);
  internal->grab();
}

Vector::Vector(const Vector &v) {
  cout << "  * Copying Vector\n";
  if(v.internal) internal = v.internal->grab();
  else internal = 0;
}

Vector::~Vector() {
  cout << "  * Destroying Vector\n";
  if(internal) internal->release();
}

Vector & Vector::operator = (const Vector &v) {
  cout << "  * Assigning Vector from Vector\n";
  if(this != &v) {
    if(internal) internal->release();
    internal = v.internal->grab();
```



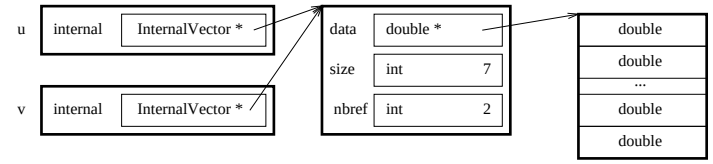
```
  }
  return *this;
}
```

```
inline double Vector::get(int k) const {
  return internal->data[k];
}

inline void Vector::set(int k, double v) {
  if(v != internal->data[k]) {
    internal = internal->own();
    internal->data[k] = v;
  }
}

inline int Vector::size() const {
  return internal->size;
}

double sum(Vector v) {
  cout << "Entering sum()\n";
  double s = 0;
  for(int i = 0; i<v.size(); i++) s += v.get(i);
  cout << "Leaving sum()\n";
  return s;
}
```

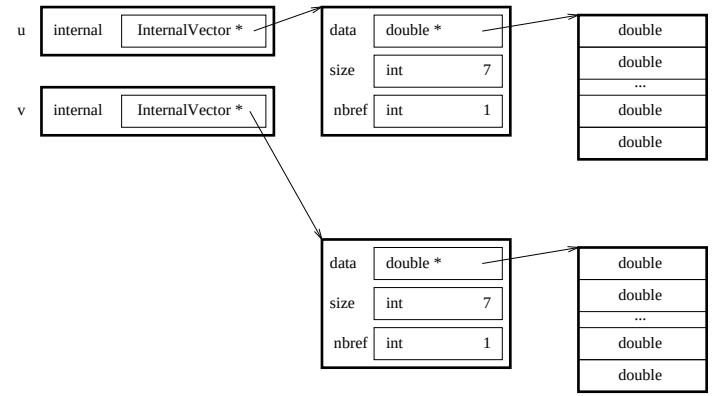
```
double a[] = { 1, 2, 3, 4, 5, 6, 7 };
Vector u(a, sizeof(a)/sizeof(double));
```




```
Vector v = u;
```




```
v.set(3, 4.0)
```




```
int main() {
  cout << "DOING double a[] = { 1, 2, 3, 4, 5, 6, 7 };\n";
  double a[] = { 1, 2, 3, 4, 5, 6, 7 };
  cout << "DOING Vector v(a, sizeof(a)/sizeof(double));\n";
  Vector v(a, sizeof(a)/sizeof(double));
  cout << "DOING Vector w;\n";
  Vector w;
  cout << "DOING w = v;\n";
```



```
  w = v;
  cout << "DOING cout << sum(v) << \"\\n\";\n";
  cout << sum(v) << "\n";
  cout << "DOING w.set(3, 2.1);\n";
  w.set(3, 2.1);
  cout << "FINISHED\n";
}
```

```
DOING double a[] = { 1, 2, 3, 4, 5, 6, 7 };
DOING Vector v(a, sizeof(a)/sizeof(double));
  * Creating Vector
  + Expensive allocation and copy
DOING Vector w;
  * Creating empty Vector
DOING w = v;
  * Assigning Vector from Vector
DOING cout << sum(v) << "\n";
  * Copying Vector
Entering sum()
Leaving sum()
  * Destroying Vector
28
DOING w.set(3, 2.1);
  + Expensive allocation and copy
FINISHED
  * Destroying Vector
  + Destruction
  * Destroying Vector
  + Destruction
```

25.4 Example : back to mappings

We have seen a way to implement mappings with a main virtual class describing the available methods. We can make this description of mappings more sophisticated by adding a formal computation of the derivative. Such an operation would lead to the following specification :

```
class Function {
public:
  virtual double eval(double x) = 0;
  virtual Function *derivative() = 0;
  virtual Function *copy() = 0;
```

```
  virtual void print(ostream &os) = 0;
};

class FConst : public Function {
  double value;
public:
  FConst(double v) : value(v) {}
  double eval(double x) { return value; }
  Function *derivative() { return new FConst(0.0); }
  Function *copy() { return new FConst(value); }
  void print(ostream &os) { os << value; }
};

class FIdentity : public Function {
public:
  FIdentity() {}
  double eval(double x) { return x; }
  Function *derivative() { return new FConst(1.0); }
  Function *copy() { return new FIdentity(); }
  void print(ostream &os) { os << 'X'; }
};

class FSum : public Function {
  Function *f1, *f2;
public:
  FSum(Function *ff1, Function *ff2) : f1(ff1), f2(ff2) {}
  ~FSum() { delete f1; delete f2; }
  double eval(double x) { return f1->eval(x) + f2->eval(x); }
  Function *derivative() { return new FSum(f1->derivative(), f2->derivative()); }
  Function *copy() { return new FSum(f1->copy(), f2->copy()); }
  void print(ostream &os) {
    os << "(";
    f1->print(os);
    os << ") + (";
    f2->print(os);
    os << ")";
  }
};

class FProd : public Function {
  Function *f1, *f2;
public:
  FProd(Function *ff1, Function *ff2) : f1(ff1), f2(ff2) {}
  ~FProd() { delete f1; delete f2; }
  double eval(double x) { return f1->eval(x) * f2->eval(x); }
```



```
  Function *derivative() { return new FSum(new FProd(f1->copy(), f2->derivative())
                                           new FProd(f1->derivative(), f2->copy())
  Function *copy() { return new FProd(f1->copy(), f2->copy()); }
  void print(ostream &os) {
    os << "(";
    f1->print(os);
    os << ") * (";
    f2->print(os);
    os << ")";
  }
};

class FExp : public Function {
  Function *f;
public:
  FExp(Function *ff) : f(ff) {}
  ~FExp() { delete f; }
  double eval(double x) { return exp(f->eval(x)); }
  Function *derivative() { return new FProd(f->derivative(), new FExp(f->copy()));
  Function *copy() { return new FExp(f->copy()); }
  void print(ostream &os) {
    os << "exp("; f->print(os); os << ")";
  }
};

int main(int argc, char **argv) {
  // f(x) = exp(x)
  Function *f = new FExp(new FIdentity());
  Function *df = f->derivative();
  df->print(cout); cout << "\n";

  delete f; delete df;

  // g(x) = exp(x*x + 2)
  Function *g = new FExp(new FSum(new FProd(new FIdentity(),
                                            new FIdentity()),
                                  new FConst(2.0)));
  Function *dg = g->derivative();
  dg->print(cout); cout << "\n";
  delete g; delete dg;
}
```

prints

```
(1) * (exp(X))
((((X) * (1)) + ((1) * (X))) + (0)) * (exp(((X) * (X)) + (2)))
```

25.5 Cast

C++ allows to force the type of a pointer to another one. It can be very useful in certain situations :

```
class Sortable {
public:
  // Will be called only with the same type inside
  virtual bool greaterThan(Sortable *s) = 0;
};

class Integer {
  int k;
public:
  Integer(int kk) : k(kk) { }
  bool greaterThan(Sortable *s) { return k >= ((Integer *) s)->k; }
};

class Couple : public Sortable {
  int a, b;
public:
  Couple(int aa, int bb) : a(aa), b(bb){ }
  bool greaterThan(Sortable *s) { return a >= ((Couple *) s)->a ||
                                         b >= ((Couple *) s)->b; }
};
```

This prevents the compiler from doing type-checking, and allow to write very weird things :

```
int main(int argc, char **argv) {
  Couple c(1, 2);
  Integer x(3);
  bool b = x.greaterThan(&c);
}
```

This piece of code will compile and run with no error or bug, even if it is meaningless. In the same situation, with data structures a bit more complex, it would crash.

25.6 dynamic_cast<type *>

We can keep dynamic type-checking by using the C++ allows to force the type of a “dynamic cast” operator. This operator will return either the pointer with the new type if the cast can be done (i.e. the “real type” of the object is one subtype of the type we try to cast it into) or 0 if not.

```
class Sortable {
public:
  // Will be called only with the same type inside
  virtual bool greaterThan(Sortable *s) = 0;
};

class Integer {
  int k;
public:
  Integer(int kk) : k(kk) { }
  bool greaterThan(Sortable *s) {
    Integer *i = dynamic_cast<Integer *> (s);
    if(i) return k >= i->k;
    else abort();
  }
};

class Couple : public Sortable {
  int a, b;
public:
  Couple(int aa, int bb) : a(aa), b(bb){ }
  bool greaterThan(Sortable *s) {
    Couple *c = dynamic_cast<Couple *> (s);
    if(c) return a >= c->a || b >= c->b;
    else abort();
  }
};
```

25.7 Summary about inheritance

- **Inheritance** allows to add data fields and methods to existing class. All methods of the superclass can be called on one instance of one of the subclass, thus an instance of a subclass can be used anywhere the superclass is expected ;

- when a non-virtual **method** is called, the compiler checks the type of the object at the call point and executes the corresponding method ;
- if a method is **virtual**, the compiler is able to check the "real type" of the object and to call the method of its real class, even if at the call point the object is referenced through one type of one of its superclasses ;
- the compiler allows to define classes without giving the code for some of the virtual methods. Such methods are called **pure virtual**. A class with such a method can not be instantiated. Thus, any pointer of to an object of this type will be in practice a pointer to one an object of one of the subtype with no pure virtual method anymore ;
- the concept of pure virtual is very useful to define abstract object through their specifications instead of defining them with their actual behavior ;
- We can cast a type into one of its superclass type with a dynamic type checking by using the **dynamic cast operator**.

25.8 Weirdness of syntax

25.8.1 Explicit call to the default constructor

The default constructor can not be called with the `()` syntax, it has to be called with no parenthesis at all :

```
class Something {
  int k;
public:
  Something() : k(0) {}
  Something(int l) : k(l) {}
  int get() { return k; }
};

int main(int argc, char **argv) {
  Something x();
  int l = x.get();
}
```

The compiler consider this as a declaration of a function `x`.

```
/tmp/chose.cc: In function ‘int main(int, char **)’:
/tmp/chose.cc:11: request for member ‘get’ in ‘x’, which is of
non-aggregate type ‘Something ()()’
```


25.8.2 Hidden methods

If a subclass has a method with same identifier as a member function in the superclass, even if this function does not have the same parameters, any call has to specify explicitly the superclass :

```
class FirstClass {
public:
  void something() {}
};

class SecondClass : public FirstClass {
public:
  int something(int a, int b, int c) {}
  int anything() { something(); }
};
```

leads to that error :

```
/tmp/chose.cc: In method ‘int SecondClass::anything()’:
/tmp/chose.cc:9: no matching function for call to ‘SecondClass::something ()’
/tmp/chose.cc:8: candidates are: int SecondClass::something(int, int, int)
```

This compiles :

```
class FirstClass {
public:
  void something() {}
};

class SecondClass : public FirstClass {
public:
  int something(int a, int b, int c) {}
  int anything() { FirstClass::something(); }
};
```

Chapter 26

Strings and more iterators

26.1 The `string` class

26.1.1 Introduction

So far, the only way to manipulate character strings is by using direct pointers to arrays of chars. To copy, concatenate, or pass by value, this type is really inefficient.

The standard C++ distribution provides a very powerful type `string`. The underlying structure of this type is an array of char with a reference counter to avoid superfluous copies.

26.1.2 Example

```
#include <string>

int main(int argc, char **argv) {
  string s = "What a beautiful weather!!!";
  string t;
  t = s;
  cout << t << '\n';
}
```


26.1.3 Principal methods and operators

`string()`	constructor
`string(const string &s)`	constructor
`string(const string &s, int pos, int n)`	constructor
`string(const char *s, int size)`	constructor
`string(const char *s)`	constructor
`string(int n, char c)`	constructor
`int length()`	number of characters in the stri
`bool empty()`	is the string empty ?
`char &operator [int k]`	access the n-th character
`string &operator =`	assignment (from other string, c
`string &operator +`	concatenation
`void swap(string &s)`	permutes both strings
`int find(const string &sub, int from)`	find the substring
`string substr(int pos, int length)`	extract a substring
`bool operator == (const string &s1, const string &s2)`	compare two strings

26.1.4 example

```
#include <iostream>
#include <string>

void something(string s) {
  cout << "s = [" << s << "]\n";
  s[0] = 'X';
  cout << "s = [" << s << "]\n";
}

int main(int argc, char **argv) {
  string s1 = "University";
  string s2 = " of ";
  string s3(" Chicago");
  string s4;
  s4 = s1 + s2;
  s4 += s3;
  cout << s4 << "\n";
  s1 = s4.substr(11, 2);
  cout << s1 << "\n";
  something(s1);
  cout << s1 << "\n";
}
```

```
University of  Chicago
of
s = [of]
s = [Xf]
of
```

26.2 Exercises

26.2.1 A small iterator

```
class IntIterator {
public:
  virtual bool hasNext() = 0;
  virtual int next() = 0;
};
```

We want such an iterator that allow to iterate on zero, one or two integers. Thus, we expect those constructors :

```
SmallIterator();
SmallIterator(int aa);
SmallIterator(int aa, int bb);
```

```
class IntIterator {
public:
  virtual bool hasNext() = 0;
  virtual int next() = 0;
};

class SmallIterator : public IntIterator {
  int a, b, n;
public:
  SmallIterator() : n(0) { }
  SmallIterator(int aa) : a(aa), n(1) {}
  SmallIterator(int bb, int aa) : a(aa), b(bb), n(2) {}
  bool hasNext() { return n > 0; }
  int next() { n--; if(n==1) return b; else if(n==0) return a; }
};
```


26.2.2 Write the class

We want to implement an abstract union, that allows to merge two existing `IntegerSet`.

```
class IntIterator {
public:
  virtual bool hasNext() = 0;
  virtual int next() = 0;
};

class IntSet {
public:
  virtual void add(int k) = 0;
  virtual IntIterator *iterator() const = 0;
};

class UnionSet : public IntSet {
public:
  UnionSet(IntSet *ss1, IntSet *ss2);
  ~UnionSet();
  void add(int k); // not needed
  IntIterator *iterator() const;
};
```

```
class IntIterator {
public:
  virtual bool hasNext() = 0;
  virtual int next() = 0;
};

class IntSet {
public:
  virtual void add(int k) = 0;
  virtual IntIterator *iterator() const = 0;
};

class UnionIterator : public IntIterator {
  IntIterator *i1, *i2;
public:
  UnionIterator(IntIterator *ii1, IntIterator *ii2) : i1(ii1), i2(ii2) { }
  bool hasNext() { return i1->hasNext() || i2->hasNext(); }
  int next() {
    if(i1->hasNext()) return i1->next();
```

```
    else i2->next();
  }
};

class UnionSet : public IntSet {
  IntSet *s1, *s2;
public:
  UnionSet(IntSet *ss1, IntSet *ss2) : s1(ss1), s2(ss1) {}
  ~UnionSet() { delete s1; delete s2; }
  // This is not so nice, but this is not the point
  void add(int k) { s2->add(k); }
  IntIterator *iterator() const {
    return new UnionIterator(s1->iterator(), s2->iterator());
  }
};
```

```
int main(int argc, char **argv) {
  IntSetArray *s1, *s2;
  s1 = new IntSetArray(10);
  s2 = new IntSetArray(20);
  for(int k = 0; k<30; k++)
    if(k<10) s1->add(k*8+2); else s2->add(k-5);
  UnionSet *s3 = new UnionSet(s1, s2);
  IntIterator *i = s3->iterator();
  while(i->hasNext()) cout << i->next() << "\n";
  delete i;
}
```

26.2.3 What does it do ?

```
#include <string>

class StuffPrinter {
public:
  virtual void printStuff() = 0;
};

class IterateSP : public StuffPrinter {
  StuffPrinter *single;
  int nb;
public:
  IterateSP(StuffPrinter *sp, int k) : single(sp), nb(k) { }
  ~IterateSP() { delete single; }
  void printStuff() { for(int n = 0; n<nb; n++) single->printStuff(); }
```



```
};

class StringPrinter : public StuffPrinter {
  string s;
public:
  StringPrinter(string t) : s(t) { }
  void printStuff() { cout << s; }
};

int main(int argc, char **argv) {
  StuffPrinter *s1 = new StringPrinter("hi!!!");
  StuffPrinter *s2 = new IterateSP(s1, 10);
  s2->printStuff();
  delete s2;
}
```

26.2.4 Write the class

In the mapping inheritance we had for instance :

```
class FSum : public Function {
  Function *f1, *f2;
public:
  FSum(Function *ff1, Function *ff2) : f1(ff1), f2(ff2) {}
  ~FSum() { delete f1; delete f2; }
  double eval(double x) { return f1->eval(x) + f2->eval(x); }
  Function *derivative() { return new FSum(f1->derivative(), f2->derivative()); }
  Function *copy() { return new FSum(f1->copy(), f2->copy()); }
  void print(ostream &os) {
    os << "(";
    f1->print(os);
    os << ") + (";
    f2->print(os);
    os << ")";
  }
};
```

write a similar class to represent composition.

```
class FCompo : public Function {
  Function *f1, *f2;
public:
```

```
  FCompo(Function *ff1, Function *ff2) : f1(ff1), f2(ff2) {}
  ~FCompo() { delete f1; delete f2; }
  double eval(double x) { return f1->eval(f2->eval(x)); }

  Function *derivative() {
    return new FProd(f2->derivative(),
                     new FCompo(f1->derivative(), f2->copy()));
  }

  Function *copy() { return new FCompo(f1->copy(), f2->copy()); }

  void print(ostream &os) {
    f1->print(os);
    os << " (";
    f2->print(os);
    os << ")";
  }
};
```

Chapter 27

Homework

27.1 Ray-tracing

27.2 Introduction

The goal of this project is to implement a simple version of the well known ray-tracing algorithm. This technique is widely used to generate synthetic pictures and allow to simulate lot of very complex light, reflection and refraction effects (see figure 27.1).

27.3 Description of the algorithm

For the first version, we will not implement reflections or refractions, just visualizing opaque objects.

1. Open a window ;
2. loop thought all pixels :
   (a) compute the associated ray Δ ;
   (b) compute the first intersection with an object of the scene ;
   (c) draw the color ;
3. wait for a key-press.

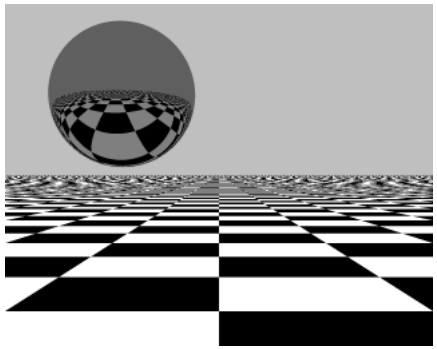


Figure 27.1: Ray-tracing is a simple technique which is able to simulate complex effect of reflexion and refraction.

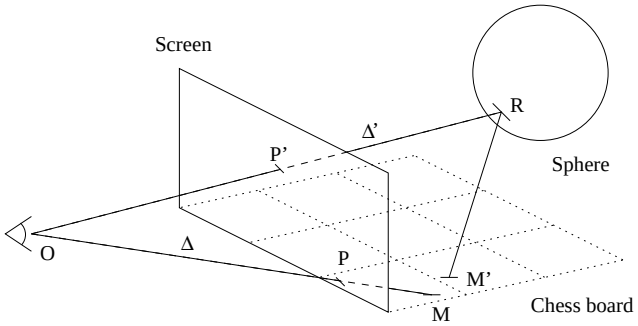


Figure 27.2: The idea of ray-tracing is to associate to each pixel of the screen a virtual *ray* and to compute which objects in the scene intersect this ray.

The objects will have to be either : a sphere of a given color, location and size, or a "infinite chess board", which is horizontal, and is defined by its height, the two colors and the size of the squares.

27.4 Some maths

27.4.1 Parameterization of a ray

A ray is defined by its origin (x_0, y_0, z_0) and its direction (v_x, v_y, v_z). The coordinates of the points that belong to it are of the form $(x_0 + \lambda v_x, y_0 + \lambda v_y, y_0 + \lambda v_y)$ with $\lambda \in R_+$.

Given the location of the observer (x_o, y_o, z_o), and the location of three corners of the screen : upper-left at (x_1, y_1, z_1), lower-left at (x_2, y_2, z_2) and lower-right at (x_3, y_3, z_3), the size of the screen $w \times h$ and the pixel (x_p, y_p), we want to estimate the ray's parameter.

The pixel's P coordinates in the scene (x, y, z) are estimated with linear interpolation. Let's define $\alpha = \frac{x_p}{w}$ and $\beta = 1 - \frac{y_p}{h}$, we have :

$$\begin{cases} x &= x_2 + \alpha(x_3 - x_2) + \beta(x_1 - x_2) \\ y &= y_2 + \alpha(y_3 - y_2) + \beta(y_1 - y_2) \\ z &= z_2 + \alpha(z_3 - z_2) + \beta(z_1 - z_2) \end{cases}$$

Thus, the ray as for origin the observer's location (x_o, y_o, z_o) and for direction $(x - x_o, y - y_o, z - z_o)$.

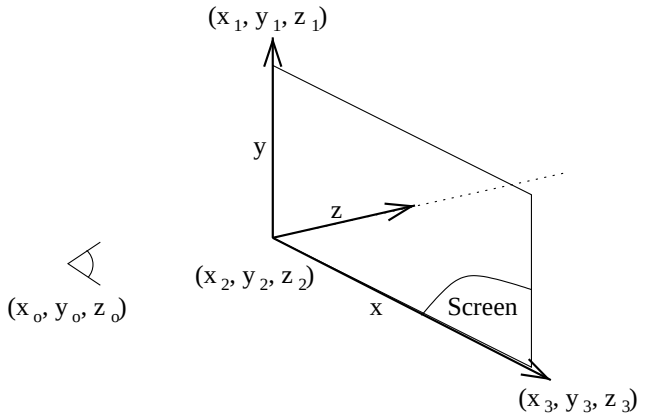
27.4.2 Sphere

A sphere is defined by the location of its center (x_c, y_c, z_c), its radius r and its color. The pixels that belongs to it verify $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$.

A ray has either zero, one or two intersections with a sphere. By substituting the coordinates of the point of the ray into the sphere's equation, we obtain a quadratic equation in λ.

27.4.3 Chessboard

A "infinite" chess board is defined by its height y_{cb} the size of the squares l and two colors c_1 and c_2. A ray meets such an object if its direction goes down



(i.e. $v_y < 0$). In such a case, the coordinates of the intersection points can be estimated by computing λ such that $y_o + \lambda v_y = y_{cb}$. The color of the met point will be c_1 if $sin(\pi \frac{x}{l}) sin(\pi \frac{z}{l}) \geq 0$ and c_2 if not.

27.5 OO organization

The proposed structure is the following :

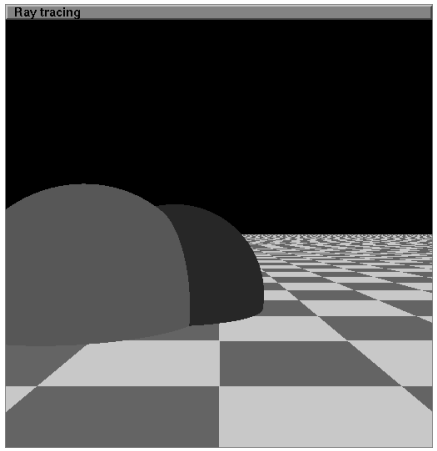
- `Color` represents a r/g/b color ;
- `Ray` represents a ray with an origin and a direction ;
- `Intersection` represents an intersection with an object and indicates both what `Object3D` is met and what is the corresponding λ . This object is able to store a new intersection only if it corresponds to a smaller λ;
- `Object3D` represents the concept of object and has methods to refresh an `Intersection` object, given a `Ray` , and to return the color of the intersection with a given `Ray`;
- `Screen3D` contains the screen size, the position of three screen corners and the observer and can compute a ray, given the coordinate of a pixel ;
- `Scene` is both a window and a `Screen3D` and contains one `Object3D` which represents the main scene.

`Objec3D` will be inherited to create three kind of objects (at least) : spheres, infinite chess board and unions of objects.

27.6 Example of `main`

```
int main(int argc, char **argv) {
  Object3D *p = new Plan(-50,             // y height
                         60,              // square size
                         200, 200, 200,   // r/g/b of color1
                         100, 100, 100    // r/g/b of color2
                         );

  Object3D *s1 = new Sphere(-30, -40, 90, // center's coordinates
                            60,           // radius
                            0, 0, 255     // r/g/b of color
                            );

  Object3D *s2 = new Sphere(-80, -40, 60, // center's coordinates
                            70,           // radius
                            255, 0, 0     // r/g/b of color
                            );

  Object3D *u1 = new Union(s1, s2);
  Object3D *u2 = new Union(u1, p);

  // This Scene class puts the observer at (0, 0, -200) and the three corners
  // of the screen at (-100, 100, 0), (-100, -100, 0) and (100, -100, 0)
  // The window is 512 x 512

  Scene sc(u2);
  sc.drawScene();
  cin.get();
  return 0;
}
```

we obtain :



Chapter 28

Templates

28.1 Introduction

As we have seen, a very important idea in "modern" programming is to be able to re-use already-written code. To be able to do that, we have to write algorithms that operate on abstract objects, defined only by their specifications. For instance :

```
class WithAPrice {
public:
  virtual float price() = 0;
};

int indexOfCheaper(WithAPrice *w, int nb) {
  int best = 0;
  float bestPrice = w[0]->price();
  for(int i = 1; i<nb; i++) if(w[i]->price() < bestPrice) {
    best = i;
    bestPrice = w[i]->price();
  }
}
```

In many situations we want to have the same "generality" for built-in types. For instance, we want to avoid to write :

```
int abs(int x) { if(x >= b) return x; else return -x; }
int abs(float x) { if(x >= b) return x; else return -x; }
```



```
int abs(double x) { if(x >= b) return x; else return -x; }
```

In that case, we would love to write the function once with a "unknown" type and let the compiler create one function for any type we ask it to, as long as the `>=` and unary `-` operators exist. We would love also to be able to define "abstract containers" able to contains any type.

Another issue is efficiency. Addressing generality thought OO mechanisms lead to a severe overload. Instead of just doing the comparison, the CPU would have to jump to a routine etc.

28.2 Examples of template

A **template** is just a piece of code (function or class) parameterized by types (or numerical parameters, but we will not deal with that in this course).

The compiler is able to **instantiate** the piece of code when required for a given type, and is also able to do **type-checking** during compilation :

```
#include <iostream>

template <class Scalar>
Scalar abs(Scalar x) { if(x >= 0) return x; else return -x; }

int main(int argc, char **argv) {
  int x = -3;
  cout << abs(x) << '\n';
  cout << abs(-9.45) << ' ' << abs(7.12) << '\n';
}
```

writes

```
3
9.45 7.12
```

28.3 Syntax

So the syntax is `template < ... list of types ... >` followed by the usual definition of either a function declaration, a function definition, a class definition, a method definition, etc.

You can later either let the compiler instantiate your template code into "real" code implicitly (like what we just did for `abs`), or you can explicitly refer to one instance (to declare a variable or to inherit from a template class for instance).

28.4 Template class

Exactly the same syntax applies to classes :

```
#include <iostream>

template <class Scalar>
class Vector {
  int size;
  Scalar *data;
public:
  Vector(int s) : size(s), data(new Scalar[s]) {}
  ~Vector() { delete data; }
  int length() { return size; }
  Scalar &operator [] (int k) {
    if((k < 0) || (k >= size)) { cerr << "Are you insane ?\n"; abort; }
    return data[k];
  }
  void print(ostream &o) {
    for(int i = 0; i<size; i++) {
      os << data[i];
      if(i <size-1) o << ' '; else o << '\n';
    }
  }
};

int main(int argc, char **argv) {
  Vector<int> v(14);
  Vector<float> u(986);
}
```

28.5 Inheritance from template class

```
class MyVectorOfInt : public Vector<int> {
public:
  MyVectorOfInt(int k) : Vector<int>(k) {}
  int sum() {
    int s = 0;
    for(int i = 0; i<length(); i++) s += (*this)[i];
    return s;
  }
};
```


28.6 Separate definition of methods

As usual, we can separate the class declaration from its definitions :

```
#include <iostream>

template <class Scalar>
class Vector {
  int size;
  Scalar *data;
public:
  Vector(int s);
  ~Vector();
  int length();
  Scalar &operator [] (int k);
  void print(ostream &o);
};
```

```
template<class Scalar>
Vector<Scalar>::Vector(int s) : size(s), data(new Scalar[s]) {}

template<class Scalar>
Vector<Scalar>::~Vector() { delete data; }

template<class Scalar>
int Vector<Scalar>::length() { return size; }

template<class Scalar>
Scalar &Vector<Scalar>::operator [] (int k) {
  if((k < 0) || (k >= size)) { cerr << "Are you insane ?\n"; abort; }
  return data[k];
}

template<class Scalar>
void Vector<Scalar>::print(ostream &o) {
  for(int i = 0; i<size; i++) {
```

```
    os << data[i];
    if(i <size-1) o << ' '; else o << '\n';
  }
}

int main(int argc, char **argv) {
  Vector<int> v(14);
  Vector<float> u(986);
}
```

A template can have more than one parameter.

```
template <class T1, class T2>
class Couple {
  T1 a;
  T2 b;
public:
  void print(ostream &os) { os << a << ' ' << b << '\n'; }
};
```

28.7 Template compilation type-checking

The compiler is able to check the consistency of types for a given template. For instance :

```
#include <iostream>

template <class Scalar>
Scalar max(Scalar a, Scalar b) { if(a >= b) return a; else return b; }

int main(int argc, char **argv) {
  int x = 3;
  float y = 5.0;
  cout << max(x, y) << '\n';
}
```

```
/tmp/chose.cc: In function `int main(int, char **)':
/tmp/chose.cc:9: no matching function for call to `max (int &, float &)'
```

Note that the compiler is not able to mix implicit conversions and argument type deduction in templates. For instance here, it will not convert implicitly the first argument into `float` to be able to instantiate the template.

28.8 Remark about compilation

The behavior of the compiler is exactly as if it was re-writing the piece of code after having substituted the type names. Thus, it may not detect syntax errors (like unknown variable) as long as the piece of code is not used (**this can depend on the compiler**) :

```
template<class T>
T insane(T x) { return y; }

int main(int argc, char **argv) {
}
```

Generates no compilation error, but

```
template<class T>
T insane(T x) { return y; }

int main(int argc, char **argv) {
  insane(3.0);
}
```

leads to :

```
/tmp/chose.cc: In function `double insane<double>(double)':
/tmp/chose.cc:5:   instantiated from here
/tmp/chose.cc:2: `y' undeclared (first use this function)
/tmp/chose.cc:2: (Each undeclared identifier is reported only once
/tmp/chose.cc:2: for each function it appears in.)
```

28.9 Exercise

28.9.1 Write a sum function

Write a function able to compute the sum of n elements of a vector of scalar (`int`, `float`, etc.), whatever the type of elements may be.

```
#include <iostream>
```

```
template<class T>
T sum(T *x, int nb) {
  T s = 0;
  for(int i = 0; i<nb; i++) s += x[i];
  return s;
}

int main(int argc, char **argv) {
  int a[] = {1, 2, 3, 4, 5, 6};
  cout << sum(a, sizeof(a)/sizeof(int)) << '\n';
}
```

28.9.2 Write a template stack class

We want a template class to "stack" elements. The method must allow to insert an object on the top of the stack (push) or to get the object at the top (pop). The constructor will take the maximum number of elements that can be in the stack.

```
#include <iostream>

template<class T>
class Stack {
  int current, max;
  T *dat;
public:
  Stack(int m) : current(0), max(m), dat(new T[m]) {}
  ~Stack() { delete dat; }
  T pop() { if(current == 0) abort(); else return dat[--current]; }
 void push(T x) { if(current == max) abort(); else dat[current++] = x; }
 void print(ostream &o) {
   for(int i = current-1; i >= 0; i--) o << dat[i] << '\n';
 }
};

int main(int argc, char **argv) {
  Stack<int> stack(100);
  stack.push(3);
  stack.push(6);
  stack.push(2);
  stack.push(8); stack.pop();
  stack.push(9);
  stack.print(cout);
}
```

Chapter 29

Tree structures

29.1 Introduction

In many situation, an efficient way to represent data structures is to use trees. A tree can be defined recursively as an object containing some data and references to a certain number of subtrees. This definition leads to a hierarchical structure, in which trees with no sub-trees are called **leaves**. The other ones are called **internal nodes**.

More mathematically, a tree is a graph with no cycles.

Those data structures are very useful to store and organize informations associated to comparable values. Here we give an example of an associative memory `int -> string`.

29.2 A simple implementation

```
class Tree {
  // The key
  int key;
  // The associated string
  string str;
  // The two subtrees
  Tree *left, *right;
public:
  Tree(int *k, string *s, int n);
```



```
  ~Tree();
  int size();
  int depth();
  int nbLeaves();
  string get(int k);
};

int main(int argc, char **argv) {
  string s[] = { "six", "four", "three", "seven",
                 "two", "one", "nine", "five", "eight", "ten" };
  int k[] = { 6, 4, 3, 7, 2, 1, 9, 5, 8, 10 };
  Tree t(k, s, 10);
  cout << t.get(3) << "\n";
}

Tree::Tree(int *k, string *s, int n) {

  for(int i = 0; i<n; i++) cout << k[i] << ":" << s[i]<< " ";
  cout << "\n";

  key = k[0];
  str = s[0];
  int *ktmp = new int[n-1];
  string *stmp = new string[n-1];
  int a = 0, b = n-2;
  for(int i = 1; i<n; i++) if(k[i] < key) {
    ktmp[a] = k[i]; stmp[a] = s[i]; a++;
  } else {
    ktmp[b] = k[i]; stmp[b] = s[i]; b--;
  }
  if(a > 0) left = new Tree(ktmp, stmp, a);
  else left = 0;
  if(b < n-2) right = new Tree(ktmp+a, stmp+a, n-2-b);
  else right = 0;
}

Tree::~Tree() {
  if(left) delete left;
  if(right) delete right;
}

int Tree::size() {
  int n = 1;
  if(left) n += left->size();
```

```
  if(right) n += right->size();
  return n;
}

int Tree::depth() {
  int dl, dr;
  if(left) dl = left->depth(); else dl = 0;
  if(right) dr = right->depth(); else dr = 0;
  if(dl > dr) return dl+1; else return dr+1;
}
```

```
string Tree::get(int k) {
  if(key == k) return str;
  else if(k < key) {
    if(left) return left->get(k);
    else abort();
  } else {
    if(right) return right->get(k);
    else abort();
  }
}
```

```
template<class T>
class Stack {
  int size, maxSize;
  T *data;
public:
  Stack(int m) : size(0), maxSize(m), data(new T[m]) {}
  ~Stack() { delete[] data; }
  T pop() { if(size == 0) abort(); else return data[--size]; }
 void push(T x) { if(size == maxSize) abort(); else data[size++] = x; }
 void print(ostream &o) {
   for(int i = size-1; i >= 0; i--) o << data[i] << '\n';
 }
};

class StringMapping {
public:
  virtual string apply(string s) const = 0;
};
```

```
void Tree::stacksElements(Stack<string> &stack) {
  stack.push(str);
  if(left) left->stacksElements(stack);
```



```
  if(right) right->stacksElements(stack);
}

void Tree::applyMapping(const StringMapping &map) {
  str = map.apply(str);
  if(left) left->applyMapping(map);
  if(right) right->applyMapping(map);
}
```

```
class AddSomething: public StringMapping {
  string stuff;
public:
  AddSomething(string s) { stuff = s; }
  string apply(string s) const { return s + stuff; }
};

int main(int argc, char **argv) {
  string s[] = { "six", "four", "three", "seven",
                 "two", "one", "nine", "five", "eight", "ten" };
  int k[] = { 6, 4, 3, 7, 2, 1, 9, 5, 8, 10 };
  Tree t(k, s, 10);
  cout << t.get(3) << "\n";
  Stack<string> stack(100);
  t.applyMapping(AddSomething(" excellent!!!"));
  t.stacksElements(stack);
  stack.print(cout);
}
```

```
6:six 4:four 3:three 7:seven 2:two 1:one 9:nine 5:five 8:eight 10:ten
4:four 3:three 2:two 1:one 5:five
3:three 2:two 1:one
2:two 1:one
1:one
5:five
10:ten 8:eight 9:nine 7:seven
8:eight 9:nine 7:seven
7:seven
9:nine
three
nine excellent!!!
seven excellent!!!
eight excellent!!!
ten excellent!!!
five excellent!!!
one excellent!!!
```

```
two excellent!!!
three excellent!!!
four excellent!!!
six excellent!!!
```

Chapter 30

Summary of everything

30.1 Variables, types, scope, default initialization

A variable is a small area of memory which is associated to an **identifier** and a **type**. The **scope** of a variable (or other identifier) is the area of the source code where the variable can be referred to, most of the time the part from the variable definition and the end of the smallest enclosing `{}` block. Note that a variable is **not initialized by default**.

```
#include <iostream>

int main(int argc, char **argv) {
  int a;
  a = a+1;                  // ouch!
  int b = 3;                // good
  if(b == 3) { int b = 5; int c = 4; } // ouch!
  cout << "b=" << b << '\n'; // here b = 3
  cout << "c=" << c << '\n'; // here can't compile : out of scope
}
```

30.2 Variables, pointers, dynamic allocation

A pointer is an **address in memory**. Its type depends upon the type of the variable it refers to. The `*` operator allow to denote not the pointer's value

but the pointed variable's value. The `new` operator allows to create a variable of a given type and to get its address. The `delete` operator (resp. `delete[]`) indicates to the computer a variable (resp. array) located at a given address is not used anymore. A variable created with `new` is called a **dynamic variable**, while a *normal* variable is called **static**. The `[]` operator allow to access either an element in a static or dynamically allocated array.

```
#include <iostream>

double *definitelyStupid() {
  double a[10];
  return a;     // ouch !!! *NEVER* do that!!!
}

int main(int argc, char **argv) {
  double *a, *b;
  a = definitelyStupid();
  delete[] a;                           // ouch!
  b = new double[10];
  for(int i = 1; i<100; i++) b[i] = i; // ouch!
  double *c;
  c[10] = 9.0                           // ouch!
}
```

30.3 Expressions, operators, implicit conversion, precedence

An expression is a sequence of one or more **operands**, and zero or more **operators**, that when combined, produce a value.

Operators are *most of the time* defined for two operands of same type. The compiler can automatically convert a numerical type into another one with no loss of precision, so that the operator exists.

Arithmetic computations can lead to **arithmetic exceptions**, either because the computation can not be done mathematically, or because the used type can not carry the resulting value. In that case the result is either a wrong value or a non-numerical value.

The precedence of operators is the order used to evaluate them during the evaluation of the complete expression. To be compliant with the usual mathematical notations, the evaluation is not left-to-right.

30.4 `if`, `while`, `for`, `while/do`

To repeat part of programs, or execute them only if a given condition is true, the C++ has four main statements :

```
if(condition) { ... }
for(init; condition; iteration) { ... }
while(condition) { ... }
do { ... } while(condition);
```

The main bugs are usage of `=` instead of `==` in tests, and never-ending loops.

```
#include <iostream>

int main(int argc, char **argv) {
  int a = 10, b = 20;
  while(a < b) { a = 0; b = 2; } // ouch!
  if(a = 3) { cout << "We have a three!!!!!\n"; } // ouch!
}
```

30.5 Declaring and defining functions

A function definition specifies the type of the value the function returns, an **identifier** for the function's name, and the **list of parameters** with their types. The **return** keyword allows to return the result of the function. The evaluation is done when the **call operator** `()` is used. One **argument** is provided to each parameter.

A function, like a variable has a scope, which starts after its **declaration**. The definition can be somewhere else :

```
int product(int a, int b);                // declaration

int square(int a) { return product(a, a); }
int product(int a, int b) { return a*b; } // definition

int main(int argc, char **argv) {
  int a = square(5);
}
```


30.6 Parameters by value or by reference

A parameter can be passed either **by value** or **by reference**. In the first case, the value of the argument at the call point is copied into the parameter. In the second case, the parameter and the value are two different identifiers for the same variable in memory. The copy has to be avoided sometime for performance issue (copying a large object like an array can be expensive).

We will usually make a difference between a **lvalue** (location value, on the left of the `=` operator), and a **rvalue** (reading value, or the right of the `=` operator).

```
#include <iostream>

void reset(int &a) { a = 0; }
void bug(int a) { a = 42; }

int main(int argc, char **argv) {
  int x = 3;
  reset(x);
  cout << x << '\n';
  bug(x);
  cout << x << '\n';
}
```

30.7 Functions, recursion

A function can have a recursive structure, and calls itself. The main bug in that case is to forget the stop criterion.

```
int something(int k) {
  if(k%1 == 0) return something(k+1); // ouch!!!
  else return 2;
}
```

30.8 Algorithm costs, Big-O notation

To estimate the efficiency of an algorithm, the programmer has to be able to estimate the **number of operations** if requires to be executed. Usually the number of operations is estimated as a function of a parameter (like the number

of data to work on, or the expected precision of a computation, etc.) and is called the **cost** of the algorithm.

For example :

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

```
for(i = 0; i < n; i++) for(j = 0; j < n * n; j++) { ... }
```

The classical way to denote an approximation of a complexity is to use the $O(.)$ notation (called "big-O").

If n is a parameter and $f(n)$ the exact number of operations required for that value of the parameter, then we will denote $f(n) = O(T(n))$ and say that f is a big-O of T if and only if :

$$\exists c,\ N,\ \ \forall n \geq N,\ f(n) \leq c.T(n)$$

it means that f is **asymptotically bounded** by a function proportional to T.

30.9 Sorting algorithms

Sorting numbers is a very basic tasks one has to do often. We have seen three different algorithms.

1. **Pivot sort**
2. **Fusion sort**
3. **Quick sort**

The *normal* cost for a sort-algorithm is $O(n \times \log(n))$

30.10 OO programming

30.11 `class` keyword

The main concept in C++ is the concept of **class**. Roughly speaking, a class is a type created by the programmer (opposed to the built-in types like `int`, `double`, etc.)

A class is defined by a name (identifier), data fields (each of them with a name and a type) and methods (each of them with a name a return type and a parameter).

An **object** is an instance of the class, i.e. an entity build from the model the class (like a physical car is an instance of the car described on a plan). You can for instance define a class standing for the concept of a rectangle, which will contains one field for the width and one for the height, and your program may manipulate several such rectangles, with actual values for those fields.

30.12 Constructors / destructor, = operator

The creation and destruction of an object involve special member functions called constructors and destructors. The : operator allow to call constructors for various data fields with no call to default constructors. The **default constructor** is a constructor that does not require parameters. The **copy constructor** is a constructor that take as parameter one instance of the class itself by reference.

The copy constructor is called each time an object has to be created equal to an existing one : definition of a variable with an initial value, or argument passed by value.

The = operator (assignment) has to be defined also in most of the case as soon as there are pointers in the data fields.

Note that when the = operator is used to specify the initial value of a static variable the compiler calls the copy constructor and not the = operator!

30.13 Inheritance

A very powerful mechanism of the OO approach consists in extending existing class through the mechanism of inheritance. Basically, it allows you to create a new class by adding members (both data and functions) to an existing class. And you new class can be used wherever the old one was used.

We call the new class a **subclass** of the old one, which is its **superclass**.

We have to define a new class, which inherits from the first one. We have to define the constructors, which can call the constructors of the initial class. And

we can add functions.

30.14 virtual methods and classes

- When a non-virtual **method** is called, the compiler checks the type of the object at the call point and executes the corresponding method ;
- if a method is **virtual**, the compiler is able to check the "real type" of the object and to call the method of its real class, even if at the call point the object is referenced through one type of one of its super-classes ;
- the compiler allows to define classes without giving the code for some of the virtual methods. Such methods are called **pure virtual**. A class with such a method can not be instantiated. Thus, any pointer of to an object of this type will be in practice a pointer to one an object of one of the subtype with no pure virtual method anymore ;
- the concept of pure virtual is very useful to define abstract object through their specifications instead of defining them with their actual behavior ;
- We can cast a type into one of its superclass type with a dynamic type checking by using the **dynamic cast operator**.

30.15 Exercises

- Write an abstract pure virtual class `Picture` that has just methods to get its width, height and to access its gray-scale pixels ;
- Write a class `RealPicture` which inherits from the preceding. It contains a width, a height, an array of `float`, a methods to rotate it clockwise or counter-clockwise, depending on a parameter, and the standard constructors ;
- extends the preceding class to add a way to apply any transformation to individual pixels ;
- write a new subclass of `Picture` that allow to manipulate a sub-picture of a `RealPicture`.

```
class Picture {
public:
  virtual int getWidth() = 0;
  virtual int getHeight() = 0;
  virtual float &pixel(int x, int y) = 0;
```



```
};

class RealPicture : public Picture {
  int width, height;
  float *dat;
public:
  RealPicture(int w, int h) : width(w), height(h), dat(new float[w*h]) {}
  RealPicture(const RealPicture &p) : dat(new float[p.width*p.height]),
                                      width(p.width), height(p.height) {
    for(int k = 0; k<width*height; k++) dat[k] = p.dat[k];
  }
  ~RealPicture() { delete[] dat; }
  int getWidth() { return width; }
  int getHeight() { return height; }
  float &pixel(int x, int y) { return dat[x + y*width]; }
  void rotate() {
    float *tmp = new float[width*height];
    for(int x = 0; x<width; x++)
      for(int y = 0; y<height; y++) tmp[y + height*x] = dat[(width-x) + width*y];
    int t = width; width = height; height = t;
    delete[] dat;
    dat = tmp;
  }
};
```

```
class SubPicture : public Picture {
  int deltax, deltay;
  int width, height;
  Picture *pic;
public:
  SubPicture(Picture &p, int dx, int dy, int w, int h) :
     pic(&p), deltax(dx), deltay(dy), width(w), height(h) {}
  int getWidth() { return width; }
  int getHeight() { return height; }
  float &pixel(int x, int y) {  return pic->pixel(x+deltax, y+deltay); }
};
```

```
class MulPicture : public Picture {
  int nx, ny;
  Picture *pic;
public:
  SubPicture(Picture &p, int nnx, int nny) : pic(&p), nx(nnx), ny(nny) {}
  int getWidth() { return pic->width * nx; }
  int getHeight() { return pic->height * ny; }
  float &pixel(int x, int y) { return pic->pixel(x%pic->width, y%pic->height); }
```

```
};
```

```
class PixelMaping {
public:
  virtual float maps(float x) const = 0;
};

class NewRealPicture : public RealPicture {
public:
  NewRealPicture(int w, int h) : RealPicture(w, h) {}
  NewRealPicture(const RealPicture &p) : RealPicture(p) {}
  void applies(const PixelMaping &pm) {
    for(int y = 0; y<getHeight(); y++) for(int x = 0; x<getWidth(); x++)
      pixel(x, y) = pm.maps(pixel(x, y));
  }
};
```

Appendix A

Midterm Exam

A.1 Cost (15 points)

Give a big-O estimation of the number of calls to the function `something(int, int)` in the following function :

```
int anything(int n) {
  for(int k = 0; k<n; k++) {
    something(k, k);
    for(int l = k; l<k+10; l++) something(k, l);
    for(int l = 0; l<k; l++) something(k, l);
  }
}
```

A.2 Some boxes and arrows! (15 points)

Draw a box-and-arrow figure for the memory configuration after those three lines have been executed:

```
double x = 5;
double *p = &x;
double *z = new double(*p);
```


A.3 Find the bug!!! (25 points)

Assuming that `main` is correct, find the three bugs in `sumByColumns`, and propose a correction :

```
#include <iostream>

double *sumByColumns(double *coeff, int w, int h) {
  double result[w];
  for(int i = 1; i<=w; i++) {
    result[i] = 0.0;
    for(j = 0; j<h; j++) result[i] += coeff[i + j*w];
  }
  return result;
}

int main(int argc, char **argv) {
  int w, h;

  cin >> w >> h;
  double *c = new double[w*h];
  for(int j=0; j<h; j++) for(int i=0; i<w; i++)
    cin >> c[i + w*j];

  double *sums = sumByColumns(c, w, h);
  for(int i=0; i<w; i++) cout << sums[i] << '\n';
  delete[] sums;

  delete[] c;
}
```

A.4 What does it print ? (25 points)

Give the result printed by the following program :

```
#include <iostream>

class PairOfInteger {
  int a, b;
public:
  PairOfInteger() { cout << "#1\n"; }
```

```
  PairOfInteger(const PairOfInteger &p) {
    a = p.a; b = p.b;
    cout << "#2\n";
  }

  PairOfInteger(int aa, int bb) {
    a = aa; b = bb;
    cout << "#3\n";
  }

  PairOfInteger &operator = (int x) {
    a = x; b = 0; cout << "#4\n";
    return *this;
  }

  void print(ostream &os) { os << a << ',' << b << '\n'; }
};

int main(int argc, char **argv) {
  PairOfInteger p1(1, 2);
  PairOfInteger p2, p3 = p1;
  p2 = 3; p1 = p2;
  p1.print(cout);
  p2.print(cout);
}
```

A.5 Class design (20 points)

We want to manipulate in a program a database of stocks. Each stock has a name and a number of units in it. We write this class :

```
class Stock {
  char *productName;
  int nbUnits;
public:
  Stock(char *pn, int nu) { productName = pn; nbUnits = nu; }
  bool inStock() { return nu > 0; }
}
```

We want to manipulate in the same program stocks of food which have an expiration date. For each of those stock we need a field for the expiration date (we consider a date can be encoded with an integer), and we need a way to read

this field. Also we'll have a new `bool inStock(int currentDate)` function which takes into account both the number of units and the expiration date.

Propose a `FoodStock` class, so that an object of this class can be used wherever `Stock` was used.

Appendix B

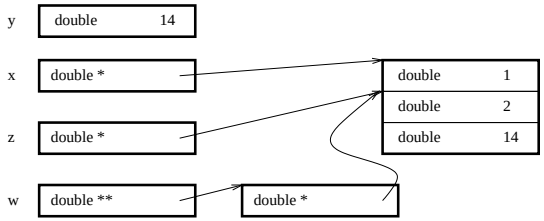
Final Exam

B.1 Some boxes and arrows (15 points)

Draw a box-and-arrow figure for the memory configuration after those three lines have been executed. Each box will contain the type, and the value when it is defined. Boxes representing static variables will have the identifier written on the left.

```
double y = 14;
double *x = new double[3];
x[0] = 1; x[1] = 2; x[2] = y;
double *z = x+1;
double **w = new (double *)(z);
```

Solution



B.2 What does it print ? (25 points)

Give the result printed by the following program, and two lines of explanations for each printed line.

```
#include <iostream>

class AnInteger {
public:
  int value;
  AnInteger(int k) : value(k) {}
  int functionA() { return value; }
  virtual int functionB() { return value; }
  int functionC() { return functionA() - functionB(); }
};

class TwoIntegers : public AnInteger {
public:
  int value2;
  TwoIntegers(int k, int l) : AnInteger(k), value2(l) {}
  int functionA() { return value + value2; }
  int functionB() { return value + value2; }
};

int main(int argc, char **argv) {
  TwoIntegers j(12, 13);
  AnInteger *k = &j;
  cout << j.functionC() << '\n';
  cout << k->functionA() << '\n';
  cout << k->functionB() << '\n';
}
```

Solution

The line 23 calls `j.functionC()`, which is defined in `AnInteger`. In that function the pointer `this` is of type `AnInteger`, even if `j` is in reality of type `TwoInteger`. Because `functionA` is not virtual, it's finally `AnInteger::functionA` which is called, and because `functionB` is virtual, `TwoInteger::functionB` is called. Finally the result is $12 - (12 + 13) = -13$, and `-13` is printed.

The line 24 is a call to a non-virtual method with a pointer of type `AnInteger *`, thus `AnInteger::functionA` is called, and `12` is printed.

The line 25 is a call to a virtual method with a pointer of type `AnInteger *` on an object of type `TwoInteger`, thus `TwoInteger::functionB` is called, and `25` is printed.

B.3 Class design (25 points)

We propose the following class to store a vector of couples of `float`s. Replace the various `[ ... ]` by the required pieces of code.

The `sumAbsValue` has to return the sum of the absolute values of the differences between the terms of the couples $\sum_i |a_i - b_i|$ (for instance, if there are the three couples $(7,3),(-2,5),(0,20)$ it will return $4+7+20=31$).

```
class VectorOfCouples {
  float *values;
  int nbCouples;
public:
  VectorOfCouples(int nb) [ ... ]
  VectorOfCouples(const VectorOfCouples &vc) [ ... ]
  ~VectorOfCouples() [ ... ]
  float &first(int k) {
    if(k<0 || k >= nbCouples) abort(); return values[k*2];
  }
  float &second(int k) {
    if(k<0 || k >= nbCouples) abort(); return values[k*2+1];
  }
  float sumAbs() [ ... ]
};
```

Solution

```
class VectorOfCouples {
  float *values;
  int nbCouples;
public:

  VectorOfCouples(int nb) : values(new float[nb*2]), nbCouples(nb) {}

  VectorOfCouples(const VectorOfCouples &vc) :
    values(new float[vc.nbCouples*2]), nbCouples(vc.nbCouples) {
    for(int i = 0; i<nbCouples*2; i++) values[i] = vc.values[i];
  }

  ~VectorOfCouples() { delete[] values; }

  float &first(int k) {
    if(k<0 || k >= nbCouples) abort(); return values[k*2];
  }

  float &second(int k) {
    if(k<0 || k >= nbCouples) abort(); return values[k*2+1];
  }

  float sumAbs() {
    float s = 0.0;
    for(int i = 0; i<nbCouples; i++) s += abs(first(i) - second(i));
    return s;
  }
};
```


B.4 Virtual class design (35 points)

We want to write a method `sumMap` in the preceding class, that returns the sum of a mathematical function of each couple.

Write a class to define the concept of "mathematical function of a couple". Write `sumMap`, and the classes to represent the mathematical functions $abs(a,b) = |a-b|$ and $prod(a,b) = a \times b$, and **one** line to compute the sum of the products $\sum_i a_i \times b_i$.

Solution

The class to define the concept of mapping, and the two subclasses for absolute value and product :

```
class CoupleMapping {
public:
  virtual float maps(float a, float b) const = 0;
};

class CoupleAbs : public CoupleMapping {
public:
```

```
  float maps(float a, float b) const { return abs(a-b); }
};

class CoupleProduct : public CoupleMapping {
public:
  float maps(float a, float b) const { return a*b; }
};
```

The sumMap method :

```
float VectorOfCouples::sumMap(const CoupleMapping &cp) {
  float s = 0.0;
  for(int i = 0; i<nbCouples; i++) s += cp.maps(first(i), second(i));
  return s;
}
```

and the computation of the sum of the products :

```
vc.sumMap(CoupleProduct());
```

Index

!= operator, 27
% operator, 24
& operator, 33
() operator, 32
& operator, 39
* operator, 39, 40
+ operator, 43
++ operator, 27
-- operator, 28
-> operator, 86
< operator, 27
<= operator, 27
== operator, 27
> operator, 27
>= operator, 27
[] operator, 37, 38, 41, 44

abort(), 35
address, 1, 171, 255
address-of operator, 39
algorithm
    sort, 259
allocation
    dynamic, 45
argc, 42
argument, 32–34, 173, 257
    by reference, 34, 174
    by value, 33, 174, 187
argv, 42
arithmetic exception, 25, 172, 256
arithmetic expression, 23
array
    of char, 37
assembler, 3
assignment operator, 27
asymptotically bounded, 259

big-O notation, 259
bit, 1
bool, 15
boolean expression, 26
boolean operator, 26
break statement, 51
BSD license, 11
bug, 55
built-in type, 14
by reference, 174, 258
by value, 33, 174, 187, 258
byte, 1

cache memory, 4
call operator, 32, 173, 257
cast, 223
    dynamic, 224
central processing unit, 3
char
    array of, 37
char, 15
class, 7, 92, 175, 259
    definition, 85
    derived, 144
    instance, 260
class, 85
class type, 14
comment, 61
compilation
    conditional, 62
conditional compilation, 62
const statement, 50
constant
    literal, 16
constructor, 97, 102
    copy, 176
    default, 176

`continue` statement, 20
conversion
  implicit, 24
copy constructor, 176, 260
cost, 76, 259
current directory, 12

data field, 92, 102, 144
deallocate, 47
declaration, 49, 173, 257
decrement operator, 27
default constructor, 176, 260
definition, 49
`delete` operator, 172
`delete[]` operator, 172
dereference operator, 40
derived class, 144
destructor, 99, 102
digits (number of), 133
directory, 3
  current, 12
  parent, 11
  root, 12
`do/while` statement, 19
`double`, 15
dynamic allocation, 44, 45, 256
dynamic cast, 224
dynamic cast operator, 225, 261
dynamic variable, 172

emacs, 5, 14
`enum`, 51
exception
  arithmetic, 25, 172
expression, 23
  arithmetic, 23
  boolean, 26
  graph of, 29

field, 85
  public, 85
file, 3
  include, 16
  object, 5
  source, 5
filename, 12
`float`, 15
folder, 3
`for` statement, 18
functio
  main, 42
function, 31
  declaration, 49
  definition, 49
  recursive, 34
fusion sort, 81, 175, 259

g++, 5
gdb, 64
GPL, 11
grammar, 29

hard disk, 3
heap, 49
hexadecimal, 2

identifier, 15, 102, 171, 173, 255, 257
  good, 60
`if` statement, 17
include, 16
increment operator, 27
indentation, 61
`inf`, 26
inheritance, 144, 224
  multiple, 136
instance (of a class), 260
instanciation, 92
instantiate, 242
`int`, 15

kernel, 9

leak (memory), 44
leaves, 249
license
  BSD, 11
  GPL, 11
linked list, 121
Linux, 9
literal constant, 16
lvalue, 30, 174, 258

main function, 42

mandelbrot set, 141
member
  private, 92
  public, 92
member operator, 102
memory
  cache, 4
  leak, 44, 58
memory allocation
  dynamic, 44
memory leak, 44, 58
method, 7, 91–93, 102, 144, 225, 261
  pure virtual, 188, 225
  virtual, 186, 225
multiple inheritance, 136

`nan`, 26
`new` operator, 172
nodes, 249
number (format of), 133
number of operations, 174, 258

object, 92, 102, 176, 260
object file, 5
object-oriented, 7
object-oriented programming, 7
OOP, 7
open-source, 9
operand, 23, 172, 256
operator, 23, 117, 172, 256
  `!=`, 27
  `&`, 33, 39
  `*`, 39, 40
  `++`, 27
  `--`, 28
  `<`, 27
  `<=`, 27
  `==`, 27
  `>`, 27
  `>=`, 27
  `[]`, 37, 38, 41, 44
  `%`, 24
  address-of, 39
  arithmetic, 23
  assignment, 27
  boolean, 26
  call, 32, 173
  decrement, 27
  `delete`, 172
  `delete[]`, 172
  dereference, 40
  dynamic cast, 225
  increment, 27
  member, 102
  `new`, 172
  precedence, 28
operator `->`, 86
operator `sizeof`, 17

parameter, 33, 173
parameters, 257
parent directory, 11
path, 11
pivot sort, 80, 175, 259
pointer, 39, 171
  addition, 43
pointer to pointer, 39
post-increment, 27
pre-increment, 27
precedence (of operators), 28
private, 102
private member, 92
protected, 149
public, 102
public field, 85
public member, 92
pure virtual method, 188, 225, 261

quantification, 2
quick sort, 175, 259

ray-tracing, 235
recursive function, 34
reference, 39
return, 173, 257
root directory, 12
rvalue, 30, 174, 258

scope (of a variable), 31
scope of a variable, 171, 255
shell, 12
  commands, 12

`sizeof` operator, 17
sort, 259
    fusion, 81, 175
    pivot, 80, 175
    quick, 175
source
    open, 9
source file, 5
stack, 48
statement, 18
    `break`, 51
    `continue`, 20
    `do/while`, 19
    `for`, 18
    `if`, 17
    `while`, 19
static variable, 45, 172, 256
stopping condition, 35
subclass, 134, 180, 260
superclass, 134, 180, 260
`switch/case`, 20
symmetry, 61

template, 242
tree, 249–253
type, 2, 102, 171, 255
    `float`, 15
    bool, 15
    built-in, 14
    casting, 223
    char, 15
    class, 14
    double, 15
    int, 15
    `unsigned`, 15
    `void`, 32
type casting, 223
type-checking, 242

`unsigned`, 15

variable, 15
    constant, 50
    dynamic, 172
    scope of, 31, 171
    static, 45, 172
virtual method, 186, 225, 261
`void`, 32

`while` statement, 19

X-Window, 9

z-buffer, 207