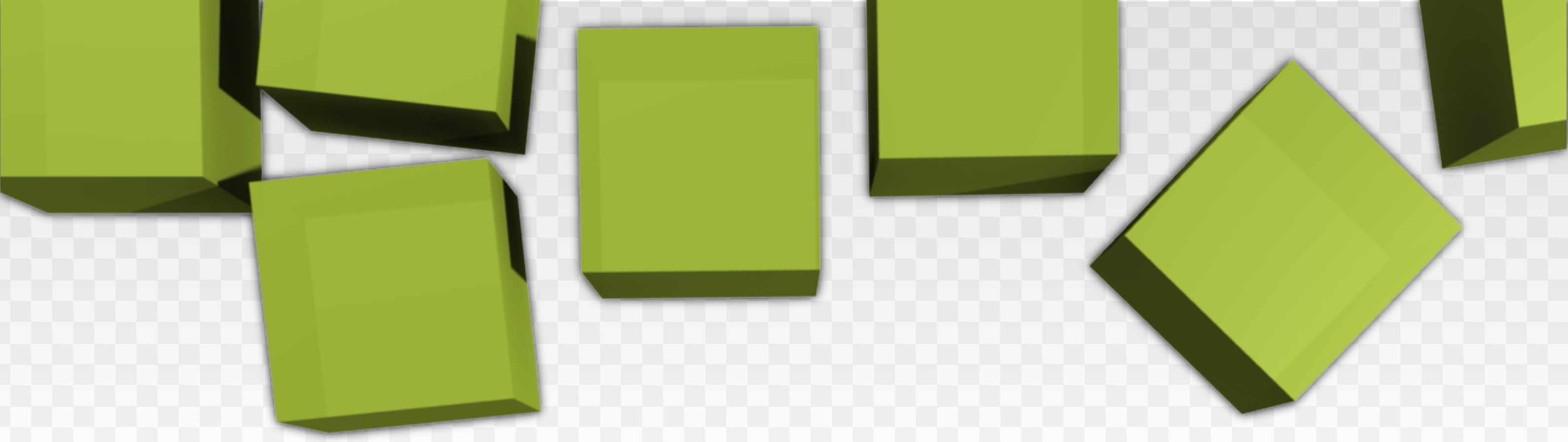


Introduction to Software Development

AUTHOR:
J. Pérez López
L. Ribas i Xirgo

COORDINATOR:
D. Megías Jiménez
J. Mas

Free Technology Academy

Introduction to software development

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Josep Anton Pérez López
Lluís Ribas i Xirgo

PID_00148375

Universitat Oberta de Catalunya
www.uoc.edu

David Megías Jiménez

Degree in IT engineering from the UAB (Universitat Autònoma de Barcelona). Master's degree in Advanced process automation techniques from the UAB. Doctorate in IT from the UAB. Lecturer in IT and Multimedia Studies at the UOC.

Jordi Mas

Software engineer at the open source company Ximian, where he works on the implementation of the free software project Mono. He works as a volunteer on the development of the Abiword word processor and on the engineering of the Catalan versions of the Mozilla and Gnome project. He is also general coordinator of Softcatalà. He has worked as a consultant for companies such as Menta, Telépolis, Vodafone, Lotus, eresMas, Amena and Terra España.

Josep Anton Pérez López

Degree in IT from the Universitat Autònoma de Barcelona. Master's degree in Graphics programming, image processing and artificial intelligence from the UAB. Currently working as a teacher in secondary education.

Lluís Ribas i Xirgo

Degree in Computer Science and Doctorate in IT from the Universitat Autònoma de Barcelona (UAB). Professor of the IT Department at the UAB. Consultant in IT and Multimedia Studies at the Universitat Oberta de Catalunya (UOC).

First edition: February 2010
© Josep Anton Pérez López, Lluís Ribas i Xirgo
All rights are reserved
© of this edition, FUOC, 2010
Av. Tibidabo, 39-43, 08035 Barcelona
Design: Manel Andreu
Publishing: Eureca Media, SL

Preface

Software has become a strategic societal resource in the last few decades. The emergence of Free Software, which has entered in major sectors of the ICT market, is drastically changing the economics of software development and usage. Free Software – sometimes also referred to as "Open Source" or "Libre Software" – can be used, studied, copied, modified and distributed freely. It offers the freedom to learn and to teach without engaging in dependencies on any single technology provider. These freedoms are considered a fundamental precondition for sustainable development and an inclusive information society.

Although there is a growing interest in free technologies (Free Software and Open Standards), still a limited number of people have sufficient knowledge and expertise in these fields. The FTA attempts to respond to this demand.

Introduction to the FTA
The Free Technology Academy (FTA) is a joint initiative from several educational institutes in various countries. It aims to contribute to a society that permits all users to study, participate and build upon existing knowledge without restrictions.

What does the FTA offer?
The Academy offers an online master level programme with course modules about Free Technologies. Learners can choose to enrol in an individual course or register for the whole programme. Tuition takes place online in the FTA virtual campus and is performed by teaching staff from the partner universities. Credits obtained in the FTA programme are recognised by these universities.

Who is behind the FTA?
The FTA was initiated in 2008 supported by the Life Long Learning Programme (LLP) of the European Commission, under the coordination of the Free Knowledge Institute and in partnership with three european universities: Open Universiteit Nederland (The Netherlands), Universitat Oberta de Catalunya (Spain) and University of Agder (Norway).

For who is the FTA?
The Free Technology Academy is specially oriented to IT professionals, educators, students and decision makers.

What about the licensing?
All learning materials used in and developed by the FTA are Open Educational Resources, published under copyleft free licenses that allow them to be freely used, modified and redistributed. Similarly, the software used in the FTA virtual campus is Free Software and is built upon an Open Standards framework.

Evolution of this book
The FTA has reused existing course materials from the Universitat Oberta de Catalunya and that had been developed together with LibreSoft staff from the Universidad Rey Juan Carlos. In 2008 this book was translated into English with the help of the SELF (Science, Education and Learning in Freedom) Project, supported by the European Commission's Sixth Framework Programme. In 2009, this material has been improved by the Free Technology Academy. Additionally the FTA has developed a study guide and learning activities which are available for learners enrolled in the FTA Campus.

Participation
Users of FTA learning materials are encouraged to provide feedback and make suggestions for improvement. A specific space for this feedback is set up on the FTA website. These inputs will be taken into account for next versions. Moreover, the FTA welcomes anyone to use and distribute this material as well as to make new versions and translations.

See for specific and updated information about the book, including translations and other formats: *http://ftacademy.org/materials/fsm/1*. For more information and enrolment in the FTA online course programme, please visit the Academy's website: *http://ftacademy.org/*.

I sincerely hope this course book helps you in your personal learning process and helps you to help others in theirs. I look forward to see you in the free knowledge and free technology movements!

Happy learning!

Wouter Tebbens
President of the Free Knowledge Institute
Director of the Free technology Academy

Acknowledgements

The authors wish to thank the Fundació per a la Universitat Oberta de Catalunya (http://www.uoc.edu) for financing the first edition of this work under the framework of the International Master's degree in Free Software offered by this institution.

The current version of these materials in English has been extended with the funding of the Free Technology Academy (FTA) project. The FTA project has been funded with support from the European Commission (reference no. 142706- LLP-1-2008-1-NL-ERASMUS-EVC of the Lifelong Learning Programme). This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Contents

Glossary

abstract data type *m* A data type defined in a program that does not exist in the programming language.

abstract windows toolkit *m* Graphics routines package included in the package ***java.awt***, which contains all of the components needed for a graphic user interface (GUI) for Java applications.
Acronym: AWT

algorithmic scheme *m* A generic algorithm whose instructions are replaced by those needed for a particular task.

application *f* Software for specific tasks: text processing, accountancy, industrial process control, entertainment, multimedia etc.

application programming interface *f* A collection of pre-written software that provides multiple capabilities such as graphics environments, communications, multi-processing etc. already incorporated into the Java platform.
Acronym API

assembly language *m* The language used to program directly in executable code as the instructions are the same as the machine code instructions of the respective machine.

bytecode *m* A code generated by the Java compiler which is ready for execution by the Java virtual machine(JVM).

class *f* An abstract data type that includes data and functions and represents a model of a generic entity.

compiler *m* A program that translates text in one programming language to a code in another programming language, above all to machine code.

concurrent programming *f* A programming technique that prepares the code to be executed concurrently over several execution flows, or in parallel, either to take advantage of multi-processor systems or to prevent waiting time for input/output processes.

constructor *m* A special member function of a class which is automatically called each time an object of this class is instanced.

data hiding *f* An attribute that limits the visibility of the variables or methods of a class for others as it is remote or these values have been redefined.

data structure *f* Groups of data organised with specific access methods.

debugger *m* A tool that allows us to see the state of a program and to control its execution. Used for locating errors.

destructor *m* A special member function of a class that is automatically called each time an object is destroyed.

dynamic data type *m* A structured data type in which both the number of variables and the relationship between them can be changed during the execution of a program.

dynamic variable *f* A variable created during the execution of a program.

encapsulation *m* An attribute by which an element acts like a "black box" containing some inputs, a general idea of its functionality and some outputs.

event *m* External events received by the processes. The most common ones are used for input/output management (keyboard, mouse etc.).

execution threads *m pl* Instruction flows that are executed simultaneously within the same process.

file *m* A data structure usually stored in secondary memory characterised by the fact that it does not have a fixed size.

filter *m* A program that runs through data sequences and whose output is the product of the individual treatment of the input data (the name derives from the fact that the output is the product of a partial copying of the input data).

free software *m* A set of programs that include the source code so that they can be freely modified.

function *f* A sub-program that performs a task using specific parameters, the result is returned to the program that makes use of it.

graphic user interface *f* A collection of graphic objects and tools used to manage the communications between the user and the computer.
Acronym: GUI

homonymy *f* An attribute by which two or more elements with the same name perform different operations.

imperative programming *f* Programming using languages whose instructions are orders to change the state of the environment of the program.

inheritance *f* An attribute by which an object of a class can receive the attributes (data and functions) of a more general class.

integrated development environment *m* A development environment that integrates various tools for the development of software (editing, compilation, error correction etc.).
Acronym: IDE

interface *f* A collection of method definitions (without their implementations) whose function is to define a behaviour protocol for a class.

Java applet *m pl* Mini-applications designed to be executed from web browsers.

Java development kit *m* A set of programs and libraries used for the development, compilation and execution of applications in Java.
Acronym: JDK

Java virtual machine *f* A software platform that allows a generic code to be executed (***bytecode***) on an underlying platform.
Acronym: JVM

library *f* An archive of pre-compiled compilation units and lists to be linked to other programs. Sets of pre-compiled functions.

light process *See* thread.

linker *m* A program that links several codes in machine code to mount a single file which contains the code of an executable program.

list *f* A dynamic data type in which the relationship between the various elements or nodes is sequential. This means that each element has one preceding it and one following it, with the exception of the first and the last.

machine code *m* The language that can be interpreted by the processor of a computer.

member function (or method) *f* A function defined within a class and that acts on or modifies the member variables of that class.

member variables *f* Variables defined within a class.

modular programming *f* A programming technique in which the resulting code is divided into sub-programs.

node *m* Each of the variables in a dynamic data structure.

object *m* The instancing of a class, or a specific element within a class.

package *m* A collection of related classes and interfaces that are grouped under the same name, these provide access protection and name space management.

pointer *m* A variable that contains a memory address that is usually the address of some other variable.

polymorphism *m* An attribute by which an object can acquire several forms.

process *m* A sequential flow of execution instructions.

program *m* A sequence of instructions.

program environment *m* The variables and the sequence of instructions that programs use and modify.

programming *f* The act of transforming an algorithm into a sequence of instructions that can be executed by a computer.

programming language *m* A language used to describe a computer program.

queue *f* A list in which elements are inserted at one end and removed from the other.

run-through *m* A process for treating a complete data sequence.

search *f* A process for the partial treatment of a sequence of data. A run-through that stops when the search criteria are fulfilled.

signature (of a function) *f* A list of attributes that are attached to the name of a function to complete the definition: return type, number of parameters and their types and the const specifier if there is one.

software *m* A set of programs.

structured programming *f* A programming technique that uses the sequential composition of instructions (the evaluation of expressions, conditional, alternative or iterative execution).

swing *m* A package of graphics routines that allow us to create a GUI, implemented from version 1.2 of Java and that replace AWT's.

thread *m* A sequence of instructions that is executed in parallel with others within the same program. Also known as a light process as it shares the execution environment with the other threads of the same program.

tool *f* A utility program used for a specific purpose. Software development tools include text editors, compilers, linkers and debuggers among others.

unified modelling language *m* A unified language for software development allowing a model to be described in a visual way including its elements and the relationships between them.
Acronym: UML

Bibliography

Antonakos, J.L.; Mansfield, K.C. (Jr.) (1997). *Structured Programming in C*. Madrid: Prentice-Hall.

Eck, D.J. (2002). *Introduction to Programming Using Java* (4th ed.). http://math.hws.edu/eck/cs124/downloads/javanotes4.pdf

Eckel, B. (2002). *Thinking in Java* (3rd ed.). http://www.BruceEckel.com.

Eckel, B. (2003). *Thinking in C++ 2nd edition*. http://www.BruceEckel.com.

Gottfried, B. (1997). *Programming in C (Second edition)*. Madrid: McGraw-Hill.

Joyanes, L.; Castillo, A.; Sánchez, L.; Zahonero, I. (2002). *Programming in C (Exercise book)*. Madrid: McGraw-Hill.

Joyanes, L.; Zahonero, I. (2002). *Programming in C. Methodology, data structure and objects*. Madrid: McGraw-Hill.

Kernighan, B.W.; Pike, R. (2000). *The Practice of Programming*. Madrid: Prentice Hall.

Kernighan, B. W.; Ritchie, D. M. (1991). *The C Programming Language (Second edition)* Mexico: Prentice-Hall.

Liberty, J.; Horvath, David B. (2001). *Learning C++ for Linux in 21 Days* Mexico: Pearson Educación.

Palma, J.T.; Garrido, M.C.; Sánchez, F.; Santos, J.M. (2003). *Concurrent Programming*. Madrid: Thomson.

Pressman, R. (1998). *Software Engineering. A Practical Guide (4.ª ed.)* Madrid: McGraw Hill.

Quero, E.; López, J. (1997). *Programación en lenguajes estructurados (Programming using Structured Languages)*. Madrid: Int. Thomson Publishing Co.

Ribas Xirgo, Ll. (2000). *Progrramació en C per a matemàtics (Programming in C for Mathematicians)*. Bellaterra (Cerdanyola): Servei de Publicacions de la UAB.

Sun Microsystems, Inc. (2003). *The Java Tutorial: A Practical Guide for Programmers*. http://java.sun.com/docs/books/tutorial

Tucker, A.; Noonan, R. (2003). *Programming Languages. Principles and Paradigms*. Madrid: McGraw-Hill.

Appendix

cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machinereadable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4) MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6) COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7) AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

8) TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their

An introduction to programming

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Josep Anton Pérez López
Lluís Ribas i Xirgo

PID_00148431

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

This unit looks at the fundamentals of programming and the basic concepts of the C language. It is assumed that the reader already has a certain knowledge of programming using either C or another language.

For this reason we will take a special look at the methodology of programming and key aspects of the C language and will not be covering the more basic aspects of these issues.

Profound understanding of a programming language does not just come from knowledge of its lexicon, syntax or semantics, but also requires an understanding of the goals which motivated its development. In this unit we will therefore be looking at the history of the C programming language from the perspective of computer programming.

Programs written in languages such as C can not be directly run by computers. We therefore need other tools (programs) to produce other programs which contain a sequence of commands that can be directly run by a computer.

As such we are going to look at the free software development environments which are available for platforms such as Microsoft and GNU/Linux. Given that the first of these requires an operating system which is not based on free software we will concentrate on the second.

Note

In this context a platform is a system made up of some type of computer running an operating system.

The rest of the unit focuses on the C programming language, the imperative programming paradigm and its execution model. The execution model deals with the way the program's instructions are performed. The imperative programming paradigm uses instructions as commands which are carried out immediately, these cause changes in the state of the processor and, in particular, store the results of calculations carried out in the execution of the instructions. As such, the final sections will deal with issues surrounding the evaluation of expressions (calculating the results of formulas), the selection of instructions to be performed and obtaining data or producing results.

In this unit we are aiming to achieve the following goals:

1) To review the basic concepts of programming and program execution models.

2) To understand the fundamental paradigm of imperative programming.

3) To acquire the concepts of C needed for this course.

4) To learn how to use a free software development environment for programming in C (GNU/Linux and GNU/C tools).

1. A bit of history on C

The C programming language was designed by Dennis Ritchie at Bell Laboratories for the development of new versions of the Unix operating system in 1972. Hence the strong relationship between C and the machine.

Note

It is interesting to note that only thirteen thousand lines of C code (and less than a thousand in assembly language) were needed to program the Unix operating system for the PDP-11 computer.

Assembly language is much closer to the machine code which is understood by the processor. In other words, each machine code instruction corresponds to an assembly language instruction.

Conversely, C language instructions can be compared to small programs in machine code, but which are often used in algorithms for computer programs. This means that instructions created in this language can only be processed by an abstract machine which does not in fact exist in reality (the processor only understands machine code). This means we can speak about C as a language with a high level of abstraction and assembly as a low level language.

This abstract machine can be partially constructed using a set of programs which manage a real machine: the ***operating system***. The other part is built using a program to translate the high-level language to machine code. These programs are called ***compilers*** and they generate code which can be directly executed by the computer, or ***interpreters*** if they need to be executed to carry out the program written in a high-level language.

In our case we would prefer that the code of the programs making up the operating system be as independent from the machine as possible. It will only be viable to adapt an operating system to any computer quickly and easily if this is the case.

The high-level language compiler must also be extremely efficient. Given the scarcity of computing resources (basically memory capacity and speed) in computers of those days, the language needed to be simple and to allow translations which were highly adapted to the processors.

For this reason the C language was derived from the language known as B, this was developed by Ken Thompson to program Unix for the PDP-7 in 1970. Evidently this version of the operating system also included a part programmed in assembly language as there were operations which could only be performed by a real machine.

We can clearly look at C as a later version of B (improved with the inclusion of data types). Furthermore, the B language was based on BCPL. This language was developed by Martin Richards in 1967 and its basic data type was the *memory* word; this being the unit of information into which computer memory is divided. This was in fact just an improved assembly language which required a very simple compiler. The programmer thus had more control over the real machine.

Despite its development as a language for programming operating systems and, therefore, with the ability to express low-level operations, C is a **general purpose**. This means it can be used to program application algorithms (groups of programs) with very different characteristics, for example, accounting software for businesses, databases for aircraft reservations, goods transport fleet management, scientific calculations etc.

Bibliography

The syntactical and semantic rules of C are included in the following work:

B.W. Kernighan; D.M. Ritchie (1978). *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall. Specifically in the Appendix "C Reference Manual".

The relative simplicity of the C language, in that it has few instructions, allows compilers to generate machine code very efficiently and also makes it easy to port from one machine to another.

On the other hand, the list of instructions in C also allows for structured programming with a high level of abstraction. This makes programming systematic, legible and easy to maintain.

This simplicity has also meant that C had to be given a very complete set of functions, making it very powerful for the development of applications of all types. Many of these functions are standard and are available on all C compilers.

Note

A function is a sequence of instructions which are executed one-by-one to perform a specific task. As such, the more functions that are programmed beforehand, the less code will be needed.

Standard C functions are contained in a library. the standard function library. Any program can therefore use all of the functions required as all compilers have them.

Lastly, given its dependence on standard functions, C encourages modular programming in that programmers themselves can also create specific functions in their programs.

All of these features meant that C was widely distributed and it was standardised by ANSI (American National Standards Institute) in 1989 based on the work of a committee created in 1983 to "provide an unambiguous definition independent of machine code". The second edition of Kernighan and Ritchie, which was published in 1988, uses this version, which is known as ANSI-C.

Note

The original version of C then became known as K&R C, meaning *Kernighan and Ritchie C*. This was to distinguish the original version from the one standardised by ANSI.

The rest of the unit will be dedicated to explaining a program development environment in C and reviewing the syntax and semantics of the language.

2. GNU/C programming environment

In the development of free software it is also important to use tools (programs) which are also free, one of the principles of free software is that its code can be modified by users.

The use of code which may depend on private software implies that this is not possible, it can not therefore be considered as being free. To avoid this problem we need to program using free software.

GNU (www.gnu.org) is an open software development project started by Richard Stallman in 1984 and it is supported by the Free Software Foundation (FSF).

GNU is a recursive acronym (it means *GNU is not Unix*) to indicate that it is free software developed using this operating system but that it is not the operating system itself. Although the software initially used a Unix platform, which is a private operating system, it was not long before the Linux *kernel* was incorporated as the foundation of an independent and complete operating system: GNU/Linux.

Note

The kernel of an operating system is the software forming the fundamental nucleus, hence the name (*kernel* means "the essential part" among other things). The nucleus is basically in charge of managing the resources of the machine for programs running on it.

The Linux kernel is compatible with Unix and was developed by Linus Torvalds in 1991, it was implemented as the kernel of the GNU operating system a year later.

We also need to remember that the so-called *Linux distributions* are in fact versions of the GNU/Linux operating system and therefore use GNU software (Emacs text editor, C compiler etc.) and other free software such as the TeX text formatter.

When developing free programs we will therefore use a computer running the Linux operating system and the C compiler for GNU (gcc). Although we could use any text editor it would seem logical to use Emacs. An ASCII text editor is one which only uses the characters defined in the ASCII table for storing text. (In Emacs, the representation of text can be modified to make comments easily distinguishable from the C program code, for example).

Although the explanations on the software development environments given in this and later units refer to GNU software, we should note that it is also possible to use free software tools for Windows, for example.

In the following sections we will learn how to use the `gcc` compiler and we will take a close look at the organisation of code in C programs.

2.1. Our first C program

> A program is a text written in a simple language allowing us to express a series of actions to be performed on objects (instructions) in an unambiguous way.

Before writing a program, as with all written text, we need to know the rules of the language so that it can be expressed correctly both in terms of lexicon and syntax.

We will progressively look at the rules of the C programming language over the course of the first three units.

We also need to make sure that "this" makes sense and that we have expressed exactly what we want the program to do. If this was not enough, we also need to take care with the appearance of the text so that it can be quickly and easily understood. Although style rules will sometimes be indicated, these will normally be implicitly described by the examples.

Note

The dollar sign ($) is used to indicate that the operating system command interpreter can accept a new order.

To write a program in C we need to run `emacs`:

```
$ emacs hello.c &
```

We can now write the following program in C:

Note

The name of the file containing the C program will have the extension ".c" so that it can be easily identified.

```
#include <stdio.h>
main()
{
 printf( "Hello world! \n" );
} /* main */
```

Example

Writing `printf` is not the same as writing `PrintF`.

It is very important to remember that C (and also in C++ and Java) is **case sensitive**. This means that the program must be written exactly as shown, with the exception of text in inverted commas or text between the symbols `/*` and `*/`.

The text editor emacs has drop-down menus at the top for most operations and it also accepts keyboard commands. You will need to hold down the control key ("CTRL" or "C-") or the Alt key when pressing the command key.

Some of the most frequently-used commands are summarised on the following table:

Note

For a better understanding of commands we recommend you to read the emacs manual, or that of the editor you have chosen to write programs with.

Table 1.

Command	Sequence	Explanation
Files → Find file	C-x, C-f	Open file. The file is copied to a *buffer* or temporary area for editing.
Files → Save buffer	C-x, C-s	Saves the content of the *buffer* to the associated file.
Files → Save buffer as	C-x, C-w	Writes the content of the *buffer* to the file indicated.
Files → Insert file	C-x, C-i	Inserts the content of the file indicated at the cursor position.
Files → Exit	C-x, C-c	Exits emacs.
(cursor movement)	C-a	Places the cursor at the start of the line.
(cursor movement)	C-e	Places the cursor at the end of the line.
(line killing)	C-k	Deletes the line (first the content and then the return).
Edit → Paste	C-y	Pastes the last text removed or copied.
Edit → Copy	C-y, ..., C-y	To copy text, it can first be deleted and then restored in the same place and then in the destination position.
Edit → Cut	C-w	Removes the text from the last marker to the cursor.
Edit → Undo	C-u	Undoes the last command.

Once the C program has been edited and saved it must be compiled to get the binary file (ones and noughts), this contains a version of the program translated into machine code. To do this we will use the gcc:

```
$ gcc –c hello.c
```

Note

Previously, gcc stood for the GNU C compiler, but as the compiler also understands other languages it has become a collection of GNU compilers. For this reason we need to indicate the language in which the program has been written using the corresponding file name extension. In this case with, the name hello.c, we will use the C compiler.

This will give us a file (hello.o), known as an object file. This file contains the program written in machine code which was derived from the program written in C code, also known as *source code*. However, we are still not able to execute this program as it requires a function (printf) contained in the standard function library for C.

To get the executable code for the program, we need to link:

```
$ gcc hello.o –o hello
```

As the location of the standard function library will always be known by the compiler this does not need to be entered in the command line. However, we do need to indicate the file we want to process (the executable file) using the -o option followed by the desired name. If we do not do this we will get the result in a file called "`a.out`".

The compilation and linking process is usually done directly using:

```
$ gcc hello.c –o hello
```

If the source file contains syntax errors the compiler will show the corresponding error message and they must be corrected before repeating the compilation process.

If all goes well we will have an executable program in a file called `hello` which will greet us each time we run it.

```
$ ./hello
Hello world!
$
```

Note

We need to indicate the access path for the executable file so that the command interpreter can find it. For safety reasons the working directory is not included by default in the set of search paths for executables of the command interpreter.

This procedure is repeated for each program written in C in a GNU environment.

2.2. The structure of a simple program

In general a C program should be organised as follows:

```
/* File: name.c                                         */
/* Content: example of the structure of a C program     */
/* Author: name of the author                           */
/* Version: preliminary                                 */
/* PREPROCESSOR COMMANDS                                */
/* -inclusion of header files                           */
#include <stdio.h>
/* -definition of symbolic constants                    */
#define FALSE 0
```

```
/* PROGRAMMER FUNCTIONS                                    */

main( ) /* Main function:                                  */
{       /* The execution of the program starts here        */
...     /* Body of the main function                       */
}       /* main                                            */
```

With this organisational layout the first lines are comments which identify the content, author and the version. This is important, we need to remember that the source code we create must be easy to use and modify by other people... and also by ourselves!

Note

Comments are any text which is written between the symbols `/*` and `*/`.

Given the simplicity of C, many of the operations performed by a program are in fact standard function calls. In order for the compiler to know the parameters they have and which values to return, we need to include declarations of the functions used in our code. To do this we use the command `#include` of the so-called *C preprocessor*, this sets up a unique input file for the C compiler.

Files which contain function declarations outside of a certain file are called *header files*. This is why the extension ".h" is used to indicate its content.

In C, header files are used to declare the name of a function, the parameters it must receive and the type of data returned.

Both these files and the source code of a program contain definitions of symbolic constants. Some of these definitions are shown below:

```
#define EMPTY        '\0' /* The ASCII NULL character   */
#define OCTAL_NUMBER 0173 /* An octal value             */
#define MAX_USERS 20
#define HEXA_CODE  0xf39b /* A hexadecimal value        */
#define PI           3.1416 /* A real number            */
#define PRECISION    1e-10 /* Another real number       */
#define STRING       "characters string"
```

These symbolic constants are replaced by their value in the file which is sent to the C compiler. It is important to note that they are used to make the code more legible and also to facilitate changes to the program when necessary.

We must remember that integer numeric constants written in base 8, or octal, must be preceded by a 0 and those expressed in hexadecimal, or base 16, by "0x",

Example

020 is not the same as 20 as the second of these corresponds to the value twenty on decimal and the first is expressed in base 8, the binary representation of which is 010000, or 16 in base 10.

The program is written in the body of the main function. This function must be present in all programs and the first instruction it contains is taken as the starting point of the program, as such it will be the first to be executed.

3. Imperative programming

Programming consists of the translation of algorithms into programming language versions which can be directly or indirectly executed by a computer.

Most algorithms are made up of a sequence of steps indicating what needs to be done. These instructions are usually imperative in nature, meaning that they are unconditional.

The programming of algorithms in these terms is known as *imperative programming*. In these types of programs each instruction involves the performance of a certain action on its surroundings, in this case, the computer running the program.

To understand how an instruction is executed we need to know more about the surroundings in which it is carried out. Most processors are organised in such a way that the data and instructions are stored in the main memory and the central processing unit (CPU) carries out the algorithm in order to execute the program in the memory:

1) Read the instruction to be executed from the memory.

2) Read the data required for execution from the memory.

3) Perform the calculation or operation indicated in the instruction and, depending on the operation, save the result in the memory.

4) Determine the next instruction to be executed.

5) Return to the first step.

The CPU references the instructions and data requested from the memory and the results it wishes to store using the position number they occupy in the memory. The position occupied by the data and instructions is known as the *memory address*.

At the lowest level each different memory address is a single byte and data and instructions are identified by the address of the first byte they occupy. At this level, the CPU corresponds to the physical CPU of the computer.

On the level of the abstract machine which executes C, data and instructions are still referenced from the physical memory address, but the instructions which can be executed by the high level CPU are more powerful than those which can be performed by the real one.

Independently of the level of abstraction at which we are working, the memory is the environment of the CPU. Each instruction carries out a modification to the environment in this execution model: they can modify data in the memory and will always determine which is the address of the next instruction to be executed.

In other words: the execution of an instruction will imply a change in the state of the program. This includes the address of the instruction being executed and the value of data in the memory. Carrying out an instruction therefore involves changing the state of the program.

In the following sections we will describe the basic data types which a C program can use and the fundamental instructions which change its state: the assignation and conditional selection of the following instruction. Lastly, we will look at the standard functions for obtaining external data (from the keyboard) and for displaying it (via the screen).

3.1. Basic data types

The types of primitive data which a language can use are those which can be processed by instructions in the same language; meaning that they are supported by the corresponding programming language.

3.1.1. Compatible with integers

In C the most common primitive data types are those which are compatible with integers. The binary representation of these is not encoded but corresponds to the numeric value represented in base 2. We can therefore calculate its numeric value in base 10 by adding the products of the intrinsic values (0 or 1) of the digits (bits) by its corresponding position values (2^{position}).

They are treated as natural numbers, or rather as representations of integers in base 2, if they can be negative. In this case, the most significant bit (the one furthest to the left) is always a 1 and the absolute value is obtained by subtracting the natural number from the maximum value which can be represented with the same number of bits plus 1.

We must always remember that the range of values for this data will depend on the number of bits used to represent it. The following table shows the various integer-compatible data types in a 32-bit computer running a GNU system.

Table 2.

Specification	Number of bits	Range of values
`(signed) char`	8 (1 byte)	from –128 to +127
`unsigned char`	8 (1 byte)	from 0 to 255
`(signed) short (int)`	16 (2 bytes)	from –32,768 to +32,767
`unsigned short (int)`	16 (2 bytes)	from 0 to 65,535
`(signed) int`	32 (4 bytes)	from –2,147,483,648 to +2,147,483,647
`unsigned (int)`	32 (4 bytes)	from 0 to 4,294,967,295
`(signed) long (int)`	32 (4 bytes)	from –2,147,483,648 to +2,147,483,647
`unsigned long (int)`	32 (4 bytes)	from 0 to 4,294,967,295
`(signed) long long (int)`	64 (8 bytes)	from -2^{63} to $+(2^{63}-1) \approx \pm 9.2\text{x}10^{18}$
`unsigned long long (int)`	64 (8 bytes)	from 0 to $2^{64}-1 \approx 1.8\text{x}10^{19}$

Note

The words of the specification in brackets are optional when declaring the corresponding variables. We must also bear in mind that the specifications can vary slightly for other compilers.

We must always remember the different value ranges that each type of variable can take to be correctly used in programs. We can adjust this value to the one which is most useful.

The character type (`char`) is an integer which identifies a position on the table of ASCII characters. To avoid having to translate the characters to numbers they can be entered between single inverted commas (for example: '`A`'). We can also represent non-visible codes such as a line feed ('`\n`') or tab ('`\t`').

Example

In this standard the letter A in capitals is found at position number 65.

3.1.2. Real numbers

This data type is more complicated than the one above and its binary representation is encoded over several fields. It does not therefore correspond to the number which can be extracted from the bits making it up.

Real numbers are represented using a sign, mantissa and exponent. The mantissa expresses the fractional part of the number and the exponent is the number to which the corresponding base is raised.

$$[+/-]\ \text{mantissa x base}^{\text{exponent}}$$

The values of the mantissa and exponent will be higher or lower depending on the number of bits used to represent them. The various types of real numbers and their approximate ranges are shown in the following table (valid for 32-bit GNU systems):

Table 3.

Specification	Number of bits	Range of values
`float`	32 (4 bytes)	$\pm 3.4 \times 10^{\pm 38}$
`double`	64 (8 bytes)	$\pm 1.7 \times 10^{\pm 308}$
`long double`	96 (12 bytes)	$\pm 1.1 \times 10^{\pm 4.932}$

As we can see from the above table, it is important to adjust the real data type to the range of values which a certain variable may acquire so as not to occupy memory unnecessarily. The reverse is also true: the use of a data type which is not able to represent the extreme values of the range being used will mean that they are not represented correctly, the program may then behave erratically.

3.1.3. Declarations

Declaring a variable involves making the compiler aware of it so that it can reserve a space in memory to store its data. A declaration is made by stating its type specification before the name (`char, int, float, double`), this may also be preceded by one or more modifiers (`signed, unsigned, short, long`). The use of a modifier makes specification `int` unnecessary except for `long`. For example:

```
unsigned short natural;  /* The 'natural' variable is       */
                         /* declared to be a                */
                         /* positive integer.               */
int          i, j, k;    /* Integer variables with a sign.  */
char         option;     /* Character-type variable.        */
float        percentile; /* Real number variable.           */
```

For greater ease a variable can be assigned a specific initial content. To do this we need to add an equal sign to the declaration followed by the initial value on execution of the program. For example:

```
int       value = 0;
char      option = 'N';
float     angle = 0.0;
unsigned  counter = MAXIMUM;
```

The name of a variable can contain any combination of alphabetic characters (meaning those of the English alphabet), numbers and also the underscore sign (_); however they may not begin with a digit.

We recommend that you choose variables names which identify their content or their use in the program.

3.2. The assignment and evaluation of expressions

As we have already mentioned, in imperative programming the execution of instructions involves changing the state of the program environment or, similarly, changing the reference of the instruction to be executed and possibly the content of one variable or another. The last of these occurs when the instruction executed is an assignment:

variable = expression in terms of variables and constant values;

The power of C (and possibly the difficulty in reading programs) arises from the use of expressions.

In fact, any expression is converted to an instruction if we put a semi-colon at the end: all instructions in C end in a semi-colon.

Note

An expression is any syntactically valid combination of operators and operands, these may be variables or constants.

Obviously, solving an expression makes no sense if we do not then assign the result to a variable which can be stored for later operations. Therefore the first operator we need to look at is that of assignment:

```
integer = 23;
destination = origin;
```

It is important not to confuse the assignment operator (=) with the equality comparator (==); in C both of these operators can be used between data of the same type.

3.2.1. The operators

Apart from assignment, the most frequently-used operators in C, and which also appear in other programming languages, are those shown in the following table:

Table 4.

Class	The operators	Meaning
Arithmetic	+ -	add and subtract
	* /	multiply and divide
	%	modulus or remainder after integer division (only for integers)
Relational	> >=	"greater than" and "equal to or greater than"
	< <=	"less than" and "equal to or less than"
	== !=	"equal to" and "not equal to"
Logical	!	*NO* (logical proposition)
	&& ||	*AND* (all parts must be satisfied) and logical *OR*

Arithmetical operators can be used for both real number and integers. For this reason we implicitly perform all operations using the data type with the greatest range.

This implicit behaviour of operators in known as *data type promotion* and it is performed each time different types of data are operated upon.

For example, the result of an operation using an integer and a real number (real constants must contain a decimal point or the letter "e" separating the mantissa and the exponent) will always be a real number. Conversely, if the operation involves two integers the result will always be expressed in the integer data type with the greatest range. So:

```
real = 3 / 2 ;
```

results in the assignment of the value `1.0` to the variable `real`, this is the result of the integer division of 3 by 2, transforming it into a real number when the assignment operation is performed. That is why it is written as 1.0 (with a decimal point) instead of 1.

Even so, the assignment operator always converts the result of the source expression to the data type of the destination variable. For example, the following assignment:

```
integer = 3.0 / 2 + 0.5;
```

assigns the value 2 to `integer`. In this case the real numbers are divided (the number 3.0 is real as it has a decimal point) and 0.5 is added to the result. This acts as a rounding factor. The resulting number is real and it is truncated (its decimal is removed) when it is assigned to the variable `integer`.

3.2.2. Typecasting

To improve the legibility of the code, to prevent incorrect interpretation and to prevent the erroneous use of automatic type promotion, we recommend that you explicitly indicate that the data type has changed. We can use typecasting to do this, this means: putting the data type we wish to convert a certain value to in brackets (whether it is a constant or a variable):

```
( type_specification ) operand
```

Following the above example it is therefore possible to convert a real number to the nearest integer by rounding in the following way:

```
integer = (int) (real + 0.5);
```

In this case we indicate that two real numbers are added and the result, which will be a real number, is explicitly converted to an integer using typecasting.

3.2.3. Relational operators

We need to remember that in C there is no type of logical data corresponding to "false" and "true". Therefore, any integer-compatible data will indicate "false" if it is 0 and "true" if it is not 0.

Note

This may not happen with real-type data as we must remember that even infinitesimally small numbers will be taken to be a "true" logical result.

As a consequence of this, relational operators return 0 to indicate that the relation has not been fulfilled and a number other than 0 if it has. The operators && and || only solve those expressions needed to determine, respectively, if all cases are satisfied or if only some are. As such, && implies that the expression which makes up the second argument will only be solved if the first has produced a positive result. Similarly, || will only solve the second argument if the first has given a "false" result.

Therefore:

```
(20 > 10) || ( 10.0 / 0.0 < 1.0 )
```

will give a "true" result despite the fact that the second argument cannot be solved (division by 0!).

In the above case the brackets are unnecessary as relational operators have greater priority than logical ones. Even so, we recommend you use brackets in expressions for greater clarity and to dispel any doubts on the order of the operators the expression may contain.

3.2.4. Other operators

As we have mentioned, C was initially conceived for programming operating systems and, as a consequence, it is closely related to the machine, this is demonstrated by the existence of a group of operators designed to facilitate the efficient translation of instructions to machine code.

Specifically, it has increment (++) and decrement (--) operators which apply directly to variables with integer-compatible content. For example:

```
counter++; /* Is equivalent to: counter = counter + 1; */

--decrement; /* Is equivalent to: decrement = decrement –
1; */
```

Whether the operator is a prefix (preceding the variable) or a postfix depends on whether it is an increment or a decrement: if it is a prefix it is executed before using the content of the variable.

Example

See how the contents of the variables are modified in the following example:

```
a = 5;   /* ( a == 5 )              */
b = ++a; /* ( a == 6 ) && ( b == 6 ) */
c = b--; /* ( c == 6 ) && ( b == 5 ) */
```

It is also possible to carry out operations between bits. These operations are performed between each of the bits of an integer-compatible number and another. We can therefore perform an AND, an OR, an EX-OR (only one of the two can be true) and a bit-by-bit logical negation between those of one data string and those of another (a zero bit means "false" and a one means "true".)

The following symbols are used for bit-level operators:

- For logical AND: & (*ampersand*)
- For logical OR: | (vertical bar)
- For logical exclusive OR: ^ (circumflex accent)
- For logical negation or complement: ~ (tilde)

Despite the fact that these are valid operations between integer-compatible data in the same way as logic operators are, we must remember that they do not produce the same result. For example: `( 1 && 2 )` is true, but `( 1 & 2 )` is false, as it gives 0. To prove this we need to look at what is happening on the bit level:

```
    1 == 0000 0000 0000 0001
    2 == 0000 0000 0000 0010
1 & 2 == 0000 0000 0000 0000
```

The list of C operators does not end there. We will look at some more later but we will leave others alone.

3.3. Selection instructions

In the program execution model, instructions are executed in a sequence, one after the other, in the same order as they appear in the code. It is true that purely sequential execution does not permit very complicated programs to be written, as the same operations will always be performed. We therefore need instructions which allow us to control the execution flow of the program. In other words, we need instructions which can alter the sequential order of their execution.

In this section we will look at instructions in C which allow us to select different sequences of instructions. These are briefly summarised in the following table:

Table 5.

Instruction	Meaning

if(condition **)**
  instruction_yes ;
else
  instruction_no ;
	The `condition` must be an expression for which the solution is an integer-compatible data type. If the result is not zero, the condition is considered to be fulfilled and it executes the `instruction_yes`. If it is not, it executes the `instruction_no`. The **`else`** is optional.

switch(expression **) {**
  case value_1 :
instructions
  case value_2 :
instructions
  default :
instructions
} /* switch */
	The evaluation of the `expression` must result in an integer-compatible data string. This result is compared with the indicated values in each case and, if it is equal to one of them, all instructions are executed from the first indicated for this case up to the end of the block of the **`switch`**. It is possible to "break" this sequence by inserting an instruction called **`break;`** this ends the execution of the instruction sequence. As an option you can also indicate a default case (**`default`**) which allows you to specify which instructions will be executed if the result of the expression has not produced data which coincides with any of the cases.

In the case of `if` it is possible to execute more than one instruction, whether the condition is satisfied or not, by grouping the instructions into a block. Instruction blocks are those written between braces:

```
{ instruction_1; instruction_2; ... instruction_N; }
```

In this sense we recommend that all conditional instructions include the instructions to be executed in each case:

```
if( condition )
{ instructions }
else
{ instructions }
/* if */
```

This avoids confusing cases such as the following:

```
if( a > b )
 larger = a ;
 smaller = b ;
difference = larger - smaller;
```

In this case we assign `b` to `smaller`, regardless of the condition, so the only instruction of the `if` is the assignment to `larger`.

As it can be seen, instructions which belong to the same block always start in the same column. To facilitate the identification of blocks they should have an indentation to the right with respect to the initial column of the instruction governing them (in this case: `if`, `switch` and `case`).

For greater convenience each block of instructions should be indented to the right with respect to the instruction determining its execution.

Given the fact that the assignment of one value or another to a variable often depends on a condition, it is possible to use a conditional assignment operator instead of an `if` instruction in these cases:

```
condition ? expression_if_true : expression_if_false
```

So, instead of:

```
if( condition )    var = expression_if_true;
else               var = expression_if_false;
```

We can write:

```
var = condition ? expression_if_true : expression_if_false;
```

What is more, we can use this operator in any expression. For example:

```
cost = ( km>km_contract? km-km_contract : 0 ) * COST_KM;
```

It is preferable to limit the use of the conditional operator to cases in which it facilitates reading.

3.4. Standard input and output functions

The C language only has flow control operators and instructions. Any other operation required must be programmed, or we could also use programs from our program library.

Note

We have already seen how a function is no more than a series of instructions which are executed as a unit to perform a specific task. To get an idea of this we can use mathematical functions which perform some operation using the arguments given and which return a solution.

The C language has a wide range of standard functions among which are the data input and output functions that we will look at in this section.

The C compiler needs to know (name and data type of the arguments and the value returned) which functions our program will use to be able to generate the executable code correctly. As such, we need to include header files containing the declarations in the code of our program. In this case:

```
#include <stdio.h>
```

3.4.1. Standard output functions

The standard output is the place where the resulting data is shown (messages, results etc.) by the program being executed. This will normally be the computer screen or a window on the screen. In the second case, the window will be associated with the program.

```
printf( "format" [, list_of_fields ] )
```

This function will print the text contained in "format" on the screen (the standard output). In this text special characters, which must be preceded by a back slash (\) will be replaced by their ASCII meaning. Furthermore, the field specifiers, which are preceded by a %, are replaced by the value resulting from the corresponding expression (normally the content of a variable) indicated in the list of fields. This value is printed using the format indicated by the specifier.

The following table shows the relationship between the format string symbols and the ASCII characters. *n* is used to indicate a digit of a number.

Table 6.

Characters	ASCII character
`\n`	*new line* (line feed)
`\f`	*form feed* (page break)
`\b`	*backspace*
`\t`	*tab*
`\nnn`	ASCII number *nnn*
`\0nnn`	ASCII number *nnn* (in octal)
`\0Xnn`	ASCII number *nn* (in hexadecimal)
`\\`	*backslash*

Field specifiers have the following format:

```
[%[-][+][width[.precision]]data_type
```

The square brackets indicate that the element is optional. The *minus* sign indicates alignment to the right which is often used when printing numbers. Furthermore, if we specify the *plus* sign, the numbers will be preceded by their sign, be it positive or negative. Width is used to indicate the minimum number of characters used to display the field and, in the specific case of real numbers, you can also specify the number of decimal points to be displayed using precision. It is obligatory to include the data-type to be displayed, this can be one of the following:

Table 7.

Integers		Real numbers		Other	
%d	In decimal	%f	In floating point	%c	Character
%i	In decimal	%e	In exponential format: [+/-]0.000e[+/-]000 with lower or upper case e (%E)	%s	Character string
%u	In decimal without a sign			%%	The % sign
%o	In octal without a sign			(incomplete list)	
%x	In hexadecimal	%g	In e, f, or d format.		

Example

```
printf( "The amount of invoice number: %5d", inv_num );
printf( "for Mr/Ms. %s is %.2f_\n", client, amount );
```

For numeric types it is possible to prefix the type indicator with an "l" in the same way as with the declaration of types with `long`. In this case, the `double` type must be treated as a "`long float`" and therefore as "`%lf`".

`putchar(` `character` **`)`**

Shows the indicated character on the standard output.

Example

```
putchar( '\n' );
```

```
puts( "character string" )
```

Shows a character string on the standard output.

Example

```
puts( "Hello world!\n"
);
```

3.4.2. Standard input functions

These obtain data from the standard input which will usually be the keyboard. They return an additional value which reports the result of the reading process. The value returned does not have to be used if it is not needed, we will usually know that data input can be performed without problems.

```
scanf("format" [, list_of_&variables ] )
```

Reads from the keyboard *buffer* and transfers the content to the variables it has as arguments. Reading is performed in accordance with the indicated format in a similar way as the specification of fields used for `printf`.

Note

The keyboard *buffer* is the memory which stores everything which is typed.

In order to be able to deposit the data in the indicated variables, this function requires that the arguments on the variable list are the memory addresses at which they are to be found. For this reason we need to use the operator "address of" (&). In this way, `scanf` directly deposits the information it has read in the corresponding memory location, naturally the affected variable will be modified to show the new value.

It is important to remember that if we specify less arguments than field specifiers in the format the results may be unpredictable, as the function will change the content of some memory locations in a random way.

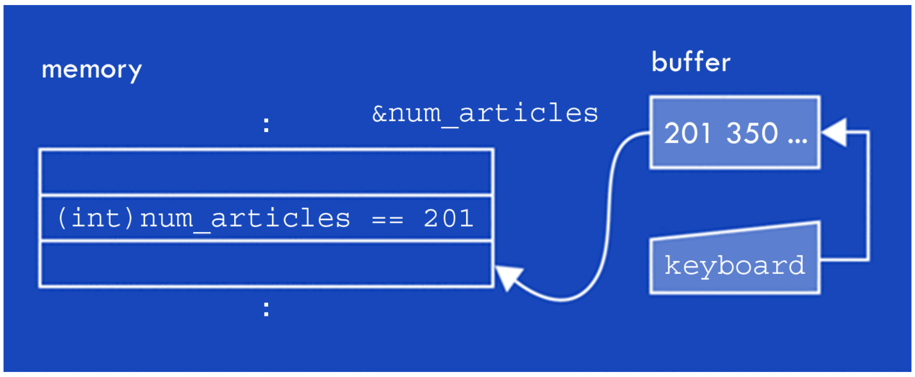
Example

```
scanf( "%d", &num_articles );
```

In the above example, `scanf` reads the characters which have been typed to convert them to an integer (presumably). The value obtained is placed in the address indicated by its argument, the one that is in the variable `num_articles`. Scanning characters from the memory *buffer* is stopped when the character read does not correspond to a possible character of the specified format. The character will be returned to the *buffer* for posterior reading.

For the input shown in the following example the function stops reading after the blank space separating the typed numbers. This and the rest of the characters stay in the *buffer* for posterior readings.

Figure 1.



The `scanf` function returns the number of pieces of data which have been correctly read. I.e. all those for which compatible text has been found with a representation of its type.

`getchar( )`

Returns a character read by the standard input (usually the keyboard *buffer*).

Example

```
option = getchar();
```

If no character can be read it returns the character `EOF`. This constant is defined in `stdio.h` and, as such, can be used to determine if the reading was successful, whether an error occurred or whether it reached the end of the input data.

`gets(` `character_string` **`)`**

Reads a series of characters from the standard input until it comes to an end of line (`'\n'`). This last character is read but it is not stored in the character string of the argument.

Example

```
gets( user_name );
```

If nothing is read it returns `NULL`, which is a constant defined in `stdio.h` for which the value is 0.

Summary

In this unit we have looked at the execution procedure for programs on a computer. The unit in charge of processing information (central processing unit or CPU) reads an instruction from the memory and executes it. This operation involves a change in the state of the environment of the program, the content of one or more of its variables and the address of the next instruction.

This execution model for instructions, which is used by most real processors, is replicated in the imperative programming paradigm for high-level languages. We have seen how this is true for the C programming language in particular.

For this reason we have reviewed the instructions in this language which allow us to modify the environment by changing data and changing the sequential order of the instructions in the memory.

The changes which affect program data are, in fact, assignments of the variables they contain. We have seen that variables are spaces in the memory of the computer which can be referenced using the name which has been declared and can be found internally using the address of the first memory word they occupy.

We have looked at the way in which expressions are resolved using priorities for operators and how we can organise the code better using brackets. We have mentioned the convenience of using explicit typecasting to prevent the incorrect use of automatic data type promotion. Promotion must be performed so that operators always work with operands of the same type and which have the greatest range.

In terms of execution flow control, which normally follows the order in which the instructions are found in the memory, we have looked at the basic instructions used for selecting sequences of instructions: The instruction `if` and the instruction `switch`.

Using these explanations we have introduced a special way of considering logic data ("false" and "true") in C. This means that any piece of data can be used as a logic value at any time. In relation to this subject, we have mentioned the special way in which the AND and OR logic operators are solved, in that the right-hand expression is not solved if the logic value result can be determined from the first argument (that produced by the preceding expression).

In the last section we reviewed the basic standard data input and output functions used to construct programs to test not only the explicit concepts and elements of C, but also those of the GNU/C development environment.

This allows us to see how compilation and program links work in practice. The task of the C compiler is to translate the C program into a machine code program. The linker is there to add the library functions used in the program to this version of the code. This is the process used to obtain an executable code.

Self-evaluation

1. Edit, compile (and link), execute and check the operation of the following program:

```
#include <stdio.h>
main()
{
  int a, b, sum;
  printf( "Type in an integer: " );
  scanf( "%d", &a );
  printf( "Type in another integer: " );
  scanf( "%d", &b );
  sum = a + b;
  printf( "%d + %d = %d\n", a, b, sum );
} /* main */
```

2. Write a program which calculates the total given a certain VAT rate applied to a certain amount of Euros.

3. Write a program which calculates the cost of 1 kg or 1l of a product if we know the price on the package and the amount of the product it contains.

4. Modify the above program so that it calculates the price of the product for the desired amount, this should also be able to be input.

5. Write a program which gives the change due, knowing the cost of the product and the amount paid. The program must indicate if the amount paid is insufficient.

6. Write a program which, given the approximate number of litres in the fuel tank of a car, the consumption per 100 km and a certain distance in kilometres, will calculate if the car is able to complete the journey. If it is not, the program should indicate the number of extra litres needed.

Answer key

1. Follow the steps indicated. As an example, if the file is called "sum.c", you will need to do the following after creating it:

```
$ gcc sum.c -o sum
$ sum
Type in an integer: 154
Type in another integer: 703
154 + 703 = 857
$
```

2.

```
#include <stdio.h>
main()
{
 float amount, VAT, total;

 printf( "Amount = " );
 scanf( "%f", &amount );
 printf( "%% VAT = " );
 scanf( "%f", &VAT );
 total = amount * ( 1.0 + VAT / 100.0 );
 printf( "Total = %.2f\n", total );
} /* main */
```

3.

```
#include <stdio.h>
main()
{
 float price, price_unit;
 int quantity;

 printf( "Price = " );
 scanf( "%f", &price );
 printf( "Quantity (grams or millilitres) = " );
 scanf( "%d", &quantity );
 price_unit = price * 1000.0 / (float) quantity;
 printf( "Price per Kg/Litre = %.2f\n", price_unit );
} /* main */
```

4.

```
#include <stdio.h>
main()
{
 float price, price_unit, price_purchase;
 int quantity, quant_purchase;

 printf( "Price = " );
 scanf( "%f", &price );
 printf( "Quantity (grams or millilitres) = " );
 scanf( "%d", &quantity );
 printf( "Quantity to purchase = " );
 scanf( "%d", &quant_purchase );
 price_unit = price / (float) quantity;
 price_purchase = price_unit * quant_purchase;
 printf( "Purchase price = %.2f\n", price_purchase );
} /* main */
```

5.

```
#include <stdio.h>
main()
{
 float amount, payment;

 printf( "Amount = " );
 scanf( "%f", &amount );
 printf( "Payment = " );
 scanf( "%f", &payment );
 if( payment < amount ) {
  printf( "IPayment is insufficient.\n" );
 } else {
  printf("Change = %.2f euros.\n", payment - amount );
 } /* if */
} /* main */
```

6.

```
#include <stdio.h>
#define RESERVE 10
main()
{
 int litres, distance, consumption;
 float consumption_average;
 printf( "Litres in tank = " );
 scanf( "%d", &litres );
 printf( "Average consumption per 100Km = " );
 scanf( "%f", &consumption_average );
 printf( "Distance to drive = " );
 scanf( "%d", &distance );
 consumption = consumption_average * (float) distance / 100.0;
 if( litres < consumption ) {
  printf( "You need %d Ltrs.\n", consumption-litres+RESERVE );
 } else {
  printf( "You can complete the journey.\n" );
 } /* if */
} /* main */
```

The structured programming

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Josep Anton Pérez López
Lluís Ribas i Xirgo

PID_00148429

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

An effective program is one which produces a legible code and which can be updated within a reasonable development time, it must also run efficiently.

Fortunately the compilers and interpreters of program code written in high-level languages will optimise programs to reduce the cost of execution. One example of this is that many compilers have the ability to use the same memory space for several variables, as long as they are not used simultaneously of course. In addition to this and the other memory optimisation functions, they can also improve the executable code to reduce the final execution time. For example, they can take advantage of common factors in expressions to avoid repeating calculations.

All this allows programmers to concentrate on the task of preparing legible programs which are easy to maintain. For example, giving variables long names poses no problems as the length of the name is not reflected on the compiled code. Similarly, it is not logical to use programming tricks which increase the efficiency of execution if these reduce the legibility of the source code. In general, programming tricks do not usually improve the execution cost significantly and will often make the program more difficult to maintain and may make it more dependent on a specific development environment or a particular machine.

This unit will therefore focus on how to organise the source code of a program. The correct organisation of programs implies a significant improvement in legibility and, as a consequence, a reduction in programming errors and easier maintenance and updating later on. What is more, this will make it easier to use parts of the program in other programs.

The first section will deal with structured programming. This programming paradigm is based on imperative programming, this imposes restrictions on the jumps which can be made during the execution of a program.

Note

A jump is a change in the order of execution of the instructions meaning the next instruction is not the one which comes after the instruction being executed at the time.

These restrictions can improve the legibility of the source code allowing readers to precisely determine the execution flow of the instructions.

In programming it is very common to find blocks of instructions which must be executed repetitively. As such it is necessary to understand how to set out the code in a structured way. Generally speaking, these cases derive from the need to process data. In the first section we will not just look at structured programming but also algorithmic structures which help us to write programs which handle a series of data strings and, of course, their corresponding instructions in C.

Even though structured programming is designed to reduce programming errors, these can never be eliminated completely. One section is therefore dedicated to the correction of programming errors and the tools that can help us to do this: debuggers.

The second section revolves around the logical organisation of program data. We must bear in mind that the information processed by computers and the results are often made up of several types of data. This data can in turn be composed of other simpler types.

Programming languages will support a certain amount of basic data types, meaning that they include mechanisms (declarations, operations and instructions) to allow the source code to use them.

As such, the information handled by a program must be represented in terms of variables which contain fundamental data types supported by the programming language. However, it is convenient to group together sets of data where the information they contain is closely related. For example: handling the number of days in each month of the year, the day, month and year of a date or a list of the holidays in a year as single units.

In the second section we therefore deal with those aspects of programming which improve the structural layout of data. In particular we will look at the existing structure classes and how to use variables containing structured data. We will also see how the definition of data types from other types, structured or not, benefits the program. Given that these new types are not recognised by the programming language, they are called *abstract data types*.

Note

An abstract data type is one which represents information not supported by the programming language used. This can occur if a certain data type is supported by one language but not by another, meaning they must be treated as being abstract.

The final section introduces the principles of modular programming, this is fundemental for understanding programming in C. In this programming model, the source code is divided into small structured programs which perform very specific tasks within the global program. This therefore divides the program into sub-programs which are much easier to read and understand. These sub-programs are called *modules*, which is where we get the name for this programming technique.

In C, all modules are functions which usually perform very specific actions on a few variables within the program. What is more, each function is usually specialised to operate on a specific type of data.

Given the importance of this subject, we will take a closer look at the declaration, definition and use of functions in C, especially the mechanisms used during the execution of programs to provide and obtain information on the functions they include.

Lastly, we will look at C preprocessor macros. These have a similar appearance to functions in C which can lead to confusion in the interpretation of the source code of the program that uses them.

The main purpose of this unit is that the reader learns to correctly organise the source code of a program as this is the fundamental indicator of the quality of the programming. More specifically, in this unit we are aiming to achieve the following goals:

1) To understand what structured programming is.
2) To learn how to correctly apply algorithmic layouts to handle sequences of data.
3) To identify the systems to be used for debugging errors in a program.
4) To learn about the basic data structures and how to use them.
5) To understand what modular programming is.
6) To understand the mechanics of the execution of functions in C.

1. Principles of structured programming.

Structured programming is a technique which arose from the analysis of the flow control structures which underlie all computer programs. This study revealed that it is possible to construct any flow control structure from three basic structures: sequential, conditional and iterative.

Additional reading

E.W. Dijkstra (1968). *The goto statement considered harmful*

E.W. Dijkstra (1970). *Notes on structured programming*

Structured programming consists of organising the code in such a way the flow of the execution of instructions is obvious to readers.

One theory formulated in 1966 by Böhm and Jacopini states that all "individual programs" should have a single input point and a single output point so that all instructions between these points are executable and there are no infinite loops.

The combination of these provide the groundwork for the construction of structured programs in which the flow control structures can be written using a very small number of instructions.

In fact, the sequential structure does not require any additional instructions as programs normally execute themselves by performing the instructions in the order in which they appear in the source code.

In the previous unit we looked at the `if` instruction which allows blocks of instructions to be executed conditionally. We must bear in mind that, for it to be an actual program, all of the instruction blocks must be able to be executed.

Iterative flow control structures will be dealt with in the following section. It is worth noting that, in terms of structured programming, we only need a single iterative flow control structure. This can be used to construct all the others.

2. Iterative instructions

Iterative instructions are flow control instructions which allow us to repeat the execution of a block of instructions. The following table shows those which are present in C:

Table 8.

instruction	Meaning
while(condition **) {** instructions **}** /* while */	This executes all the `instructions` in the looped block as long as the expression of the `condition` results in an integer-compatible data type with a value other than zero, meaning, as long as the condition is satisfied. The `instructions` could not be executed ever.

do {
instructions
} while (condition **);**
	Similarly to the `while`loop, this executes all the `instructions` in the looped block as long as the expression of the `condition` is satisfied. The difference lies in the fact that the `instructions` are executed at least one time (checking the condition for the possible repetition of the instructions is performed at the end of the block).

for(initialisation ; condition ; continuation **) {** instructions **}** /* for */	This behaves in a similar way to the while loop: as long as the condition is satisfied the instruction block is executed. In this case however, it is possible to indicate which instruction or instructions are to be executed before the start of the loop (`initialisation`) and which instruction or instructions are to be executed at the end of execution of the instructions (`continuation`).

As can be seen, all the loops can be reduced to a "while" loop. Despite this, there are cases in which it is more logical to use one of the variations.

We must remember that the flow control structure of a program in a high-level language does not reflect what the processor is actually doing (conditional and unconditional jumps) in the aspect of the flow control of the execution of the program. Even so, the C language has instructions which approximate the machine, such as the forced loop break (`break;`) and forced loop continuation (`continue;`). We also have an instruction for an unconditional jump (`goto`) which should never be used in a high-level program.

The programming of a loop normally implies determining the block of instructions to be repeated and, above all, under what conditions it is to be executed. As such, it is very important to remember that the condition governing the loop determines the validity of the repetition and, especially, the end of the loop when it is not satisfied. Note that there should be a case in which the solution of the condition expression results in a "false" value. If this is not so, the loop will repeat itself indefinitely (what we call an "infinite loop").

Having determined the iterative block and the condition governing it, we also need to program the possible preparation of the environment before the loop and instructions needed on its conclusion: its initialisation and end.

The iterative instruction should be selected based on the condition governing the cycle and its possible initialisation.

In cases where there is a possibility that the instructions of the loop are not executed we should use `while`. For example, to calculate the number of divisors of a given positive integer:

```
/* ... */
/* Initialisation: ___________________________*/
divisor = 1; /* Number to be divided */
ndiv = 0;    /* Number of divisors */
/* Loop: ____________________________________*/
while( divisor < number ) {
 if( number % divisor == 0 ) ndiv = ndiv + 1;
 divisor = divisor +1;
} /* while */
/* End: _____________________________*/
if( number > 0 ) ndiv = ndiv + 1;
/* ... */
```

Sometimes the condition governing the loop will depend on a variable which can be used as a repetition counter, meaning that its content reflects the number of iterations performed. In these cases we can consider using a `for`. This could be specifically interpreted as "iterating the following set of instructions for all the values of a counter between given initial and final values". The following example demonstrates this interpretation in C code:

```
/* ... */
unsigned int counter;
/* ... */
for( counter = START ;
 counter <= LAST ;
 counter = counter + INCREMENT
) {
instructions
} /* for */
/* ... */
```

Despite the fact that the above example is very common, the variable acting as the counter does not need to be incremented, nor does this need to be done by a specific step, nor does the condition only need to check that the last value has been reached or that it is an additional variable which is not used in the body of instructions to be iterated. It would therefore be very useful if it was a variable whose content was modified on each iteration and that it could therefore be used as a counter.

We recommend avoiding the use of `for` in cases in which there is no counter. Instead it is much better to use `while`.

In some cases a large part of the initialisation will coincide with the body of the loop, or we need to demonstrate that the loop will be executed at least once. If this is the case, it is better to use a `do...while`. As an example we will look at the code of a program which adds up various amounts until the amount read is equal to zero:

```
/* ... */
float total, amount;
/* ... */
total = 0.00;
printf( "SUM" );
do {
  printf( " + " );
  scanf( "%f", &amount );
  total = total + amount;
} while( amount != 0.00 );
printf( " = %.2f", total );
/* ... */
```

The real numerical constant `0.00` is used to indicate that only two fractional digits are significant as otherwise it would be the same thing to write `0.0` or even `0` (in the last case, the whole number would be converted into a real number before assignment).

In all cases the aim is always to produce a code which is understandable. The selection of the type of iterative instruction also depends on the stylistic taste of the programmer and his/her experience and it will not affect the efficiency of the program in terms of execution cost.

3. Processing data sequences

Much of the information handled will consist of sequences of data which can be either implicit or explicit.

Example

In the first case we are processing the information from a series of data coming from the standard input device.

An example of the second would be the processing of a series of values acquired by the same variable.

In both cases the treatment of the sequence can be observed in the program code as it must be performed by an iterative instruction. This loop usually corresponds to a specific algorithm. In the following sections we will look at the two fundamental layouts for the treatment of data sequences.

3.1. Algorithmic schemes: run through and searching

Algorithmic schemes for processing data sequences are patterns which are repeated frequently in many algorithms. We therefore have equivalent patterns in programming languages such as C. In this section we will look at some of the basic patterns for the treatment of sequences: the run-through and the search.

3.1.1. Run-through

Running through a sequence implies that all the members of the sequence are treated in the same way.

In other words, all of the elements of the sequence are treated, from the first to the last.

If the number of elements making up the series is known *a priori* and the initialisation for the loop is very simple, it may be appropriate to use a `for`. If not then the `while` or `do...while` loops are more appropriate, as long as we know that there will be at least one element in the data sequence.

The algorithmic scheme of running through a sequence in its version for C will be as shown below:

```
/* initialisation for sequence processing        */
/* (may include treatment of the first element)  */
while( ! /* end of sequence */ ) {
    /* treat element */
    /* advance through sequence */
} /* while */
/* end of sequence processing      */
/* (can include treatment of the last element)  */
```

The above pattern can be performed with other iterative instructions if circumstances require it.

To illustrate several examples of *run-through* patterns let us suppose we wish to find the average temperature at a weather station. To do this we will write a program and supply it with the temperatures recorded at regular intervals by the weather station's thermometer, this should then calculate the average of the values entered.

The body of the loop will therefore simply sum the temperatures (treatment of the element) and then read a new temperature (advancing the sequence):

```
   /* ... */
   accumulated = accumulated + temperature;
   amount = amount + 1;
   scanf( "%f", &temperature );
   /* ... */
```

In this iterative block we can observe that `temperature` should have a certain value before it can be summed in the `accumulated` variable, which in turn also needs to be initialised. Similarly, `amount` must be initialised to zero.

Once done, the initialisation and sequence preparation stage is complete:

```
   /* ... */
   unsigned int number;
   float accumulated;
   /* ... */
   number = 0;
   accumulated = 0.00;
   scanf( "%f", &temperature );
   /* loop ... */
```

We still need to resolve the problem of establishing the condition for ending the data sequence. The data sequence could be given an end marker, or its length could already be known.

In the first of these cases, the end marker should be a special element of the sequence which has a different value from any of the other data. As we know that a temperature reading can never be below –273.16 ºC (especially not an ambient temperature), we could use this value as the end marker. For clarity this marker will be defined as a constant in the preprocessor.

```
#define MIN_TEMP -273.16
```

When it is found it should not be processed but should end the *run-through* and cause the average to be calculated:

```
/* ... */
float average;
/* ... end of loop */
if( number > 0 ) {
  average = accumulated / (float) number;
} else {
  average = MIN_TEMP;
} /* if */
/* ... */
```

Before calculating the average it checks that there is data present. If there is not, the final marker temperature is assigned to average. With everything included, the final code would appear as follows:

```
/* ... */
number = 0;
accumulated = 0.00;
scanf( "%f", &temperature );
while( ! ( temperature == MIN_TEMP ) ) {
   accumulated = accumulated + temperature;
   amount = amount + 1;
   scanf( "%f", &temperature );
} /* while */
if( number > 0 ) {
 average = accumulated / (float) number;
} else {
   average = MIN_TEMP;
} /* if */
/* ... */
```

If the end marker of the sequence is provided separately, the iterative instruction should be a `do..while`. In this case, the input sequence is supposed to be made up of elements with two pieces of data: the temperature and an integer value taken as a logical value indicating if it is the last element:

```
/* ... */
```

```
number = 0;
accumulated = 0.00;
do {
   scanf( "%f", &temperature );
   accumulated = accumulated + temperature;
   amount = amount + 1;
   scanf( "%u", &the_last );
} while( ! the_last );
if( number > 0 ) {
   average = accumulated / (float) number;
} else {
   average = MIN_TEMP;
} /* if */
/* ... */
```

If we already know the number of temperatures (`NTEMP`) which have been recorded we only need to use a `for`:

```
/* ... */
accumulated = 0.00;
for( num = 1; num <= NTEMP; num = num + 1 ) {
   scanf( "%f", &temperature );
   accumulated = accumulated + temperature;
} /* for */
average = accumulated / (float) NTEMP;
/* ... */
```

3.1.2. Search

Searches are mostly partial *run-throughs* of sequences of input data. They run through the sequence of data until they find the one satisfying a certain condition. Obviously if they do not find an element satisfying the condition they will perform a complete *run-through* of the sequence.

In general, a search consists of running through a sequence of data until a certain condition is satisfied or it reaches the last element in the sequence. The condition does not need to affect only one element.

Following the above example, it is possible to perform a search which stops the *run-through* when the progressive average remains within ±1 °C or the detected temperature for at least 10 records.

This algorithmic scheme is very similar to the *run-through* except that it incorporates a search condition and that it will need to check if the search condition has been satisfied or not at the end of the loop:

```
/* initialisation for sequence processing        */
/* (may include treatment of the first element)  */
found = FALSE;
while( ! /* end of sequence */ && !found ) {
   /* treat element */
   if( /* found condition */ ) {
    found = TRUE;
   } else {
    /* advance through sequence */
   } /* if */
} /* while */
/* end of sequence processing      */
if( found ) {
   /* instructions */
} else {
   /* instructions */
} /* if */
```

In this scheme, the constants FALSE and TRUE are supposed to have been defined in the following way:

```
#define FALSE 0
#define TRUE 1
```

If we apply the above template to search for a stable progressive average, the source code will be as follows:

```
/* ... */
number = 0;
accumulated = 0.00;
scanf( "%f", &temperature );
sequential = 0;
found = FALSE;
while( ! ( temperature == MIN_TEMP ) && ! found ) {
   accumulated = accumulated + temperature;
   amount = amount + 1;
   average = accumulated / (float) number;
   if( average<=temperature+1.0 || temperature-1.0<=average ) {
    sequential = sequential +1;
   } else {
    sequential = 0;
   } /* if */
   if( sequential == 10 ) {
    found = TRUE;
   } else {
    scanf( "%f", &tempera
   } /* if */
} /* while */
/* ... */
```

In search schemes it is not usually appropriate to use a `for`, as this is usually an iterative instruction which uses a counter to take a series of values from the first to the last. This means that it performs a *run-through* of the implicit sequence of all the values taken by the counting variable.

3.2. Filters and pipes

Filters are programs which generate a sequence of data from a *run-through* of a sequence of input data. Usually the output data sequence contains the processed data from the input.

The name *filter* is used because it is very common that the output sequence is simply a data sequence which is very similar to the input sequence but where some of the elements have been removed.

For example, a filter would be a program whose output is the partial sums of the input numbers:

```
#include <stdio.h>
main()
{
   float sum, summing;
   sum = 0.00;
   while( scanf( "%f", &summing ) == 1 ) {
   sum = sum + summing;
   printf( "%.2f ", sum );
  } /* while */
} /* main */
```

Another more useful filter could be a program which replaces tabs by the number of blank spaces required to reach the next tab column:

```
#include <stdio.h>
#define TAB 8
main()
{
   char character;
   unsigned short position, tab;
   position = 0;
   character = getchar();
```

```
    while( character != EOF ) {
     switch( character ) {
      case '\t':/* advance to the next column */
       for( tab = position;
          tab < TAB;
          tab = tab + 1 ) {
        putchar( ' ' );
       } /* for */
       position = 0;
       break;
      case '\n': /* new line means column 0 */
       putchar( character );
       position = 0;
       break;
      default:
       putchar( character );
       position = (position + 1) % TAB;
     } /* switch */
     character = getchar();
    } /* while */
 } /* main */
```

These small programs can be useful either by themselves or combined with others. As such, the output sequence of one can be used as the input sequence for another, making up what we call a ***pipe*** the visual idea is that we enter data into one end of the pipe and we get another flow of processed data at the other end. The pipe can include one or more filters which retain and/or transform the data.

Example

A filter can be used to convert a sequence of input data consisting of three numbers (item code, price and amount) to an output data sequence with two numbers (code and amount), the next one could be a sum filter to produce the total amount.

To do this we will need the help of the operating system. It is not necessary for the data to be input using the keyboard nor is it necessary for it to be output using the screen, even though they are the standard input and output devices. In Linux (and other OS types) we can redirect the standard input and output of data using the redirection commands. In this way we can make a specific file the standard input and we can store the output data in another file which is used as the standard output.

In the above example we can suppose that there is a file (`ticket.dat`) containing the input data and we wish to obtain the total of the purchase. To do this we can use a filter to calculate the partial amounts, whose output will be the input of another which produces the total.

In order to apply the first filter we will need to execute the corresponding program (which we will call `calculate_amounts`) redirecting the standard input to the file `ticket.dat`, and the output to the file `amounts.dat`:

```
$ calculate_amounts <ticket.dat >amounts.dat
```

Thus, `amounts.dat` will collect the sequence of data pairs (item code and amount) which the program has generated at the standard output. Redirections are determined using the "less than" symbol for the standard input and the "more than" symbol for the standard output.

If we want to calculate the amounts of other purchases in order to calculate the sum, we would need to add all of the partial amounts of all of the purchase tickets to `amounts.dat` . This can be done using a double output redirection operator, the meaning of this could be "add to the standard output file of the program":

```
$ calculate_amounts <other_ticket.dat >>amounts.dat
```

Once we have collected all the partial amounts we want to add up, we can call up the program which calculates the sum:

```
$ sum <amounts.dat
```

If we are only interested in the sum of one purchase ticket we can build a pipe in which the output of the calculation for the partial amounts is the input for that of the sum:

```
$ calculate_amounts <ticket.dat | sum
```

As we can see in the above command, the pipe has been assembled with the pipe operator represented by the vertical bar symbol. The data from the standard output of the execution of that preceding it transmits them to the standard input of the program coming after.

4. Debugging programs

The removal of errors from a program is known as debugging. Errors can be caused by both the programming and by the algorithm itself. Debugging can therefore involve modifications to the algorithm causing the problem. When an error is caused by the incorrect programming of an algorithm this is known as a **logic error**. If the error is caused by a violation of the rules of the programming error it is called a **syntax error** (although errors can also be lexical or semantic in nature).

Note

Debugging is the term used in English to refer to the removal of computer programming errors. This verb can be translated as "bug removal" and has its origin in a 1945 report on a test of a Mark II computer performed at Harvard University. The report showed that a moth was found in one of the relays causing it to malfunction. To prove that the bug had been removed (and the error resolved) the moth was included in the actual report. It was attached using adhesive tape and a footnote was added saying "first case of a moth found". This was also the first appearance of the verb to debug which is still used today.

Syntax errors are detected by the compiler as they prevent an executable code being generated. If the compiler can still generate the code despite the possible existence of an error it will usually generate a warning.

For example, it may be that an `if` expression contains an assignment operator, it is usually a confusion of assignment and comparison operators:

```
/* ... */
if( a = 5 ) b = 6;
/* ... */
```

The above code also shows a programming error given that the instruction appears to indicate that `b` may not be 6. If we look at the `if` condition, it assigns the value of 5 to the variable `a`, the result being equal to the assigned value. As the value 5 is not 0, the result is always positive and consequently `b` always takes the value 6.

For this reason it is highly recommendable that the compiler gives us all the warnings it can. It should be executed using the following argument:

```
$ gcc -Wall -or program program.c
```

The argument `-Wall` indicates that a warning will be given in most cases in which there may be a logic error. Despite the fact that the argument appears to indicate that it will warn us in all situations, there are still some situations in which it will not.

The most difficult errors are logic errors which can escape detection even by the compiler. These errors are caused by incorrect programming of the corresponding algorithm, meaning that the algorithm itself is incorrect. Once an error has been detected it must be located in the source code.

In order to locate an error we need to know under what condition or state of the environment they occur. As such, we need to find out which values of the variables lead the execution flow of the program to the instruction in which the error occurs.

Unfortunately, errors usually present themselves in a later state to the one in which they actually caused a fault in the behaviour of the program. To detect a fault it is therefore necessary to observe the state of the program at all times and to follow its evolution until the error presents itself.

To increase the ***intelligibility*** of a program, we usually insert text printings (also called *notes*) into the code to show us the content of certain variables. This procedure implies the modification of the program each time new *notes* are added, those which are no longer needed can be removed or changed.

However, in order to improve error localisation we need to control the flow of execution. The ***controllability*** implies the ability to modify the content of the variables and to choose between different execution flows. In order to achieve a certain level of control we need to make significant changes to the program under examination.

Instead of doing all the above steps, it is far easier to use a tool which allows us to observe and control the execution of programs for debugging. These tools are called *debuggers*.

For the debugger to be able to do its job, we need to compile programs in such a way that the resulting code includes information relating to the source code. Therefore, to debug a program we need to compile it using the option `-g`:

```
$ gcc -Wall -or program -g program.c
```

In GNU/C there is a debugger called `gdb` , this allows us to execute a program, to stop it under certain conditions, to examine the state of the program once stopped and, lastly, make changes to find possible solutions.

The debugger is invoked in the following way:

```
$ gdb program
```

The following table shows some of the commands we can use with GDB:

Table 8.

Command	Action
`run`	Executes the program from the first instruction. The program will only stop at a breakpoint, when the debugger receives a stop warning (using the Ctrl and the C key simultaneously) or when it is waiting for data to be input.
`break line_num`	This establishes a breakpoint before the first instruction found at the line indicated in the source code. If the line number is omitted it will be established at the first instruction of the current line, meaning the line at which it stopped.
`clear line_num`	This removes the breakpoint established at the indicated line or at the current line if the number is omitted.
`c`	Continues execution after a stoppage.
`next`	Executes the following instruction and stops.
`print expression`	Prints the result of solving the indicated expression. The expression can be a variable and the result of its resolution, its content.
`help`	Shows the list of commands.
`quit`	Exits the GDB

Breakpoints are flags in the executable code which allow the debugger to know if it should stop the execution of the program or if it should continue to execute it. These flags can be set or removed by using the debugger itself. This allows you to execute portions of the code as a unit.

It can be particularly useful to insert a breakpoint in `main` before executing it, this will make it stop at the first instruction and allow us to track execution better. To do this we use the command `break main`, as it is possible to indicate function names as breakpoints.

It is always much more practical to use a graphics environment in which the source code can be seen at the same time as the output of the program being executed. We could use DDD to do this for example (*Data Display Debugger*) or the XXGDB. Both environments use the GDB as the debugger and therefore have the same options. However it is easier to use because most of the commands are visible and we also have the drop-down menus available.

5. Data structures

The basic data types (real and integer-compatible) can be grouped into homogeneous and heterogeneous structures to facilitate (and clarify) access to elements within a program.

A homogenous structure is one in which all the data is of the same type, a heterogeneous one may be made up of data of different types.

In the following sections we will review the main data structures in C, although they exist in all structured programming languages. Each section is organised in such a way to allow us to see how to perform the following operations on the variables:

- Declare them so that the compiler can reserve the necessary space.
- Initialise them so that the compiler gives them an initial content (which can be changed) in the resulting executable program.
- Reference them so that their content can be accessed, both for modifying and reading them.

As it is obligatory to precede the variable by its type, it is recommendable to identify the types of structured data with a type name. These new data types are known as *abstract data types*. The last section will deal with these.

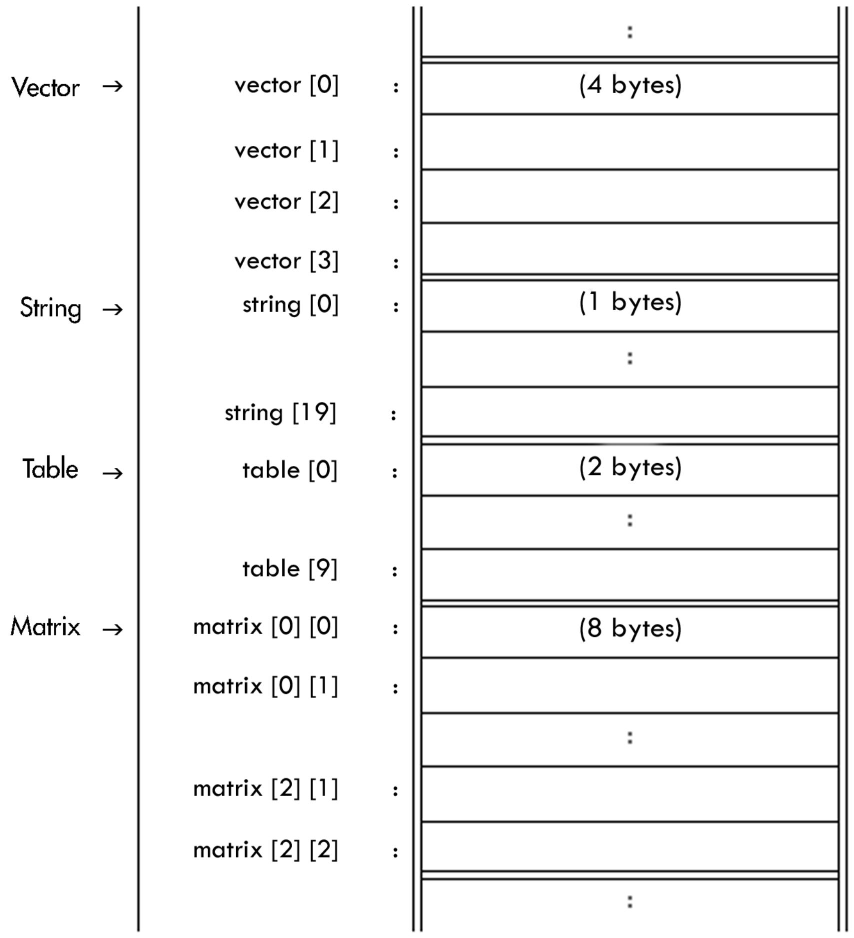
6. Matrices

Matrices are data structures of a fixed size. This means they always represent a piece of information using a certain amount of data. These are also called **arrays**, **tables** (for those with one or two dimensions) or **vectors** (if they only have one dimension). In the particular case of character vectors, they take the name **character strings** or *strings*.

6.1. Declaration

We will now look at four declarations of different matrices and a scheme for their distribution in the memory of the computer. The number of bytes in each division will depend on the data type used by the matrix. The scheme will include the name of each piece of data within the matrix, this will distinguish the common name of the matrix from the identifier of the particular piece of data, which will correspond to the position of the element within the matrix. It is very important to bear in mind that in C positions are always numbered starting from 0.

Figure 2.



This shows the declarations of the variables which lead to the distribution in memory shown in Fig. 2:

```
       int             vector[4];
       char            string[20];
       unsigned short  table[10];
       double          matrix[3][3];
```

The first of these declarations prepares a vector of 4 signed integers; the second, a string of 20 characters; the third, a table of 10 positive integers and the last reserves a space for a 3 x 3 matrix of real numbers with double precision.

Note

The matrix is stored in memory by rows, meaning the first row appears first followed by the second and so on up to the last.

If we need to declare a matrix with a higher number of dimensions, we just need to include its size in square brackets between the name of the structure and the semi-colon.

As we have already mentioned, character strings are in fact just single-dimensional matrices in C. This means that they have a maximum length fixed by the space reserved in the corresponding vector. Even so, character strings can vary in length and an end flag is used to do this. In this case it will be the NULL character in ASCII code, whose numerical value is 0. In the above example, the string can contain any text up to 19 characters in length, since we need to make sure that the last character is the end of string (\0').

In all cases, and especially when we are dealing with variables which must contain constant values, we can give each of the elements they contain an initial value.

We need to remember that matrices are stored in rows in memory and that it is possible to not specify the first dimension (that which appears immediately after the name of the variable) of a matrix. In this case it will take the dimensions needed to hold the data given at initialisation. The other dimensions must be fixed in such a way that each element of the first dimension occupies a known amount of memory.

In the following examples we will look at different ways of initialising the declarations of the above variables.

```
  int             vector[4] = { 0, 1, 2, 3 };
  char string[20] = { 'H', 'e', 'l', 'l', 'o', '\0' } ;
  unsigned short table[10] = { 98, 76, 54, 32, 1, };
  double          matrix[3][3] = {{0.0, 0.1, 0.2},
                                 {1.0, 1.1, 1.2},
                              {2.0, 2.1, 2.2} };
```

In the case of the character string, elements coming after `\0`' will not have an initial value. What is more, they can have any value. This is therefore an incomplete initialisation.

In order to make it easier to initialise character strings, we can also do it in the following way:

```
char string[20] = "Hello";
```

Furthermore, if this string does not need to be changed we can use the fact that we do not need to specify the dimension if it can be calculated from the initialisation performed for the corresponding variable.

```
char string[] = "Hello";
```

In the case of a `table`, we perform a complete initialisation by using the last comma to indicate that all subsequent elements will have the same value as the last one given.

6.2. Reference

In order to reference an element of a matrix in a particular expression we need to indicate its name and the position it occupies:

$$\text{matrix}[i_0][i_1]...[i_n]$$

where i_k are expressions for which the result must be an integer value. The expressions are usually very simple: one variable or one constant.

For example, to read the data for the 3 x 3 double real matrix from the standard input which we declared above, we could write the following programs in which the variables `row` and `column` are obviously positive integers:

```
/* ... */
for( row = 0; row < 3; row = row + 1 ) {
   for( column = 0; column < 3; column = column + 1 ) {
    printf( "matrix[%u][%u]=? ", row, column );
    scanf( "%lf ", &data );
    matriz[row][column] = data;
   } /* for */
} /* for */
/* ... */
```

It is important to bear in mind that the C compiler does not add code to check the validity of the matrix indexes. As such, the matrix limits are not checked and we can reference any element whether it belongs to the matrix or not. This is always the responsibility of the programmer!

What is more, in C, the square brackets are operators for accessing homogenous data structures (or matrices) which calculate the position of an element from the memory address in which they are found and the argument which gives them. This implies that it is possible to access one column of a square matrix (for example: `int A[3][3];`) indicating only its first index (for example: `pcol = A[0];`). Furthermore, it is possible to commit the error of referring to an element in the form of `A[1,2]` (common in other programming languages). In this case the compiler will accept the reference as it is a valid way of accessing the last column of the matrix, since the comma is a concatenation operator for expressions, the result of which is that of the last expression solved, so in the above example, `A[1,2]` would in fact be, `A[2]`.

6.3. Examples

In the first example, the program checks to see if a word or phrase is a palindrome, i.e. if it reads the same backwards as forwards.

Example

One of the most well-known palindromes is the following: A man, a plan, a canal, Panama!

```
#include <stdio.h>
#define LENGTH 81
#define NULL     '\0'
main()
{
   char          text[LENGTH];
   unsigned int  length, left, right;
   printf( "Palindrome checker.\n" );
   printf( "Enter text: ");
   gets( text );
   length = 0;
   while( text[length] != NULL ) {
    length = length + 1;
   } /* while */
   left = 0;
   right = length;
   while( ( text[left] == text[right] ) && ( left < right ) ) {
    left = left +1;
    right = right - 1;
   } /* while */
   if( left < right ) {
    printf( "This is not a palindrome.\n" );
   } else {
    printf( "This is a palindrome!\n" );
   } /* if */
} /* main */
```

As `gets` uses the reference of the whole character string as an argument, meaning the address of the initial position it occupies in memory, we do not need to use the operator "address of".

The following program stores the coefficients of a polynomial in a vector for later evaluation. The polynomial has the following form:

$$P(x) = a_{\text{MAX\_DEGREE-1}} x^{\text{MAX\_DEGREE}} + \ldots + a_2 x^2 + a_1 x + a_0$$

Polynomials would be stored in a vector in the following way:

$$\texttt{a[MAX\_DEGREE-1]} = a_{\text{MAX\_DEGREE-1}}$$

$$: \quad :$$

$$\texttt{a[2]} = a_2$$

$$\texttt{a[1]} = a_1$$

$$\texttt{a[0]} = a_0$$

The program should evaluate the polynomial for a certain χ using the Horner method in which the polynomial is handled as if it was expressed in the form:

$$P(x) = (\ldots (a_{\text{MAX\_DEGREE-1}} x + a_{\text{MAX\_DEGREE-2}})x + \ldots + a_1)x + a_0$$

As such, the highest level coefficient is multiplied by χ and the coefficient of the preceding degree is added to it. The result is again multiplied by χ as long as it has not reached the independent term in the process. If it has then we have obtained the final result.

Note

This method reduces the number of operations that will need to be performed as we do not need to calculate any power of *x*.

```
#include <stdio.h>
#define MAX_DEGREE 16
main()
{
   double a[MAX_DEGREE];
   double x, result;
   int    degree, i;

   printf( "Evaluation of polynomials.\n" );
   for( i = 0; i < MAX_DEGREE; i = i + 1 ) {
    a[i] = 0.0;
   } /* for */
   printf( "maximum degree of the polynomial = ? ");
```

```
     scanf( "%d", &degree );
     if( ( 0 <= degree ) && ( degree < MAX_DEGREE ) ) {
      for( i = 0; i <= degree; i = i + 1 ) {
       printf( "a[%d]*x^%d = ? ", i, i );
       scanf( "%lf", &x);
        a[i] = x;
       } /* for */
       printf( "x = ? " );
       scanf( "%lf", &x);
       result = 0.0;
       for( i = degree; i > 0; i = i - 1 ) {
        result = x * result + a[i-1];
       } /* for */
       printf( "P(%g) = %g\n", x, result, x) );
      } else {
       printf( "The degree must be between 0 and %d!\n",
        MAX_DEGREE-1
       ); /* printf */
     } /* if */
  } /* main */
```

We recommend you to program these examples to gain practice in programming with matrices.

7. Heterogeneous structures

Heterogeneous data structures are able to contain data of different types. They are generally groups of data (tuples) which form a logical unit with respect to the information processed by the programs which use them.

Example

Dates (day, month, year), personal information (name, surname, address, town etc.) entries in telephone directories (number, user, address) and many others.

7.1. Tuples

Tuples are groups of different data types. Each element within a tuple is identified by a specific field name. In C these tuples are called *structures* (`struct`).

Similarly as with matrices, they are used to organise data from a logical point of view. This logical organisation implies that we can treat groups of closely-related data as a single unit. This means that programs which use them will acknowledge the relationship between them and are therefore much more intelligible and less prone to errors.

Note

We can achieve much more clarity if we use a tuple to describe a date than if we use three distinct integers (day, month, year). Furthermore, references to the date fields include the fact that they are a part of the tuple, this does not happen when using independent variables.

7.1.1. Declaration

Declarations of heterogeneous structures or tuples in C must begin with `struct`, this must be followed by a block of declarations of the variables which belong to the structure followed by the name of the variable or those of a list of variables which contain data of the type being declared.

Given that the procedure we have just described must be repeated for the declaration of other identical tuples, we need to put a name (between `struct` and the field declaration block) for the declared structures. In doing this we only need to include the declaration of the fields of the structure in the first variable of this type. For the rest it is enough to specify the name of the structure.

The names of heterogeneous structures usually make them easier to identify. In this case we will use one of the most common: add `"_s"` to the name as a postfix.

The following example describes how we could relate a data structure to the location of a plane on radar at an air traffic control centre and the corresponding variable (`plane`). As can be observed, the declaration of the fields of the same is not repeated in the subsequent declaration of a vector for these structures to contain the information of up to MAXNAV planes (we assume that this is the maximum number of planes within range and that it has been defined beforehand):

```
struct plane_s {
      double    radius, angle;
      double    height;
      char      name[33];
      unsigned  code;
} plane;
struct plane_s planes[MAXNAV];
```

We can also give initial values to the structures using an assignment at the end of the declaration. The values of the different fields must be separate using commas and be placed between brackets:

```
struct person_s {
   char   name[ MAXLENGTH ];
   unsigned short age;
} person = { "Carmen" , 31 };
struct person_s winner = { "unknown", 0 };
struct person_s people[] = { { "Eva", 43 },
                             { "Pedro", 51 },
                             { "Jesús", 32 },
                             { "Anna", 37 },
                             { "Joaquín", 42 }
                             }; /* struct person_s people */
```

7.1.2. Reference

Referencing a specific field of a tuple is done using the name of the field after the name of the variable containing it, the two must be separated using a structure field access operator (a full stop).

Example

```
winner.age = 25;
initial =
people[i].name[0];
```

In the following program we will use structured variables which contain two real numbers to indicate a Cartesian point on a plane (`struct cartesian_s`) and polar (`struct polar_s`). The program will ask for the Cartesian coordinates of a given point and then it will transform them into polar coordinates (angle and radius, or distance from the origin). Observe that two variables are declared using direct initialisation: `prec` indicates the precision we are working at and `pi` stores the value of the constant Π with the same precision.

```
#include <stdio.h>
#include <math.h>
```

```
main()
{
   struct cartesian_s { double x, y; } c;
   struct polar_s { double radius, angle; } p;
   double prec = 1e-9;
   double pi = 3.141592654;
   printf( "Cartesian coordinates to polar coordinates.\n" );
   printf( "x = ? "); scanf( "%lf", &(c.x) );
   printf( "y = ? "); scanf( "%lf", &(c.y) );
   p.radius = sqrt( c.x * c.x + c.y * c.y );
   if( p.radius < prec ) { /* if the radius is zero ... */
    p.angle = 0.0; /* ... the angle is zero. */
   } else {
    if( -prec<c.x && c.x<prec ) { /* if c.x is zero ... */
     if( c.y > 0.0 )p.angle = 0.5*pi;
     elsep.angle = -0.5*pi;
    } else {
     p.angle = atan( c.y / c.x );
    } /* if */
   } /* if */
   printf( "radius = %g\n", p.radius );
   printf( "angle = %g (%g degrees sexagesimal)\n",
    p.angle,
    p.angle*180.0/pi
   ); /* printf */
} /* main */
```

The above program makes use of the standard mathematical functions `sqrt` and `atan` to calculate the square root and the arc tangent respectively. This means that we need to include the corresponding header files (`#include <math.h>`) in the source code.

7.2. Multiple type variables

These are variables whose content can vary between different data types. The data type must be one of those indicated in its declaration and the compiler will reserve space for the one which has the largest size. Their declarations are similar to those for tuples.

Example

```
union number_s {
 signed integer;
 unsigned natural;
 float real;
} number;
```

The use of this class of variable can allow us to save space. However, we have to remember that to manage these variable type fields we need to have information (explicit or implicit) on the type of data being stored in them at a certain time.

As such, they are often combined in tuples which contain a field which will show the type of data they contain. For example, look at the declaration of the following variable (`insurance`), in which the field `good_type` shows us which multiple-type structures are present in its content.

```
struct insurance_s {
   unsigned   policy;
   char       holder[31];
   char       NIF[9];
   char       asset_type;   /* 'C': residence, */
                            /* 'F': life, */
                            /* ''M': vehicle. */
   union {
    struct {
     char ref_catastro[];
     float area;
    } residence;
    struct {
     struct date_s birth;
     char beneficiary[31];
    } life;
    struct {
     char registration[7];
     struct date_s manufacture;
     unsigned short accidents;
    } vehicle;
   } information;
   unsigned value;
   unsigned premium;
} insurance;
```

Using this we can include information about a number of insurance policies on the same table no matter what type of policies they are.

```
struct insurance_s insured[ NUMINSURANCE ];
```

In all cases, the data type `union` is only used infrequently.

8. Abstract data types

Abstract data types are data types which are attributed with a meaning relating to the problem we wish the program to solve, they therefore have a higher level of abstraction than the computational model. These data types are therefore transparent to the compiler and subsequently irrelevant in the corresponding executable code.

In practice, any defined data structure is in fact a group of data relating to the problem and, as such, an abstract data type. An integer which is used for a different purpose may also be one, for example one which is used to store logical values ("true" or "false").

In all cases, the use of abstract data types will improve the legibility of programs (among other things we will look at later). They also allow us to use declarations of the types described above without having to repeat a part of the declaration as we only need to indicate the name which was assigned to it.

8.1. Defining abstract data types

To define a new data type name we just need to declare a variable preceded by `typedef`. In this declaration the name of the variable will in fact be the name of the new data type. Several examples are shown below.

```
typedef char boolean, logical;
#define MAXSTRLEN 81
typedef char string[MAXSTRLEN];
typedef struct person_s {
   string   name, address, town;
   char     post_code[5];
   unsigned telephone;
} person_t;
```

In the above definitions we can observe that the syntax does not vary with respect to the declaration of the variables except for the inclusion of the key word `typedef`, this is short for "type definition". Using these definitions we are now ready to declare variables of the corresponding types:

```
boolean   correct, ok;
string    teacher_name;
person_t students[MAX_GROUP];
```

We always recommend using a type name which identifies the content of the variable in a meaningful way in the problem that we wish the program to resolve.

From this point on, all the examples will use abstract data types when necessary.

In the context of this book we prefer that type names always end in "_t".

8.2. Enumerated types

Enumerated data types are an integer-compatible data type in which an integer and a certain symbol are associated (the enumeration constant). In other words, it is an integer data type in which a set of values are given a name (enumerated). Its use can replace the symbolic constant definition command in the preprocessor (#define) when these are composed of integers.

The following example shows how they are declared and how they can be used:

```
/* ... */
enum { RED, GREEN, BLUE } rgb;
enum bool_e { FALSE = 0, TRUE = 1 } logical;
enum bool_e found;
int colour;
/* ... */
rgb = GREEN;
/* An enumerate can be assigned to an integer:           */
colour = RED;
logical = TRUE;
/* An integer can be assigned to an enumerate,           */
/* even if it does not have a symbol associated with it: */
found = -1;
/* ... */
```

The enumerated variable `rgb` can contain any integer value (of type `int`), but there will be three integer values identified by the names `RED`, `GREEN` and `BLUE`. If the value associated with the symbols does matter, we must assign a value to each symbol using the equals sign, as shown in the declaration for `logical`.

The enumerated type can have a specific name (`bool_e` in the example) which avoids repeating the enumerate in a subsequent declaration of a variable of the same type (in the example: `found`).

It is also possible and also advisable to define an associated data type:

```
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
```

In this particular case we use the name `bool` instead of `bool_t` or `logical` as it matches the name of the primitive data type in C++. Given that it is used very often, it will be considered to be defined in the rest of the text. (However, we do need to remember that a variable of this type can acquire values other than 1 and be, conceptually `TRUE`.)

8.3. Example

In the programs we have already looked at, we have used variables of structured data types which are (or may be) frequently used in other programs of the same nature. It is therefore convenient to convert structured data type declarations to type definitions.

In particular, the program for evaluating polynomials using the Horner method should have a structured data type which represents the information of a polynomial (maximum degree and coefficients).

The program shown below contains a data type definition `polynomial_t` to identify its components as data within the same polynomial. The maximum degree is also used to find out which elements of the vector contain the coefficients for each degree and which do not. This program performs the symbolic derivation of a given polynomial (symbolic derivation implies obtaining another polynomial which represents the polynomial function given as the input).

```
#include <stdio.h>

#define MAX_DEGREE 16

typedef struct polynomial_s {
   int     degree;
   double a[MAX_DEGREE];
} polynomial_t;
main()
{
  polynomial_t p;
  double      x, coeff;
  int         i, degree;
  p.degree = 0; /* initialisation of (polynomial_t) p    */
  p.a[0] = 0.0;
  printf( "Symbolic derivation of polynomials.\n" );
  printf( "Degree of polynomial =  " );
  scanf( "%d", &(p.degree) );
```

```
    if( ( 0 <= p.degree ) && ( p.degree < MAX_DEGREE ) ) {
     for( i = 0; i <= p.degree; i = i + 1 ) { /* reading      */
      printf( "a[%d]*x^%d = ? ", i, i );
      scanf( "%lf", &coeff );
      p.a[i] = coeff;
     } /* for */
     for( i = 0; i < p.degree; i = i + 1 ) { /* derivation    */
      p.a[i] = p.a[i+1]*(i+1);
     } /* for */
     if( p.degree > 0 ) {
      p.degree = p.degree -1;
     } else {
      p.a[0] = 0.0;
     } /* if */
     printf( "Derived polynomial:\n" );
     for( i = 0; i < p.degree; i = i + 1 ) { /* printing      */
      printf( "%g*x^%d +", p.a[i], i );
     } /* for */
     printf( "%g\n", p.a[i] );
    } else {
     printf( "The degree of the polynomial must be" );
     printf( " between 0 and %d!\n", MAX_DEGREE-1 );
    } /* if */
  } /* main */
```

9. Files

A file is a homogenous data structure which is different in that the data is stored outside of the main memory. These are therefore data structures which are found in an external or secondary memory (however it is possible that some temporary files are only to be found in the main memory).

Example

Floppy drives, hard disks, CD's, DVD's, memory cards etc.

To access the data in a file, the computer must have the appropriate devices available and be able to read them and, optionally, write data to the appropriate media.

Given that they reside on permanent information media, they can maintain information between several executions of a single program or serve as a source or repository for information for any program.

Given the capacities of these devices, the size of the files can be much larger, even larger than the main memory held in the computer. For this reason, the main memory will only hold a part of the content of the file in use and the information needed for handling it.

No less important is that files are data structures of an undefined number.

Over the next few sections we will look at aspects relating to files in C, these are known as byte streams. These files are simple homogenous data structures in which each piece of data is a single byte. There are usually two types: ASCII text files (each byte is a character) and binary files (each byte matches a byte which forms part of a piece of data of one of the data types which exist).

9.1. Byte stream files

Files of the *byte stream* type in C are sequences of bytes which can be considered to be either a copy of the memory content (binary) or as a character string (textual). In this section we will mostly look at the latter as they are more common.

Given that they are stored on an external device, we need to have information about them in the main memory. This means all the information needed to control a file of this type and part of the data it contains (or which it will need to contain if it is written) will be contained in a variable of the `FILE`.

The FILE data type is a tuple composed of, among other fields, the name of the file, its length, the position of the last byte read or written and a *buffer* (temporary memory) which contains BUFSIZ file bytes. This is needed to prevent access to the affected peripheral device and, given that read and write operations are done in blocks of bytes, to perform them more rapidly.

Fortunately there are standard functions available to perform all the operations we have just mentioned. As with the FILE structure and the BUFSIZ constant, these are declared in the file stdio.h. In the next section we will look at the most common ones.

9.2. Standard file functions

To access the information in a file it must first be "opened". This means that we have to localise and create a variable of the FILE. When the open function is executed it will return the memory address of the variable it creates or NULL if it has not been able to open the file indicated.

When a file is opened we need to specify if its content is to be read (opening_mode = "r"), if we need to add more data (opening_mode = "a"), or if we want to create it again (opening_mode = "w").

Note

We have to be careful with the third of these: If the file already exists we will lose its content!

We should also indicate whether the file is a text file (the ends of lines can be transformed lightly) or whether it is binary. This is done by adding a "t" or a "b"respectively to the opening mode. If we omit this information the file will open in text mode.

Once opened, we can either read the content or write new information to it. Once we have finished, we need to close it. This means that we need to tell the operating system that we are not going to work on it any more and any pending information needs to be written to the corresponding *buffer* for each one. All this is done by the standard file closing function.

The following code shows the algorithmic scheme for working with files and also details the file re-opening function which uses the same control data structure. Bear in mind that we must have already closed the above-mentioned file:

```
/* ... */
/* We declare a variable to contain                          */
/* the reference for the FILE structure:                     */
FILE* file;
/* ... */
file = fopen( file_name, opening_mode );
/* The opening mode could be "r" for read,                   */
/* "w" for write, "a" for add or                             */
/* "r+", "w+" or "a+" for updating (read/write).             */
/* We can add the suffix                                    */
/* "t" for text or "b" for binary.                          */
if( file != NULL ) {
  /* Treatment of the file data.                            */
  /* Possible re-opening of the same file:                  */
  file = freopen(
         file_name
         opening_mode
         file
        ); /* freopen */
  /* Treatment of the file data.                            */
  fclose( file );
} /* if */
```

The next sections will deal with the standard files for working with streaming files or *streams*. The variables which are used in the examples are of the appropriate type and, in particular, `stream` is of the `FILE*` type, meaning that it refers to a file structure.

9.2.1. Standard data output functions (write) for files

These functions are very similar to those we have already seen for reading data from the standard input. However with these it is very important to know if we have reached the end of the file and that there is therefore no more data to read.

```
fscanf( stream, "format" [,list_of_&variables])
```

Functioning in a similar way to `scanf()`, it returns the number of arguments actually read as a result. It therefore provides an indirect way of determining if we have reached the end of the file. In this case it will activate the end of file condition. In fact, a lower number of assignations can be simply due to an unexpected input such as, for example, the input of an alphabetical character for a `"%d"`.

Conversely, this function returns `EOF` (*end of file*) if it has reached the end of the file and has not been able to perform an assignation. It is therefore much more convenient to use the function which checks this condition before indirectly checking using the number of correctly-read parameters (the file may contain more data) or through the return of `EOF` (will not occur if at least one piece of data has been read).

Example

```
fscanf( stream, "%u%c", &num_dni, &letter_nif );
fscanf( stream, "%d%d%d", &code, &price, &amount );
```

```
feof( stream )
```

Returns 0 if the end of file has not been reached. If it has, it will return a number other than zero, meaning that the end of file condition is true.

```
fgetc( stream )
```

This reads a character from the stream. If it can not read a character as the end of file has been reached it will return `EOF`. This constant is already defined in the header file `stdio.h;` and can therefore be freely used in the code.

Note

It is important to remember that some files may have an `EOF` character in the middle of the stream as the end of file is determined by its length.

```
fgets( string, maximum_length, stream )
```

This reads a character string from the file until it finds an end of line, until it reaches `maximum_length` (–1 for the end of string marker) of characters, or up to the end of file. It returns `NULL` if it finds the end of file during reading.

```
if( fgets( string, 33, stream ) != NULL ) puts( string );
```

9.2.2. Standard data output functions (write) for files

These functions also behave in a similar manner to the functions for outputing data to the standard device. They all write characters in the indicated output stream:

```
fprintf( stream, "format" [, list_of_variables] )
```

The `fprintf()` function writes characters to the indicated formatted output stream. If a problem occurs, the function will return the last character written or the `EOF`.

```
fputc( character, stream )
```

The `fputc()` function writes characters to the indicated output stream character by character. If a writing error occurs or the media is full, the `fputc()` function activates a file error indicator. This indicator can be checked using the `ferror( stream )`function which returns a zero (false logic value) if there is no error.

```
fputs( string, stream )
```

The function `fputs()` writes characters to the indicated output stream allowing complete strings to be recorded. If a problem occurs, this function acts in a similar way to `fprintf()`.

9.2.3. Standard stream file position functions

With stream files we can determine the read or write position, i.e. the position of the last byte read or written. This is done using the `ftell(stream)` function which returns a `long` integer indicating the position or -1 if there has been an error.

There are also functions for changing the read position (and the write position if the files are to be updated):

```
fseek( stream, displacement, address )
```

This displaces the read/write "header" with respect to the current position by the value of the long integer indicated by `displacement` if `address` is equal to `SEEK_CUR`. If this address is `SEEK_SET`, then `displacement` becomes a displacement with respect to the beginning and, as such, indicates the final position. On the other hand, if it is `SEEK_END`, it will indicate the position with respect to the last position of the file. If repositioning is correct, it will return 0.

```
rewind( stream )
```

This locates the "header" at the beginning of the file. This function is equivalent to:

```
seek( stream, 0L, SEEK_SET );
```

where the constant of the `long int` type is indicated using the suffix "L". This function will therefore allow us to re-read a file from the beginning.

9.2.4. Input/output functions for standard devices

Standard terminal inputs and outputs can be performed using standard input/output functions and also using stream file handling functions. For the second of these, we need to use the standard device references for the files which are opened on the execution of a program. In C there are at least three pre-defined files: `stdin` for the standard input, `stdout` for the standard output and `stderr` for the output of error messages, which often matches `stdout`.

9.3. Example

Below is shown a small program that counts the number of words and lines in a text file. The program understands a word as being a string of characters between two blank spaces. A blank space is any character which causes `isspace()` to return a true value. Ends of lines are marked using the return character (ASCII number 13), meaning a `'\n'`. It is important to observe the use of functions relating to byte stream files.

As we will see, the structure of programs that work with these files include the coding of some of the algorithmic schemes for the processing of data sequences (stream files are in fact sequences of bytes). As we are counting the words and the lines, we will need to run through the whole input sequence. We can therefore observe that the code of the program perfectly follows the algorithmic scheme for running through sequences.

```
/* File: nwords.c                                              */
#include <stdio.h>
#include <ctype.h> /* Contains: isspace()                      */

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
main()
{
   char file_name[FILENAME_MAX];
   FILE* stream;
   bool in_word;
   char c;
   unsigned long int nwords, nlines;

   printf( "Word and line counter.\n" );
   printf( "Name of file: ");
   gets( file_name );
   stream = fopen( file_name, "rt" );
   if( stream != NULL ) {
    nwords = 0;
    nlines = 0;
    in_word = FALSE;
    while( ! feof( stream ) ) {
     c = fgetc( stream );
     if( c == '\n' ) nlines = nlines + 1;
     if( isspace( c ) ) {
      if( in_word ) {
       in_word = FALSE;
       nwords = nwords + 1;
      } /* if */
     } else { /* if the character is not a blank space         */
      in_word = TRUE;
     } /* if */
    } /* while */
    printf( "Number of words = %lu\n", nwords );
    printf( "Number of lines = %lu\n", nlines );
   } else {
    printf( "The file can not be opened!\n" );
   } /* if */
} /* main */
```

Note

Words are detected by checking the end of the word, this must consist of a character which is not a blank space followed by one which is a blank space.

10. Principles of modular programming

Reading the source code of a program implies tracking the flow of the execution of its instructions (flow control). Obviously the execution of instructions in their sequential order does not require much attention. But, as we have seen, programs can also contain conditional, alternative and iterative instructions. With all these, flow control tracking can get complicated if the source code occupies more space than can be viewed (more than 20 lines for example).

It is therefore convenient to group those parts of the code which perform a specific function into an individually-identified sub-program. This is even more useful when dealing with functions which are performed several times during the execution of a program.

In the following sections we will see how a C program can be divided into several sub-programs. This layout is similar for other programming languages.

11. Functions

In C, the groups of code into which a program is divided are called *functions*. What is more, in C all code must be divided into functions and in fact the whole program is merely a function: the main function (`main`).

Generally speaking, the code of a function will include at least the programming of a few algorithmic schemes for processing data sequences and some conditional or alternative instructions. At least those required to perform a specific task.

11.1. Declaration and definition

The declaration of any object (function or variable) implies demonstrating its existence to the compiler, while a definition implies describing its content. We have seen the difference between these two in terms of variables but have only mentioned it in terms of functions.

Declaration is exactly the same as for variables: the demonstration of existence. In this case, however, we will need to describe the arguments it takes and the result returned so that the compiler can generate the code and use them.

Header files contain the function declarations.

Conversely, defining a function deals with the program, which is its content. So, in a similar way to variables, the content can be defined by the position of the first of its bytes in the main memory. The first byte is the first of the first instruction executed to perform the program's task.

11.1.2. Declarations

The declaration of a function consists of specifying the type of data returned, the name of the function, the list of parameters it receives in parenthesis and a semi-colon to end the declaration.

```
data_type function_name( parameter_list );
```

We should remember that we can not reference a function which has not been declared beforehand. For this reason we need to include the header files for the standard functions of the C library as `stdio.h`, for example.

If a function has not been declared beforehand, the compiler will assume that it returns an integer. In the same way, if we omit the data type it returns, it will assume that it is an integer.

The parameter list is optional and consists of a list of declarations of variables which will contain data taken as arguments of the function. Each declaration is separated from the next using a comma. For example:

```
float average_score( float theo, float prb, float pract );
bool pass( float mark, float tolerance );
```

If the function returns no value or no argument is needed, this should be indicated using the empty data type (`void`):

```
void warning( char message[] );
bool yes_or_no( void );
int read_code( void );
```

11.1.3. Definitions

The definition of a function should always be preceded by its declaration, which should now include the list of parameters, if there are any. This header does not end with a semi-colon but will be followed by the body of the function written between opening and closing brackets:

```
data_type function_name( parameter_list )
{ /* body of the function: */
  /* 1) declaration of local variables */
  /* 2) instructions of the function      */
} /* function_name */
```

As we mentioned before, defining the function assumes that it has already been declared. Therefore, functions which perform tasks from other programs and, in particular, the main program (the `main` function) must be defined beforehand.

11.1.4. Calls

The mechanism for using a function in the code is the same as for the standard functions of the C library: We only need to refer to them by their name, provide the arguments required for it to carry out its task and, optionally, use the returned data in an expression which will usually be condition or an assignment.

The procedure by which the flow of the execution of instructions goes to the first instruction of a function is called a *call procedure*. We therefore talk about calling functions each time we wish to indicate that a function is to be used in a program.

We will now look at the sequence of a call procedure:

1) Prepare the execution environment for the function, this means reserving the space for the returned value, the formal parameters (the variables which are identified with each of the arguments present), and the local variables.

2) Perform the parameter step, this means copying the resulting values from the evaluation of each of the arguments of the call instruction to the formal parameters.

3) Execute the corresponding program.

4) Free the space occupied by the local environment and return the possible return value before going back to the instruction execution flow where the call was found.

This last step is done using the return instruction which will obviously be the last instruction executed in the function:

```
return expression;
```

Note

This instruction should appear empty or not appear at all if the function is a `void` type, meaning if it has been explicitly declared to not return any data.

The body of a function can also include a call to itself. This is known as a *recursive call* as the function is defined in terms of itself. These types of call are not incorrect but we need to make sure they do not repeat indefinitely, meaning that there must be a case where the flow of the execution of instructions does not imply performing another recursive call and also that the transformations applied to the parameters leads, at some point, to the prior execution conditions. In particular we cannot do the following:

```
/* ... */
void menu( void )
{
  /* show options menu,      */
  /* execute selected option   */
  menu();
/* ... */
```

The above function will result in an indefinite number of calls to `menu()` and therefore the continual creation of local environments without freeing them afterwards. In this situation, it is possible that the program will not execute correctly after a while due to a lack of memory for the creation of new environments.

11.12. Scope of variables

The scope of a variable refers to the parts of a program which can use them. In other words, the scope of a variable covers all those instructions which can access them.

The code of a function can use all global variables (those which are "visible" to any instruction within the program), all formal parameters (variables which are the arguments of the function) and all local variables (those which are declared within the body of a function).

In some cases it may not be appropriate to use global variables as they make it harder to compress the source code, this will make debugging and maintenance of the program harder later on. To illustrate this let's look at the following example:

```
#include <stdio.h>

unsigned int A, B;

void reduce( void )
{
  if( A < B ) B = B - A;
  else A = A - B;
} /* reduce */

void main( void )
{
  printf( "The MCD of: " );
  scanf( "%u%u", &A, &B );
 while( A!=0 && B!=0 ) reduce();
  printf( "... is %u\n", A + B );
} /* main */
```

Although this program works correctly we can not directly deduce what the `reduce()`function does, nor can we determine the variables it uses nor those it affects. We therefore need to adopt the rule that no function may depend on or affect global variables. Due to the fact that in C all code is divided into functions, we can easily see that there should be no global variables.

All variables are therefore local in scope (formal parameters and local variables). In other words, they are declared in the local environment of a function and can only be used by instructions within the same.

Local variables are created at the moment the corresponding function is activated, meaning after the execution of the instruction which calls this function. For this reason their storage class is known as automatic, as they are created and destroyed automatically during the function call procedure. This storage class can be made explicit using the key word: `auto`:

```
int any_function( int a, int b )
{
  /* ... */
  auto int local_variable;
  /* rest of the function */
} /* any_function */
```

Sometimes it is useful that the local variable is stored temporarily in one of the processor registries to avoid having to update it continually in the main memory, this speeds up the execution of the instructions (normally these will be iterative). In these cases we can ask the compiler to create the machine code in this way; i.e. so that the local variable is stored in one of the processor registries. However many compilers are able to perform this kind of optimisation autonomously.

This storage class can be indicated using the keyword `register`:

```
int any_function( int a, int b )
{
  /* ... */
  register int counter;
  /* rest of the function */
} /* any_function */
```

To achieve the opposite effect we can indicate that a local variable should always reside in memory using the indication `volatile` as the storage class. This is only appropriate when the variable can be modified from outside the program.

```
int any_function( int a, int b )
{
  /* ... */
  volatile float temperature;
  /* rest of the function */
} /* any_function */
```

In the above cases these are automatic variables. However, we will sometimes want a function to keep the information contained in a variable between separate calls. This allows the corresponding algorithm to "remember" an aspect of a past condition. To achieve this we need to indicate that the variable has a `static` storage class, meaning that it is static or immovable in the memory.

```
int any_function( int a, int b )
{
  /* ... */
  static unsigned number_calls = 0;
  number_calls = number_calls + 1;
  /* rest of the function */
} /* any_function */
```

In the above case it is important to initialise the variables in the declaration, if we do not, we will not know the initial content before the function is called.

As a final note, we should mention that storage classes are rarely used in C programming. In fact, with the exception of `static`, they have virtually no effect in current compilers.

11.3. Parameters by value and by reference

Passing parameters refers to the action of transforming formal parameters to real parameters, this means assigning a content to the variables representing the arguments.

```
type function_call(
  formal_parameter_1,
  formal_parameter_2,
  ...
);
calling_function( ... )
{
  /* ... */
  calling_function( real_parameter_1, real_parameter_2, ... )
  /* ... */
} /* calling_function */
```

There are two possibilities for this: that the arguments receive the result of the evaluation of the corresponding expression or they are replaced by the variable indicated in the real parameter at the same position. The first case is known as **passing parameters by value**, while the second is known as **passing a variable** (any change to the argument is a change to the variable which constitutes the real parameter).

Passing by value consists of assigning the resulting value of the real parameter at the same position to the variable of the formal parameter. Passing a variable consists of replacing the variable of the real parameter with that of the corresponding formal parameter and, consequently, being able to use it within the same function with the name of the formal parameter.

In C, passing parameters is only done by value, this means that all of the parameters in the call are calculated and the results assigned to the corresponding formal parameter in the function.

In order to modify a variable which we wish to pass as an argument when calling a function, we need to pass the memory address at which it is to be found. This means we need to use the get address operator (&) the result of which is the memory address at which the argument resides (variable, tuple field, or matrix element among others). This is the mechanism which makes the `scanf` function deposit the values read into the variables passed to it as arguments.

In call functions, the formal parameter which receives a reference to a variable instead of a value must be declared in a special way, an asterisk must be placed before its name. The asterisk in this context can be read as the "content whose initial position is found in the corresponding variable". Therefore, in a function such as the one shown below, the message would be: "the content whose initial position is found in the formal parameter numerator" is of an integer type. It would be read the same for the denominator:

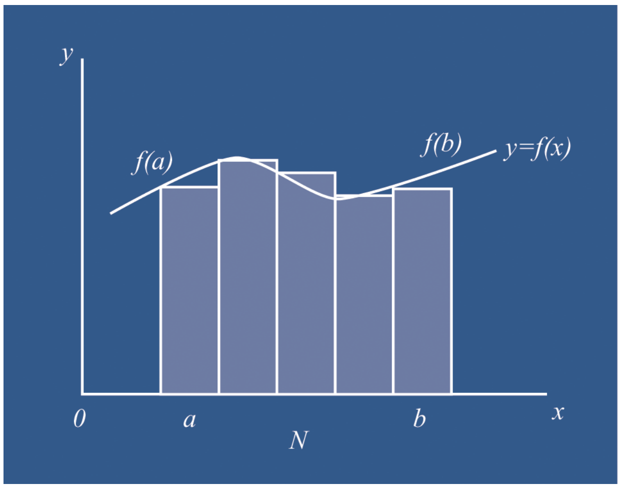
```
void simplifies( int *numerator, int *denominator )
{
  int mcd;
  mcd 0 maximum_common_divisor( *numerator, *denominator );
  *numerator = *numerator / mcd;
  *denominator = *denominator / mcd;
} /* simplifies */
/* ... */
  simplifies( &a, &b );
/* ... */
```

Although we look at this more closely later on, we need to remember that the asterisk in the code part must be read as "the content of the variable which is stored in the memory position of the corresponding argument". We must therefore use `*formal_parameter` each time we wish to use the variable passed by reference.

11.4. Example

The following program numerically calculates the integral of a function in a certain interval using the Simpson rule. Basically this method consists of dividing the integration interval into a certain number of segments of the same length which will form the base of a rectangle, the height of which will be determined by the value of the function to integrate at the initial point of the segment. The sum of the areas of these rectangles will give the approximate area defined by the function, the axis of the X's and the perpendicular lines to that which passes through the initial and final points of the integration segment.

Figure 3.



```
/* File: simpson.c                                          */

#include <stdio.h>
#include <math.h>

double f( double x )
{
  return 1.0/(1.0 + x*x);
} /* f */

double integral_f( double a, double b, int n )
{
  double result;
  double x, dx;
  int i;
  result = 0.0;
  if( (a < b) && (n > 0) ) {
   x = a;
   dx = (b-a)/n;
   for( i = 0; i < n; i = i + 1 ) {
    result = result + f(x);
     x = x + dx;
    } /* for */
   } /* if */
  return result;
} /* integral_f */

void main( void )
{

  double a, b;
  int    n;

  printf( "Numerical integration of f(x).\n" );
  printf( "Initial point of the interval, a = ? " );
  scanf( "%lf", &a );
  printf( "Final point of the interval, b = ? " );
  scanf( "%lf", &b );
  printf( "Number of divisions, n = ? " );
  scanf( "%d", &n );
  printf(
   "Result, integral(f)[%g,%g] = %g\n",
   a, b, integral_f( a, b, n )
  ); /* printf */
} /* main */
```

12. C preprocessor macros

The preprocessor does not just replace simple symbols like the ones we have seen. It can also perform actions with parameters. Definitions of symbol replacements using parameters are called "macros".

```
#define symbol constant_expression
#define macro( arguments ) const_expression_with_arguments
```

The use of macros can help to clarify small sections of code by using similar syntax to function calls.

In this way certain simple operations can benefit from having a meaningful name instead of using constructions in C which can make it harder to understand their meanings.

Example

```
#define absolute( x ) ( x < 0 ? -x : x )
#define rounding( x ) ( (int) ( x + 0.5) )
#define truncate( x ) ( (int) x )
```

We must bear in mind that the name of the macro and the left-hand bracket can not be separated and that the continuation of the line (if the command is too long) must be done using a backslash before the line feed character.

We should also be aware that macros replace each parameter name appearing in the definition by the section of source code indicated as the argument. Therefore:

```
absolute( 2*integer + 1 )
```

would be replaced by:

```
( 2*integer + 1 < 0 ? -2*integer + 1 : 2*integer + 1 )
```

it would therefore not be correct if it was negative.

Note

In this case it is possible to prevent the error if we place brackets around the argument in the definition.

Summary

The organisation of the source code is essential for writing readable programs which are easy to maintain and update. This is especially true for open code program, i.e. free software.

In this unit we have reviewed the fundamental aspects involved in organising code. In essence, the correct organisation of the source code of a program depends as much on the instructions as in the data. We have therefore looked at how to use data structures as well as how to organise the program.

The correct organisation of the program starts with a clear execution flow of the instructions. Given that the simplest instruction flow is one in which instructions are executed sequentially as they appear in the code, it is essential that flow control instructions have a single input point and a single output point. Structured programming is based on this method. In this method there are only two types of flow control instructions: alternative and iterative.

Iterative instructions provide another challenge in determining the control flow as we need to determine that the condition which stops the iteration is satisfied at least once. For this reason we have reviewed the algorithmic schemes used for the treatment of data sequences and we have looked at small programs which, as well as serving as examples of structured programming, are useful for performing data filtering operations using pipes (daisy-chained processes).

To correctly organise the code and to be able to process complex information we need to use data structuring. In this aspect we need to remember that the program must reflect those operations performed on the information and not so much the data elements they contain. We have therefore looked at how to declare and use structured data, whether they are homogenous or heterogeneous and we have described how to define new data types from basic and structured data types. These new types are known as *abstract data types* as they are transparent to the programming language.

When talking of data we have also looked at byte stream files. These homogenous data structures are characterised by having an undefined number of elements and residing in secondary memory, meaning on an external media device, and lastly, in that they require specific functions for accessing their data. We have therefore looked at the standard C functions for using these types of files. Basically, programs which use them implement algorithmic schemes which perform a *run-through* or search and which include a specific initialisation for opening the files, end of file checks for the iteration condition, read and write operations for the processing of data sequences and, lastly, a finalisation which includes closing the files used, among other things.

In the last section we looked at modular programming which involves grouping instruction sequences into sub-programs which perform a specific function and can be used more than once in the same program or in others. The sub-program is therefore replaced in the execution flow of the program by an instruction which will execute the corresponding sub-program. These sub-programs are known as "functions" in C and the instruction which executes it is known as a "call instruction". We have looked at how to call a function and how parameter passing is the most important aspect of this.

Parameter passing involves the transfer of a data set to a function which uses it to perform its task. Given that the function may need to return results which cannot be stored in a simple variable, some of these parameters are used to pass references to variables which could also contain returned values. We therefore also analysed the problems associated with passing parameters by value and by reference.

Self-evaluation

1. Write a program to determine the number of digits needed to represent a given whole number. The algorithm should divide the number by 10 until the result is a number lower than 10.

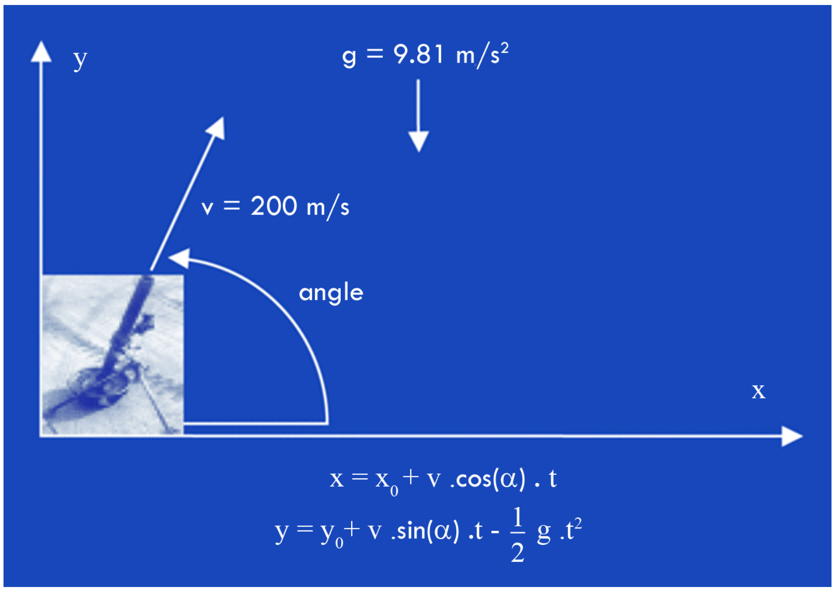
2. Write a program to determine the position of a projectile launched from a mortar at any time. It should show the height and distance at regular time intervals until it hits the ground. We will assume that the ground is flat and the input data is made up of the angle of the barrel and the time interval you wish to use for displaying the data. We will assume that the exit speed of the projectile is 200 m/s.

Note

We will assume that the mortar barrel is 1 m long and that the angle can vary between 45 and 85 degrees.

The following scheme summarises the various formulas which are needed to resolve the problem:

Figure 4.


y
$g = 9.81\ m/s^2$
$v = 200\ m/s$
angle
x
$x = x_0 + v\ .\cos(\alpha)\ .\ t$
$y = y_0 + v\ .\sin(\alpha)\ .t - \frac{1}{2}\ g\ .t^2$

where x_0 and y_0 are the initial position (can be considered as 0 for both), and α is the angle in radians (Π radians = 180°).

3. We wish to calculate the final amount accumulated by a pension plan using the initial amount, the age of the insured person (we will assume they retire at 65) and the payments and interest rate percentages returned each year. (we will assume that payments are made on an annual basis.)

4. Program a filter which will calculate the average, the maximum and the minimum of a series of real input numbers.

5. Use the filters from the last example in section "3.2 Filters and pipes"; meaning: calculate the amounts of a data sequence { item code, price, amount } which generate another data sequence { item code, amount } and then add together the amounts of the second data sequence.

6. Write a program which calculates the standard deviation for the occupancy figures of a public car park over a period of 24 hours. There will therefore be 24 pieces of input data.

This data should refer to the occupancy percentage (the number of occupied places in relation to the total number of places) calculated at the end of each hour. It should also indicate the times of the day having an occupancy percentage lower than the average minus twice the standard deviation and those above the average plus twice the standard deviation.

Note

The standard deviation is calculated as the square root of the sum of the squares of the differences between the data and the average, divided by the number of samples.

7. Check whether the letter of a given fiscal identity number is correct or not. The procedure for calculating it will consist of performing the modulus 23 of the corresponding number. The result of a position in a sequence of letters (TRWAGMYFPDXBNJZSQVHLCKE). The letter at this position will be the NIF letter.

Note

To be able to compare letters we will need to change the input letter into upper case. To do this we will use `toupper()`, whose declaration is in `ctype.h` and it returns the character of the letter received in upper case as the argument. If it is not an alphabetical character, or is already a capital letter, it will return the same character.

8. Write a program which calculates the minimum number of coins needed to give change if we know the total amount to pay and the amount received in payment. The maximum coin value is 2 Euros and the smallest is 1 cent.

Note

We should create a vector with the values of coins ordered by value.

9. Summarise the activity at a sales terminal by item. The program should show the number of units sold for each item code. To do this you will need a file generated by the terminal made up of pairs of integers: the first of these will be the code and the second, the number sold. If items were returned, the amount will be displayed as a negative value. We also know that there will never be more than 100 different item codes.

Note

We should use a vector of 100 tuples to store the corresponding information on the code and the amount. As we do not know how many tuples will be needed, bear in mind that we will need a variable which indicates the number stored in the vector (from 0 to the number of different codes -1).

10. Re-program the above exercise so that the operations which affect the vector are carried out in the specific function body.

Note

Define a new data type which contains the information on the products. We suggest the one shown below for example.

```
typedef struct products_s {
 unsigned int n; /* Number of products. */
 sale_t product[MAX_PRODUCTS];
} products_t;
```

Remember that we will need to pass this type of variable by reference.

11. Search for a word in a text file. Write a program which asks for both the text of the word and the name of the file. The result should be a list of all the lines in which the word is found.

We will assume that a word is a sequence of alphanumeric characters. We will need to use the `isalnum()`macro which is declared in `ctype.h`, to determine whether a character is alphanumeric or not.

Note

We will use the functions declared below in the proposed solution.

```
#define WORD_LENGTH 81
typedef char word_t[WORD_LENGTH;
bool same_words( word_t p1, word_t p2 );
unsigned int read_word( word_t p, FILE *input );
void first_word( word_t word, char *phrase );
```

Answer key

1.

```
/* ---------------------------------------------------- */
/* File: ndigits.c */
/* ---------------------------------------------------- */

#include <stdio.h>

main()
{
 unsigned int number;
 unsigned int digits;
  printf( "The number of digits to display: " );
  scanf( "%u", &number );
  digits = 1;
  while( number > 10 ) {
   number = number / 10;
   digits = digits + 1;
  } /* while */
  printf( "... is %u.\n", digits );
} /* main */
```

2.

```
/* ---------------------------------------------------- */
/* File: morter.c */
/* ---------------------------------------------------- */

#include <stdio.h>
#include <math.h>

#define INITIAL_V 200.00          /* m/s                */
#define L_TUBE     1.0            /* m                  */
#define G          9.81           /* m/(s*s)            */
#define PI 3.14159265

main()
{
  double angle, inc_time, t;
  double v_x, v_y; /* Horizontal and vertical speeds. */
  double x0, x, y0, y;

  printf( "Mortar shot.\n " );
  printf( "Angle of shot [sexagesimal degrees] =? " );
  scanf( "%lf", &angle );
  angle = angle * PI / 180.0;
  printf( "Sample cycle [seconds] =? " );
  scanf( "%lf", &inc_time );
  x0 = L_TUBE * cos( angle );
  y0 = L_TUBE * sin( angle );
  t = 0.0;
  v_x = INITIAL_V * cos( angle );
  v_y = INITIAL_V * sin( angle );
  do {
   x = x0 + v_x * t;
   y = y0 + v_y * t - 0.5 * G * t * t;
   printf( "%6.2lf s: ( %6.2lf, %6.2lf )\n", t, x, y );
   t = t + inc_time;
  } while ( y > 0.0 );
} /* main */
```

3.

```
/* ---------------------------------------------------- */
/* File: pensions.c */
/* ---------------------------------------------------- */
#include <stdio.h>

#define RETIREMENT_AGE 65

main()
{
   unsigned int age;
   float        capital, interest, payments;

   printf( "Pension plan.\n " );
   printf( "Age =? " );
   scanf( "%u", &age );
   printf( "Initial payment =? " );
   scanf( "%f", &capital );
   while( age < RETIREMENT_AGE ) {
     printf( "Interest paid [en %%] =? " );
     scanf( "%f", &interest );
     capital = capital*( 1.0 + interest/100.0 );
     printf( "New payment =? " );
    scanf( "%f", &payment );
     capital = capital + payment;
     age = age + 1;
     printf( "Capital accumulated after %u years: %.2f\n",
     age, capital
     ); /* printf */
    } /* while */
 } /* main */
```

4.

```
/* ---------------------------------------------------- */
/* File: statistics.c */
/* ---------------------------------------------------- */

#include <stdio.h>
main()
{
  double       sum, minimum, maximum;
  double       number;
  unsigned int amount, read_ok;

  printf( "Minimum, average and maximum.\n " );
  read_ok = scanf( "%lf", &number );
  if( read_ok == 1 ) {
   number = 1;
   sum = number;
   minimum = number;
   maximum = number;
   read_ok = scanf( "%lf", &number );
   while( read_ok == 1 ) {
    sum = sum + number;
    if( number > maximum ) maximum = number;
    if( number < minimum ) minimum = number;
    amount = amount + 1;
    read_ok = scanf( "%lf", &number );
   } /* while */
   printf( "Minimum = %g\n", minimum );
   printf( "Average = %g\n", sum / (double) amount );
   printf( "Maximum = %g\n", maximum );
  } else {
   printf( "Entry empty.\n" );
  } /* if */
 } /* main */
```

5.

```
/* ---------------------------------------------------- */
/* File: calc_amounts.c */
/* ---------------------------------------------------- */
#include <stdio.h>
main()
{
  unsigned int read_ok, code;
  int quantity;
  float        price, amount;

  read_ok = scanf( "%u%f%i",
   &code, &price, &quantity
  ); /* scanf */
  while( read_ok == 3 ) {
   amount = (float) quantity * price;
   printf( "%u %.2f\n", code, amount );
   read_ok = scanf( "%u%f%i",
    &code, &price, &quantity
   ); /* scanf */
  } /* while */
} /* main */

/* ---------------------------------------------------- */
/* File: totalise.c */
/* ---------------------------------------------------- */
#include <stdio.h>
main()
{
  unsigned int read_ok, code;
  int quantity;
  float        amount;
  double       total = 0.0;

  read_ok = scanf( "%u%f", &code, &amount );
  while( read_ok == 2 ) {
   total = total + amount;
   read_ok = scanf( "%u%f", &code, &amount );
  } /* while */
  printf( "%.2lf\n", total );
} /* main */
```

6.

```
/* ---------------------------------------------------- */
/* File: occupy_pk.c */
/* ---------------------------------------------------- */
#include <stdio.h>
#include <math.h>
main()
{
  unsigned int hour;
printf( "Daily occupation statistics:\n" );
/* Read occupation ratios per hour:            */
for( hour = 0; hour < 24; hour = hour + 1 ) {
 printf(
  "Percentage occupation at %02u hours =? ",
  hour
 ); /* printf */
 scanf( "%f", &percentage );
 ratio_accum[hour] = percentage;
} /* for */
/* Calculation of average:                                 */
 average = 0.0;
for( hour = 0; hour < 24; hour = hour + 1 ) {
 average = average + ratio_accum[hour];
} /* for */
average = average / 24.0;
/* Calculation of standard deviation:                    */
dev = 0.0;
for( hour = 0; hour < 24; hour = hour + 1 ) {
 diff = ratio_accum[ hour ] - average;
 dev = dev + diff * diff;
} /* for */
dev = sqrt( dev ) / 24.0;
/* Print results:                          */
printf( "Average occupation for the day: %.2lf\n", average );
printf( "Standard deviation: %.2lf\n", dev );
printf( "Hours with low percentage: ", dev );
for( hour = 0; hour < 24; hour = hour + 1 ) {
 diff = average - 2*dev;
 if( ratio_accum[ hour ] < diff ) printf( "%u ", hour );
} /* for */
```

```
  printf( "\nHours with high percentage: ", dev );
  for( hour = 0; hour < 24; hour = hour + 1 ) {
  diff = average +2*dev;
  if( ratio_accum[ hour ] > diff ) printf( "%u ", hour );
 } /* for */
 printf( "\nEnd.\n" );
 } /* main */
```

7.

```
/* ---------------------------------------------------- */
/* File: valid_nif.c */
/* ---------------------------------------------------- */

#include <stdio.h>
#include <ctype.h>
main()
{
  char         NIF[ BUFSIZ ];
  unsigned int national identity number;
  char         letter;
  char code[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;

  printf( "Enter the fiscal identity number (letter of your identity document): " );
  gets( NIF );
  sscanf( fiscal identity number, "%u", &identity document);
  identity document = identity document % 23;
  letter = identity document[ strlen(identity document)-1 ];
  letter = toupper( letter );
  if( letter == code[ identity document ] ) {
   printf( "fiscal identity number valid.\n" );
  } else {
   printf( "Letter %c not valid!\n", letter );
  } /* if */
} /* main */
```

8.

```
/* ---------------------------------------------------- */
/* File: change.c */
/* ---------------------------------------------------- */

#include <stdio.h>

#define NUM_COINS 8

main()
{
  double       amount, paid;
  int          change_cents;
  int          i, ncoins;
  int          cents[NUM_COINS] = { 1, 2, 5, 10,
               20, 50, 100, 200 };
  printf( "Amount : " );
  scanf( "%lf", &amount );
  printf( "Paid : " );
  scanf( "%lf", &paid );
  change_cents = (int) 100.00 * ( paid - amount );
  if( change_cents < 0 ) {
   printf( "UNDERPAYMENT\n");
  } else {
   printf( "To be returned : %.2f\n",
    (float) change_cents / 100.0
   ); /* printf */
   i = NUM_COINS - 1;
   while( change_cents > 0 ) {
    ncoins = change_cents / cents[i] ;
    if( ncoins > 0 ) {
     change_cents = change_cents - cents[i]*ncoins;
     printf( "%u coins of %.2f Euros.\n",
      ncoins,
      (float) cents[i] / 100.0
     ); /* printf */
    } /* if */
    i = i - 1;
   } /* while */
  } /* if */
} /* main */
```

9.

```
/* ------------------------------------------------------- */
/* File: summary_tpv.c */
/* ------------------------------------------------------- */
#include <stdio.h>

typedef struct sale_s {
  unsigned int code;
  int quantity;
} sale_t;
#define MAX_PRODUCTS 100

main()
{
 FILE         *input;
 char         name[BUFSIZ];
 unsigned int read_ok;
 int          num_prod, i;
 sale_t      vale, product[MAX_PRODUCTS];

 printf( "Summary for the day in TPV.\n" );
 printf( "File: ");
 gets( name );
 input = fopen( name, "rt" );
 if( input != NULL ) {
  num_prod = 0;
  read_ok = fscanf( input, "%u%i",
   &(sale.code), &(sale.quantity)
  ); /* fscanf */
  while( read_ok == 2 ) {
   /* Search the code in the table:               */
   i = 0;
   while( ( i < num_prod ) &&
       ( product[i].code != sale.code )
   ) {
    i = i +1;
   } /* while */
   if( i < num_prod ) { /* Code found:        */
   product[i].quantity = product[i].quantity +
                                sale.quantity;
       } else { /* Code not found => new product: */
        product[num_prod].code = sale.code;
        product[num_prod].quantity = sale.quantity;
        num_prod = num_prod + 1;
       } /* if */
       /* Read next sale: */
       read_ok = fscanf( input, "%u%i",
        &(sale.code), &(sale.quantity)
       ); /* fscanf */
      } /* while */
      printf( "Summary:\n" );
      for( i=0; i<num_prod; i = i + 1 ) {
       printf( "%u : %i\n",
        product[i].code, product[i].quantity
       ); /* printf */
      } /* for */
     } else {
      printf( "Can not open %s!\n", name );
     } /* if */
  } /* main */
```

10.

```
/* ... */

int search( unsigned int code, products_t *pref )
{
 int i;

  i = 0;
  while( (i < (*pref).n ) &&
     ( (*pref).product[i].code != code )
  ) {
     i = i +1;
  } /* while */
  if( i == (*pref).n ) i = -1;
  return i;
} /* search */

void add( sale_t sale, products_t *pref )
{
  unsigned int n;

  n = (*pref).n;
  if( n < MAX_PRODUCTS ) {
   (*pref).product[n].code = sale.code;
   (*pref).product[n].quantity = sale.quantity;
   (*pref).n = n + 1;
  } /* if */
} /* add */
void update(   sale_t   sale,   unsigned   int   position,
products_t *pref )
{
  (*pref).product[position].quantity =
   (*pref).product[position].quantity + sale.quantity;
} /* update */

void sample( products_t *pref )
{
  int i;

  for( i=0; i<(*pref).n; i = i + 1 ) {
   printf( "%u : %i\n",
      (*pref).product[i].code,
      (*pref).product[i].quantity
    ); /* printf */
   } /* for */
} /* sample */

void main( void )
{
  FILE         *input;
  char         name[BUFSIZ];
  unsigned int read_ok;
  int i;
  sale_t      sale;
  products_t  products;

  printf( "Summary for the day in TPV.\n" );
  printf( "File: ");
  gets( name );
  input = fopen( name, "rt" );
  if( input != NULL ) {
   products.n = 0;
   read_ok = fscanf( input, "%u%i",
    &(sale.code), &(sale.quantity)
   ); /* scanf */
   while( read_ok == 2 ) {
    i = search( sale.code, &products );
    if( i < 0 ) add( sale, &products );
    else update( sale, i, &products );
    read_ok = fscanf( input, "%u%i",
     &(sale.code), &(sale.quantity)
     ); /* scanf */
    } /* while */
    printf( "Summary:\n" );
    sample( &products );
   } else {
    printf( "Can not open %s!\n", name );
   } /* if */
} /* main */
```

11.

```
/* ---------------------------------------------------- */
/* File: search.c */
/* ---------------------------------------------------- */
#include <stdio.h>
#include <ctype.h>
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
#define WORD_LENGTH 81
typedef char word_t[WORD_LENGTH;
bool same_words( word_t p1, word_t p2 )
{
  int i = 0;
  while( (p1[i]!='\0') && (p2[i]!='\0') && (p1[i]==p2[i])) {
   i = i +1;
  } /* while */
  return p1[i]==p2[i];
} /* same_words */
unsigned int read_word( word_t p, FILE *input )
{
  unsigned int i, nlin;
  bool         term;
  char         character;
  i = 0;
  nlin = 0;
  term = FALSE;
  character = fgetc( input );
  while( !feof( input ) && !term ) {
   if( character == '\n' ) nlin = nlin + 1;
   if( isalnum( character ) ) {
    p[i] = character;
    i = i +1;
    character = fgetc( input );
   } else {
    if( i > 0 ) {
     term = TRUE;
```

```
        } else {
          character = fgetc( input );
        } /* if */
      } /* if */
    } /* while */
    p[i] = '\0';
    return nlin;
  } /* read_word */

  void first_word(word_t word, char *phrase )
  {
    int i, j;
    i = 0;
    while( phrase[i]!='\0' && isspace( phrase[i] ) ) {
      i = i +1;
    } /* while */
    j = 0;
    while( phrase[i]!='\0' && !isspace( phrase[i] ) ) {
      word[j] = phrase[i];
      i = i +1;
      j = j + 1;
    } /* while */
    word[j] = '\0';
  } /* first_word */

  void main( void )
  {
    FILE          *input;
    char          name[BUFSIZ];
    word_t    word, word2;
    unsigned int numlin;

    printf( "Search words.\n" );
    printf( "File: ");
    gets( name );
    input = fopen( name, "rt" );
    if( input != NULL ) {
      printf( "Word: ");
      gets( name );
      first_word( word, name );
      printf( "Searching %s in file...\n", word );
      numlin = 1;
```

```
   while( !feof( input ) ) {
    numlin = numlin + read_word( word2, input );
    if( same_words( word, word2 ) ) {
     printf( "... line %lu\n", numlin );
    } /* if */
   } /* while */
  } else {
   printf( "Can not open %s!\n", name );
  } /* if */
} /* main */
```

Advanced programming in C. The development of efficient applications

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Josep Anton Pérez López
Lluís Ribas i Xirgo

PID_00148427

Index

Introduction

Programming a computer application usually produces a source code of considerable size. Even using modular programming techniques and standard library functions it can be a complex task to organise the code effectively. And much more if we bear in mind that the code will usually have been developed by a team of programmers.

We also need to remember that much of the information handled will not have a pre-defined size and it will often not be presented in the most suitable form to be processed. This means we will need to restructure the data so that the algorithms treating it are more effective.

Last but not least, we should remember that most applications are constructed from more than one program. This means that we need to organise the code using a set of software development tools and the operating system in which we are executing it.

In this unit we will look at several aspects of the above-mentioned problems. From the point of view of a program in the context of an application which contains it (even if it is the only one), it is important to adapt the actual size of the data to be processed, its layout in dynamic structures and the organisation of the code to the algorithm using it. Lastly, we will need to use the operating system for coordination with other programs both inside and outside the same application and also for interaction with the user.

Representing information in dynamic data structures allows us to adjust the memory requirements of the program to the minimum required for solving the problem and, also, to internally represent the information in such a way that it is processed more efficiently. A dynamic data structure is no more than a collection of data whose relationship has not been established *a priori* and which can be modified during the execution of the program. This can not be done using a vector for example as the data they contain is linked by the position within the vector, its size must also be pre-defined within the program.

In the first section we will look at dynamic variables and their use as data stores which respond to the requirements for adaptability to the information to be represented and the needs of the algorithm which has to handle it.

The source code of an application, whether a set of programs or a single one, must be organised in such a way that the characteristics of a good code are maintained (intelligible, easy to maintain and with an optimum execution cost). To do this, we will need to use both modular and structured programming and we will need to distribute it over several files to make it more manageable and also to maintain legibility.

The section on the top-down design of programs will deal with aspects which affect programming beyond modular programming. We will look at aspects relating to the division of the program into algorithmic terms and the grouping of sets of functions which are closely related. Given that the source code is distributed over several files we will also look at aspects to do with compilation and, especially, working with tools such as *make*.

Given the complexity of the source code of any applications it is advisable to all the standard functions provided by the operating system. This makes it more independent from the machine and also allows us to reduce the complexity by leaving out certain tasks such as simple call instructions. In the last section we will therefore look at the relationships between programs and operating systems and how to make programs communicate with it and with each other.

Note

We saw an example relating this issue in the last unit when we looked at pipes.

Lastly we will take a brief look at how to distribute the execution of a program over several execution flows (or threads). This will deal with how to write concurrent programs so that several tasks can be performed within the same time period.

In this unit we will try to demonstrate more involved aspects of programming using algorithms with a higher level of abstraction. At the end of this chapter we hope readers will have achieved the following goals:

1) The appropriate use of dynamic variables in a program.
2) Awareness of dynamic data structures and especially lists and their applications.
3) An understanding of the principle of top-down design in programming.
4) The ability to develop a library of functions.
5) To have basic knowledge on the relationship between the program and the operating system.
6) To assimilate the concept of the execution thread and the rudiments of concurrent programming.

1. Dynamic variables

The information processed by a program is made up of a data set which will often not have a fixed size, the maximum size is not known and the data are not related to each other in the same way etc.

Example

A program performing a syntactical analysis (which may also form part of a text processing application) must be able to process phrases of different sizes and with different numbers and categories of words. The words may also be related in very different ways. For example, adverbs are related to verbs and both of these are distinct from those which make up the noun phrase, among others.

The relationships existing between the elements forming a specific piece of information can be represented using additional data which represents these and which allows us to create a more efficient algorithm once they have been calculated. In the above example it will be far more effective to perform the required syntactic analysis using the syntactic tree and not from the simple succession of words in the phrase to be handled.

The size of the information, meaning the number of pieces of data it is made up of, will significantly affect the performance of a program. What is more, the use of static variables for storage means that we must either be aware of its maximum size *a priori* , or we must limit the treatment to only a portion of the information. Although it is possible to know the maximum size we will be wasting a lot of memory if the information occupies a much smaller size.

The **static variables** are those which are declared in such a way that they have a reserved memory space during the complete execution of the program. Only global variables are really static variables in C.

The **local variables**, on the other hand, are automatic because space is only reserved for them during the execution of part of a program, they are then destroyed automatically. Even so, they are still static variables with respect to the scope in which they are used as the space reserved for them is limited.

The **dynamic variables**, however, can be created and destroyed during the execution of a program and are global in nature, meaning they are "visible" from any point of the program. As it is possible to create an indefinite number of these variables, they can be adjusted to the size required to represent the information for a particular problem without wasting any memory space.

In order to create dynamic variables during the execution of the programs, we need to use the relevant operations. On the other hand, dynamic variables do not have a name and they are only identified by the address of the first memory location in which they reside.

To do this we will need to use data which can contain references to dynamic variables to allow them to be used. As these references are memory addresses, their type will be that of a memory address or *pointer*, as its value will indicate where the referenced variable is to be found.

In the following sections we will specifically look at everything pertaining to dynamic variables and the data structures which can be constructed using them.

2. Pointers

Pointers are variables which contain the memory addresses of other variables, also known as references to other variables. Obviously, the data type will be a memory location, making it an integer-compatible type. Even so, they have peculiarities which we will look at later.

A pointer is declared by declaring the data type of the variable which will contain addresses. We therefore use an indirection operator (an asterisk) or that which can be read as "content of the address", which is same. In the following examples we will declare several types of pointers:

```
int      *ref_integer;
char     *string;
other_t  *pointer;
node_t   *ap_node;
```

- In `ref_integer` , the content of the stored memory address is an integer data type.
- In `string` , the content of the stored memory address is a character.
- In `pointer` , the content of the stored memory address is of the type `other_t`.
- In `ap_node`, the content of the stored memory address is of the type `node_t`.

A reference to the variable without the indirection operator is simply the memory address it contains.

The type of pointer is determined by the data type held at the address. As such, we can say, for example, that `ref_integer` is an integer-type pointer or that it points "to" an integer type of data. It is also possible to declare pointers to pointers etc.

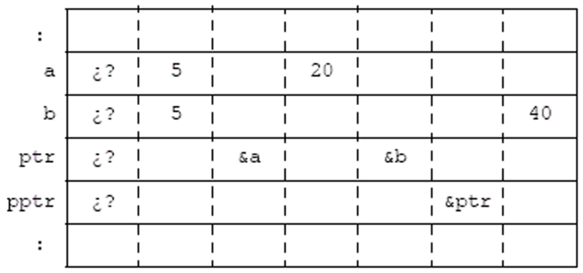
In the following example we can observe that, to reference the value pointed to by the address contained in a pointer, we need to use the "content of the address of":

```
int a, b; /* two integer-type variables.                       */
int *ptr; /* a pointer to an integer.                          */
int **pptr; /* a pointer to a pointer to an integer.           */
/* ... */
a = b = 5;
```

```
ptr = &a;
*ptr = 20; /* a == 20                                          */
ptr = &b;
pptr = &ptr;
**pptr = 40;/* b == 40                                         */
/* ... */
```

In the following figure we can appreciate how the program modifies the variables as they are executed. Each vertical column represents the modifications carried out by an instruction in the environment. To begin with (the far-left column), we do not know the content of the variables:

Figure 5.



In the case of tuples we need to remember that they can also be accessed using the indirection operator applied to the pointers which contain the initial addresses. Access to their fields does not change:

```
struct student_s {
   string_t name;
   unsigned short dni, mark;
} student;
struct student_s *ref_student;
/* ... */
student.score = *ref_student.score;
/* ... */
```

To be clear, when accessing a field of a variable whose address is in a variable, it is preferable to use the following:

```
/* ... */
student.mark = (*ref_student).mark;
/* ... */
```

If we want to emphasise the idea of the pointer, we can use the indirection operator for tuples which is similar to an arrow pointing to the corresponding tuple:

```
/* ... */
student.score = ref_student->score;
```

```
/* ... */
```

In the above examples, all the variables were static or automatic, but they are primarily used to reference dynamic variables.

2.1. The relationship between pointers and vectors

Vectors in C are hypothetical concepts of the programmer, as an operator (brackets) is used to calculate the initial address of an element within a vector. When doing this, we need to remember that the names used to declare them are in fact pointers to the first positions of the first elements of each vector. The following declarations are therefore practically the same:

```
/* ... */
int real_vector[DIMENSION];
int *virtual_vector;
/* ... */
```

In the first, the vector has a specific dimension while in the second, the `virtual_vector` is a pointer to an integer, meaning a variable which contains the address of integer-type data. Even so, it is possible to use the identifier of the first as a pointer to the second. In fact, it contains the address of the first integer of the vector:

```
real_vector == &(real_vector[0])
```

It is very important not to modify the content of the identifier as it may affect the reference to the whole vector!

Pointers are handled using a special type of arithmetic: adding and subtracting pointers of the same type and integers are allowed. Adding and subtracting integers will in fact be the addition and subtraction of multiples of integers as they are multiplied by the size, in bytes, of those they point to.

Adding or subtracting the contents of pointers is very uncommon. More often, they are decremented or incremented to point at a previous or subsequent element respectively. The following example illustrates this special arithmetic. This frees the programmer from having to think about how many bytes each data type occupies:

```
/* ... */
int vector[DIMENSION], *ref_integer;
/* ... */
ref_integer = vector;
ref_integer = ref_integer + 3;
```

```
*ref_integer = 15; /* This is equivalent to vector[3] = 15 */
/* ... */
```

We will always have the `sizeof` operator, this will return the size in bytes of the data type in the argument. The following example is therefore in fact an increment of `ref_integer` where `3*sizeof(int)` is added to the initial content:

```
/* ... */
ref_integer = ref_integer + 3;
/* ... */
```

Therefore, in the case of vectors it is true that:

```
vector[i] == *(vector+i)
```

Meaning that the element found at location i is that which is found at the memory location resulting from the addition of `i*sizeof(*vector)` to the initial address indicated in the vector.

Note

The `sizeof` operator applies to the element pointed to by `vector`, as by applying it to `vector` we would get the size, in bytes, which a pointer occupies.

In the following example we can see the relationship between pointers and vectors in more detail. The program shown below takes a complete name as an input and separates it into the name and surname.

```
#include <stdio.h>
#include <ctype.h>
typedef char phrase_t[256];
char *copy_word( char *phrase, char *word )
/* Copies the first word of the phrase to word.            */
/* phrase : points to a character vector.                  */
/* word : points to a character vector.                    */
/* Returns the address of the last character read          */
/* in the phrase.                                          */
{
  while( *phrase!='\0' && isspace( *phrase ) ) {
   phrase = phrase + 1;
  } /* while */
  while( *phrase!='\0' && !isspace( *phrase ) ) {
   *word = *phrase;
   word = word + 1;
   phrase = phrase + 1;
  } /* while */
  *word = '\0';
```

```
    return phrase;
  } /* word */

  main( void ) {
    phrase_t complete_name, name, surname1, surname2;
    char *position;

    printf( "Name and surnames? " );
    gets( complete_name );
    position = copy_word( complete_name, name );
    position = copy_word( position, surname1 );
    position = copy_word( position, surname2 );
    printf(

     "Thank you Mr/Ms %s.\n",
      surname1
    ); /* printf */
  } /* main */
```

Note

We will take another look at the relationships between pointers and vectors when dealing with character strings later on.

2.2. Function references

References to functions are in fact the address of the first executable instruction they contain. They can therefore be stored in function pointers.

The declaration of a pointer to a variable is done in a similar way to the declaration of pointers to variables: we just need to include an asterisk in its name. A pointer to a function which returns a real number, the result of performing some kind of operation on the argument, is declared in the following way:

```
  float (*ref_function)( double x );
```

Note

The brackets surrounding the name of the pointer and the asterisk preceding it is required so that it is not confused with the declaration of a function, the returned value of which is a pointer to a real number.

We will use a program for the numerical integration of a certain set of functions as an example. This program is similar to the one we looked at in the previous unit but where the numerical integration function has a new argument using the reference of the function for which we have to calculate the integral:

```
  /* Program: integrals.c */
```

```
#include <stdio.h>
#include <math.h>
double f0( double x ) { return x/2.0;          }
double f1( double x ) { return 1+2*log(x);     }
double f2( double x ) { return 1.0/(1.0 + x*x);}
double integral_f( double a, double b, int n,
                   double (*fref)( double x )
) {
  double result;
  double x, dx;
  int i;
  result = 0.0;
  if( (a < b) && (n > 0) ) {
   x = a;
   dx = (b-a)/n;
   for( i = 0; i < n; i = i + 1 ) {
    result = result + (*fref)(x);
    x = x + dx;
   } /* for */
  } /* if */
  return result;
} /* integral_f */

void main( void )
{
  double a, b;
  int    n, fnum;
  double (*fref)( double x );
printf( "Numerical integration of f(x).\n" );
printf( "Initial point of the interval, a = ? " );
scanf( "%lf", &a );
printf( "Final point of the interval, b = ? " );
  scanf( "%lf", &b );
  printf( "Number of divisions, n = ? " );
  scanf( "%d", &n );
  printf( "Function number, fnum = ?");
  scanf( "%d", &fnum );
  switch( fnum ) {
   case 1: fref = f1; break;
   case 2: fref = f2; break;
   default: fref = f0;
  } /* switch */
  printf(
   "Result, integral(f)[%g,%g] = %g\n",
   a, b, integral_f( a, b, n, fref )
  ); /* printf */
} /* main */
```

As we can observe, the main program is able to replace the function reference assignments by calls to the same. This would make the program far more clear. As we will see later on, this allows the function performing the numerical integration to be stored in a library and to be used in any program.

3. The creation and destruction of dynamic variables

As we mentioned at the beginning of this unit, dynamic variables are those which are created and destroyed during the execution of the program that uses them. Conversely, the others are static or automatic variables which do not need specific actions to be taken by the program in order to be used.

Before being able to use a dynamic variable we must reserve space for it using the standard function (declared in `stdlib.h`) which locates and reserves a space in the main memory of size `number_bytes` to allow it to contain the various data of a variable:

Note

The data type `size_t` is simply an unsigned integer-type which has been given this name because it shows sizes.

```
void * malloc( size_t number_bytes );
```

As the function does not know the data type of the future dynamic variable, it returns an empty pointer which needs to be coerced to the correct data type.

```
/* ... */
char *pointer;
/* ... */
pointer = (char *)malloc( 31 );
/* ... */
```

If it cannot reserve the space, it returns `NULL`.

It is often difficult to know exactly how many bytes each type of data occupies and its size may also depend on the compiler and the machine being used. For this reason we suggest to always use the `sizeof`. The above example should therefore have been written in the following way:

```
/* ... */
pointer = (char *)malloc( 31 * sizeof(char) );
/* ... */
```

`sizeof` returns the number of bytes needed to contain the data type of the variable or the data type it uses as the argument, except in the case of matrices in which it returns the same value as for a pointer.

It will sometimes be necessary to adjust the size of the space reserved for a dynamic variable (above all with vector types), either because there is no space for new data or because a large part of the memory is being wasted. To this end we can use the variable "reallocation" function:

```
void * realloc( void *pointer, size_t new_size );
```

The behaviour of the above function is similar to that of `malloc:` it returns `NULL` if it has not been able to find a new location for the variable of the required size.

When a dynamic variable is not needed any more it must be destroyed, meaning the space it occupies must be freed so that other dynamic variables can use it. To do this we will use the function `free:`

```
/* ... */
free( pointer );
/* ... */
```

As this function only frees up the space occupied, but does not affect the content of the pointer, this will still have the reference to the dynamic variable (its address) and the possibility of accessing a non-existent variable therefore exists. To avoid this, we suggest assigning the pointer to `NULL:`

```
/* ... */
free( pointer );
pointer = NULL;
/* ... */
```

Any incorrect reference to the erased dynamic variable therefore will produce an error which can easily be corrected.

4. Dynamic data types

Dynamic data types include those whose structure can be varied over the course of the execution of a program.

Changes to the structure may only apply to the number of elements as is the case with character strings, they may also apply to the relationships between them as could be the case with a syntax tree.

Dynamic data types can be stored in static data structures but, as they are groups of data, they must be vectors or, less commonly, multi-dimensional matrices.

Note

Static data structures are, by definition, the opposite of dynamic data structures. In these, neither the number of pieces of data nor their relationships change over the course of the execution of the program. For example, a vector will always have a certain length and all the elements, with the exception of the first and last, will have preceding and subsequent elements.

When storing dynamic data structures in static structures it is advisable to check whether we know the maximum number and the average number of pieces of data it may contain. If these two values are similar, we could use a static or automatic vector variable. If they are very different or we do not know them, it is better to adjust the size of the vector to the number of elements present in the data structure at a given time, as such we would store the vector in a dynamic variable.

Dynamic data structures are commonly stored using dynamic variables. We can therefore look at a dynamic data structure as a collection of dynamic variables whose relationship is established using pointers. In this way we can easily modify both the number of pieces of data in the structure (creating or destroying the variables which contain them) and the structure itself by changing the addresses contained in the pointers to the elements. In this case the elements are usually tuples which are called *nodes*.

In the following sections we will look at the two cases, dynamic data structures stored in static data structures and as collections of dynamic variables. The first case deals with character strings as these are the most commonly used dynamic data structures. The second deals with lists and their applications.

4.1. Character strings

Character strings are a special type of vector in which the elements are characters. An end marker is also used (the NULL character or `'\0'`) which bounds the actual length of the string represented in the vector.

The following declaration would cause many problems due to the non-inclusion of an end marker, as this rule must be respected in C when using all standard functions for string processes.

```
char string[20] = { 'H', 'e', 'l', 'l', 'o' } ;
```

This would therefore need to be declared in the following way:

```
char string[20] = { 'H', 'e', 'l', 'l', 'o', '\0' } ;
```

The declaration of character strings initialised using text implies that the end marker must always be added. The above declaration is therefore equivalent to:

```
char string[20] = "Hello";
```

Although the format for representing character strings is standard in C, there are no instructions or operators which work with strings: It is not possible to make assignations or comparisons with strings, we need to use the standard functions (declared in `string.h`) in order to handle strings.

```
int    strlen ( char *string );
char * strcpy ( char *destination, char *source );
char * strncpy ( char *destination, char *source, int char_num );
char * strcat ( char *destination, char *source );
char * strncat ( char *destination, char *source, int char_num );
char * strdup ( char *origin );
char * strcmp ( char *string1, char *string2 );
char * strncmp ( char *strn1, char *strn2, int char_num );
char * strchr ( char *string, char character );
char * strrchr ( char *string, char character );
```

The real length of a character string `strn` at a certain time can be obtained using the following function:

```
strlen ( strn )
```

The content of the character string pointed to by `strn` type to the `strn9`, can be copied using `strcpy( strn9, strn )`. If the source string is able to be longer than the capacity of the destination vector, use `strncpy( strn9, strn, LENGTH_STRN9 – 1 )`. In this last case we need to make sure the resulting string does not contain a `'\0'` at the end. To resolve this, we need to reserve the last character of the counted copy to put a `'\0'` by default. If there is no space reserved for the string it would need to be done as follows:

```
/* ... */
char *strn9, strn[MAX_LENGTH];
/* ... */
strn9 = (char *) malloc( strlen( strn ) + 1 );
if( strn9 != NULL ) strcpy( strn9, strn );
/* ... */
```

Note

In this way, `strn9` is a string with the space adjusted to the number of characters of the string stored in `strn`, this can be freed using `free( strn9 )` when it is no longer needed. The above procedure can be replaced by

```
/* ... */
strn9 = strdup( strn );
/* ... */
```

Strings are compared character by character starting with the first of the two strings and continuing with the following ones as long as the difference between the ASCII codes is 0. The function `strcmp()` returns the value of the last difference. This will mean a negative value if the second string is alphabetically greater than the first and positive in the opposite case, and 0 if they are the same. To understand this better we will look at a possible code for the function for comparing strings.

```
int strcmp( char *string1, char *string2 )
{
  while( (*string1 != '\0') &&
      (*string2 != '\0') &&
      (*string1 == *string2)
  ) {
   string1 = string1 + 1;
   string2 = string2 + 1;
  } /* while */
  return *string1 - *string2;
} /* strcmp */
```

The function `strncmp()` does the same as `strcmp()` with the first `char_num` characters.

Finally, although there is more, we will look at the functions for searching characters in strings. These functions return the pointer to the searched character or `NULL` if it is not found in the string:

- `strchr()` performs the search from the first character.

- `strrchr()` inspects the string starting from the right or the end.

Example

```
char *strchr( char *string, char character )
{
   while( (*string != '\0') && (*string2 != character) )
   {
    string = string + 1;
   } /* while */
   return string;
} /* strchr */
```

All of the above functions are declared in `string.h` and, therefore, to use them we need to include this file in the source code of the corresponding program.

In `stdio.h` there are also standard functions for operating with strings, such as `gets()` and `puts()`, which are used for inputting and outputting data which are character strings and which have already been described. It also contains the declarations for `sscanf()` and `sprintf()` for reading and writing formatted strings. These two last functions behave in exactly the same way as `scanf()` and `printf()` with the exception that reading and writing is done using a character string instead of using the standard input or output device.

```
sprintf(
  char *destination,            /* String to "print" to.                  */
  char *format
          [, list_of_variables]
); /* sprintf */
          int sscanf(           /* Returns the number of variables       */
                                /* whose content has been updated.       */
  char *origin,                 /* String being "read" from.             */
  char *format
         [, list_of_&variables ]
); /* sscanf */
```

When we use `sprintf()` we must check that the `destination` string has enough space to contain the result of printing with the given format.

When we use `sscanf()`, we always need to check that all the fields have been read: the inspection of the `origin` string stops when it finds the end marker, independently of the field specifiers indicated in the format.

In the following example we will look at the code of a string conversion function which represents a hexadecimal value (for example: "3D") to a positive integer (following the above example: $3D_{(16} = 61$) through the use of the functions mentioned above. The first is used to prefix the string with "0x", given that this is the format of hexadecimal numbers in C, and the second is used to read the string obtained taking advantage of the fact that it reads numbers in any standard C format.

```
unsigned hexaVal( char *hexadecimal )
{
  unsigned number;
  char *hexaC;

  hexaC = (char *) malloc(
   ( strlen( hexadecimal ) + 3 ) * sizeof( char )
  ); /* malloc */
  if( hexaC != NULL ) {
   sprintf( hexaC, "0x%s", hexadecimal );
   sscanf( hexaC, "%x", &number );
   free( hexaC );
  } else {
   number = 0; /* The conversion has not been done!*/
  } /* if */
  return number;
} /* hexaVal */
```

Note

Remember the importance of freeing up the space of the dynamically created character strings which are not going to be used. If you do not perform a `free( hexaC )`, the variable will continue to occupy space despite the fact that it can no longer be accessed, as the address is contained in the pointer `hexaC`, this is automatic and is therefore destroyed when execution of the function has ended. These types of errors can end up wasting a lot of memory.

4.2. Lists and queues

Lists are one of the most commonly-used dynamic data types and consist of homogenous sequences of elements with an unpredetermined size. As with character strings, they can be stored in vectors as long as we know the maximum length and the average length during execution. If we do not, we use dynamic variables which are "linked", meaning variables which contain pointers to others within the same dynamic data structure.

The advantage of representing a list using a vector is that it removes the need for a pointer field to the next one. We need to underline the fact that it is only possible to use it if it does not cause an excessive wastage of memory and when the program does not perform frequent insertions and removals of elements in any position of the list.

In this section we will look at a way of programming basic operations using list type dynamic data structures through dynamic variables. In this case each element of the list will be a node of the following type:

```
typedef struct node_s {
  int            data;
  struct node_s *next;
} node_t, *list_t;
```

The `node_t` type corresponds to a node of the list and `list_t` is a pointer to a node whose tuple has a `next` field, which is the pointer to another node, and so on until the whole list of nodes is linked. These types of lists are called ***simple linked lists*** as there is only one link between one node and the next.

Simple linked lists are suitable for algorithms which perform frequent sequential *run-throughs*. For those which perform partial *run-throughs* in both directions (forwards and backwards in the sequence of nodes) it is better to use **double linked lists**, these are lists whose nodes contain pointers to the next and the previous elements.

In both cases, if the algorithm performs insertions of new elements and the destruction of unnecessary elements on a frequent basis, it can be more convenient to have the first and last elements linked. These structures are known as ***circular lists*** and, usually, the first elements are marked with a specific piece of data or a special field.

4.2.1. Fundamental list operations

A list of elements must allow us to perform the following operations:

- Accessing a specific node.
- Deleting an existing node.
- Inserting a new node.

In the following sections we will look at these three operations and we will show the programs of the functions used to perform them on a simple linked list.

To access a specific node it is essential to get its address. If we assume we want to get the nth element of a list and return its address, the arguments needed by the corresponding function will be the position of the element being searched and the address of the first element of the list, this could be `NULL` if it is empty.

Evidently, the function will return the address of the nth node or `NULL` if it does not find it:

```
node_t *nth_node( list_t list, unsigned int n )
```

```
{
  while( ( list != NULL ) && ( n != 0 ) ) {
   list = list ->next;
   n = n - 1;
  } /* while */
  return list;
} /* nth_node */
```

In this case the first position will be considered to be position 0 similarly as with vectors in C. It is essential to check that `( list != NULL )` is fulfilled as, if it is not, it will not be able to execute `list = list→next;:` as it can not access the next field of a node which does not exist.

To delete an element from a simple linked list we need to have the address of the previous element as the next field of this one will need to be updated. To do this we will need to extend the previous function so that it returns both the address of the searched element and the previous element. As it needs to return two pieces of data we will need to do it using a pass by reference: we need to pass the addresses of the node pointers which contain the addresses of the nodes:

```
void nth_pq_node(
  list_t    list,   /* Pointer to the first node.                      */
  unsigned int n,   /* Position of the node we are looking for.        */
  node_t **pref,    /* Ref. pointer to previous node.                  */
  node_t **qref)    /* Ref. pointer to current node.                   */
{
  node_t *p, *q;
  p = NULL;         /* The one before the first does not exist         */
  q = list;
  while( ( q != NULL ) && ( n != 0 ) ) {
   p = q;
   q = q->next;
   n = n - 1;
  } /* while */
  *pref = p;
  *qref = q;
} /* nth_pq_node */
```

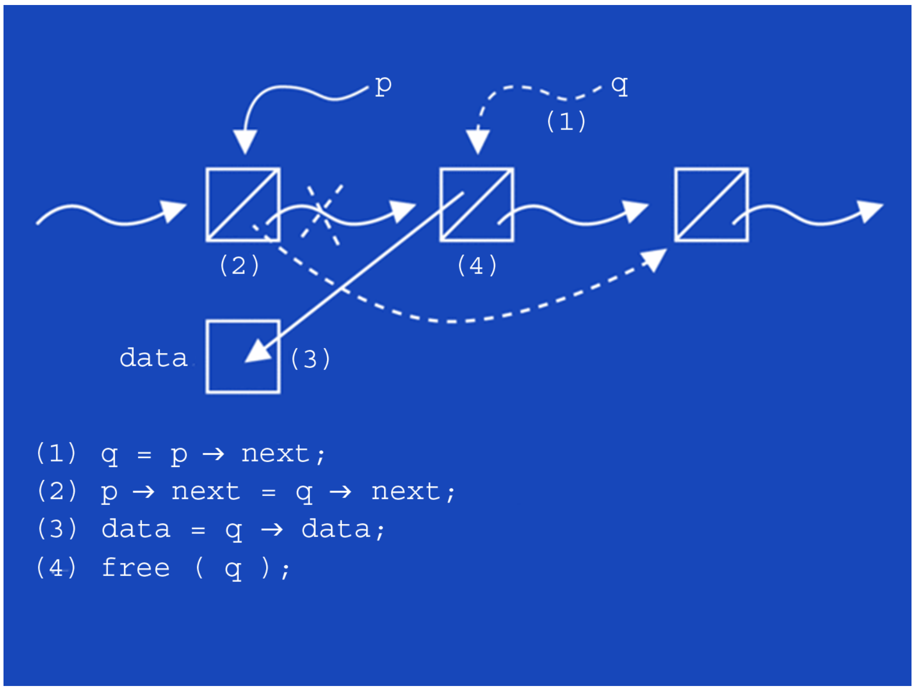
The program of the function that deletes a node will need to know both the address (stored in `q`) and that of the possible previous element (stored in the pointer `p`). In other words, we are tying to delete the element following the one pointed to by `p`.

When writing these programs it is highly recommendable to create a chart of the dynamic data structure, on which you should note the effects of the various modifications to the same.

Therefore, in order to program this function we first need to establish the general case on a chart of the structure from which we wish to remove a node and then program it, always paying attention to the various exceptions which the general case may have. These are usually relating to the treatment of the first and last elements as these do not have a preceding or a following element and, as a consequence, they do not follow the same rules as the others in the list.

The following figure summarises the general procedure for removing the node following node `p`:

Figure 6.



```
(1) q = p → next;
(2) p → next = q → next;
(3) data = q → data;
(4) free ( q );
```

Note

We can obviously not remove a node when `p == NULL`. In this case neither (1) nor (2) can be executed as they imply referencing non-existent variables. This is even more true if `p == NULL` is fulfilled and we want to remove the first element, as this is the only one which does not have a preceding element. We therefore need to "protect" the execution of (1) and (2) using a conditional instruction which decides if the general case can be performed or, conversely, the first element is deleted. Something similar happens with (3) and (4), these can not be executed if `q == NULL`.

The function used to delete the node may be as follows:

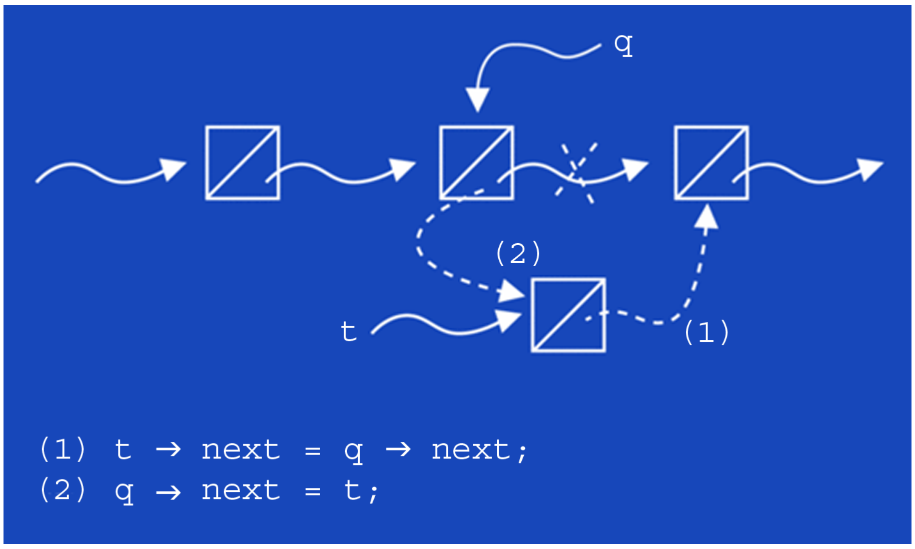
```
int destroy_node(
  list_t *listref,   /* Pointer to reference 1st node.          */
  node_t *p,         /* Pointer to previous node.               */
  node_t *q)         /* Pointer to node to be destroyed.        */
```

```
{
  int data = 0       /* Default data value.     */
  if( p != NULL ) {
   /* q = p->next; (not necessary) */
   p->next = q->next;
  } else {
   if( q != NULL ) *listref = q->next;
  } /* if */
   if( q!= NULL ) {
    data = q->data;
    free( q );
   } /* if */
   return data;
} /* destroy_node */
```

When deleting the first element we need to change the address contained in the list so that it points to the new first element, except when q is also NULL).

To insert a new node whose address is in t, we will need to perform the operations in the following figure:

Figure 7.



Note

In this case, the node t will be inserted after the node q. As we can see, it is necessary that q != NULL to be able to perform an insertion.

The code for this function will be as follows:

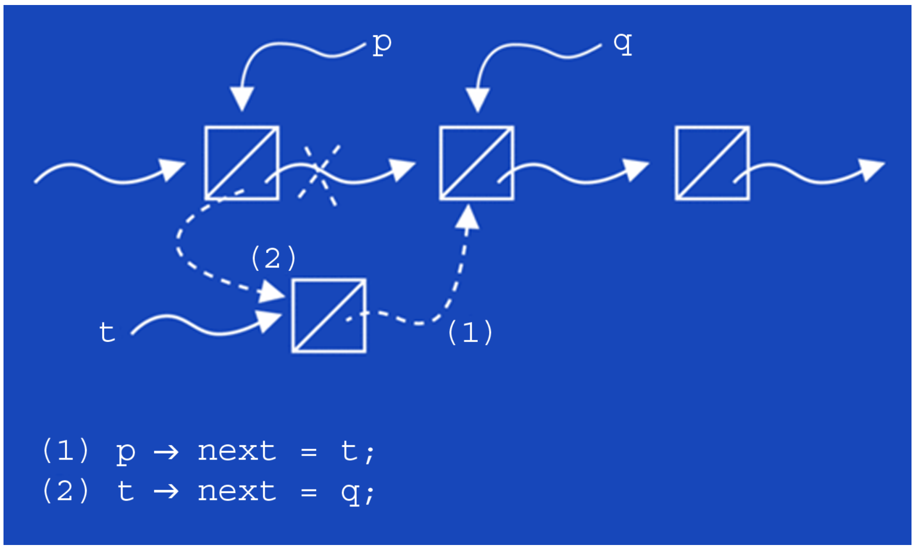
```
void insert_next_node(
  list_t *listref,   /* Pointer to reference 1st node.         */
  node_t *q)         /* Pointer to node at location.           */
  nodo_t *t)         /* Pointer to node to be inserted.        */
{
  if( q!= NULL ) {
   t->next = q->next;
   q->next = t;
  } else { /* The list is empty.                               */
   *listref = t;
  } /* if */
} /* insert_next_node */
```

For this function to be useful we will need to have a function which creates the nodes of the list. In this case:

```
node_t *create_node( int data )
{
  node_t *noderef;
  noderef = (node_t *)malloc( sizeof( node_t ) );
  if( noderef != NULL ) {
   noderef->data = data;
   noderef->next = NULL;
  } /* if */
  return noderef;
} /* create_node */
```

If we are inserting at a specific location we usually insert the new node at the position preceding the one indicated; i.e. occupying the position of the referenced node and so displacing the rest of the nodes one position "to the right". The operations needed to do this in the general case are shown in the following figure:

Figure 8.



Following the indications of the figure above, the code for this function will be as follows:

```
void insert_node(
  list_t *listref,   /* Pointer to reference 1st node.      */
  node_t *p,         /* Pointer to preceding node.          */
  node_t *q)         /* Pointer to node at location.        */
  nodo_t *t)         /* Pointer to node to be inserted.     */
{
  if( p != NULL ) {
   p->next = t;
  } else { /* A new first element is inserted.              */
  *listref = t;
  } /* if */
  t->next = q;
} /* insert_node */
```

Therefore the insertion of a node at the nth position could be constructed as follows:

```
bool insert_nth_list(
  list_t *listref,       /* Pointer to reference 1st node.  */
  unsigned     int n,    /* Position of the insertion node. */
  int          data)     /* Data to insert.                 */
{ /* Returns FALSE if it can not.     */
  node_t *p, *q, *t;
  bool retval;
  t = create_node( data );
  if( t != NULL ) {
   nth_pq_node( *listref, n, &p, &q );
   insert_node( listref, p, q, t );
   retval = TRUE;
  } else {
   retval = FALSE;
  } /* if */
  return retval;
} /* insert_nth_list */
```

We could also write a code for the function of the destruction of the nth element of a list in the same way.

Normally the elements of a list will be more complex than integers and we will need to replace the definition of the data type `node_t` with a more suitable one.

The search criteria for nodes are usually more sophisticated than those for the searching for a certain position. We can therefore use the above functions as a model from which we can derive real applications.

4.2.2. Queues

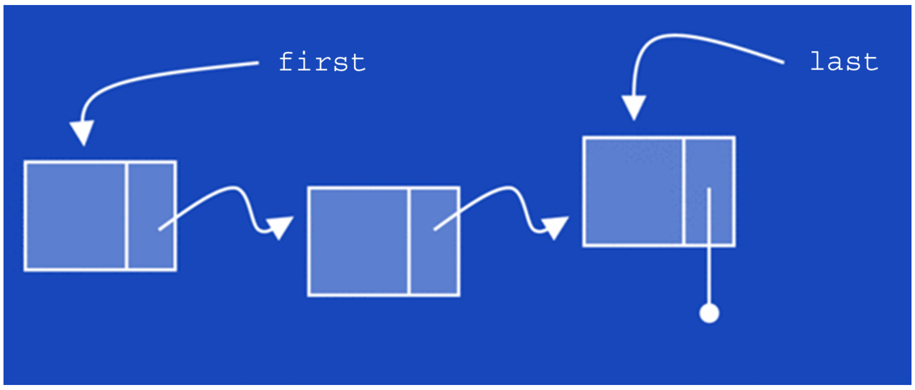
Queues are in fact lists in which elements are inserted at one end and removed from the other. In other words, they are lists in which insertion and deletion operations are restricted to very specific cases. This allows us to manage them much more effectively. It is therefore convenient to have a tuple which will facilitate direct access to both the first and last elements. We can therefore resolve the insertion and removal operations without the need to carry out a search on the list.

This class of queues should therefore have a control tuple such as the following:

```
typedef struct queue_s {
  node_t *first;
  node_t *last;
} queue_t;
```

Graphically:

Figure 9.



To demonstrate the two operations we will assume that the elements of the queue are simple integers, meaning that the queue will be a list of nodes of the `node_t` data type we have looked at before.

An insertion will therefore be as follows:

```
bool enqueue( queue_t *queueref, int dato )
/* Returns FALSE if the data can not be added.*/
{
  node_t *q, *t;
```

```
    bool retval;

    t = create_node( data );
    if( t != NULL ) {
     t->next = NULL;
     q = queueref->last;
     if( q == NULL ) { /* Queue empty: */
      queueref->first = t;
      } else {
       q->next = t;
      } /* if */
      queueref->last = t;
      retval = TRUE;
     } else {
      retval = FALSE;
     } /* if */
     return retval;
  } /* enqueue */
```

And for removal:

```
  bool dequeue( queue_t *queueref, int datoref )
  /* Returns FALSE if the data can not be removed.      */
  {
    node_t *q;
    bool retval;
    q = queueref first;
    if( q!= NULL ) {
     queueref->first = q->next;
     *datoref = destroy_node( &q );
     if( queueref->first == NULL ) { /* Queue empty:    */
      queueref->last = NULL;
     } /* if */
     retval = TRUE;
    } else {
     retval = FALSE;
    } /* if */
    return retval;
  } /* dequeue */
```

The function `destroy_node` for the above deletion is as follows:

```
  int destroy_node( node_t **pref )
  {
    int dato = 0;
    if( *pref != NULL ) {
     dato = (*pref) ->dato;
```

```
      free( *pref );
      *pref = NULL;
    } /* if */
    return dato;
  } /* destroy_node */
```

Queues are often used when resources are shared by many users.

Example

- To manage a printer, the resource is the printer itself and the users are the computers connected to it.
- To control an automatic queuing machine: the resource is the vendor and the clients are the users.

Generally speaking, when we make insertions in a queue we need to be aware of which element is being inserted, i.e. they are not always done at the end, the element can be positioned according to its privileges over the others. In this case we are talking about **priority queues**. In these types of queues, removal is always performed at the beginning but insertion implies positioning the new element at the last position of the elements with the same priority.

There are certainly other more specialised types of list management and, beyond lists, other types of data structures such as trees (a syntax tree for example) and graphs (a road network for example). Unfortunately we do not have time to go into these but we must bear them in mind if the problem data requires their use.

5. Top-down program design

Remember that modular programming is based on dividing the code into sub-programs which perform a specific function. In this unit we will specifically look at how these sub-programs can be grouped according to the tasks they are to perform and how to organise them to improve the programming of the corresponding algorithm.

Complex algorithms are usually made up of programs with many lines of code. Programming therefore needs to be done with great care so that the code is readable and easy to maintain.

5.1. Description

The result of modular programming is a code made up of several programs of a few lines which are linked together using calls. Each of these should be easy to understand and therefore easy to maintain.

The top-down design technique is a technique for designing algorithms in which the main algorithm is resolved by abstracting the details which are then resolved by other algorithms in the same way. This means we are using a higher level of abstraction and, for all those actions which can not be directly translated to an instruction in the chosen programming language, the corresponding algorithms are designed independently from the main one following the same principle.

Top-down design consists of writing programs for algorithms with fewer instructions and implementing non-primitive instructions with functions whose programs follow the same rules as before.

Note

A primitive instruction will be one which can be programmed in a programming language.

In practice, this means we have to design algorithms so that they can be programmed in a completely modular way.

5.2. Example

Top-down program design therefore consists of starting the program with a main algorithm and then refining the "gross" instructions, converting them to sub-programs with instructions which are more "refined". This is where we get the concept of refinement. Obviously, this process will be complete when there are no more "gross" instructions to refine.

In this section we will look at a simple example of top-down design used for resolving a fairly common problem in programming: the ordering of data for easier viewing, for example.

This is one of the most studied problems in computer science and there are several methods for resolving it. One of the simplest ways is to select the element which should start the classification (for example, the largest or the smallest if we are dealing with numbers), putting them in an ordered list and repeating the process with the rest of the elements to be ordered. The main program of this algorithm could be as follows:

```
/* ... */
list_t    pending, ordered;
element_t   element;
/* ... */
initialise_list( &ordered );
while( ! list_is_empty( pending ) ) {
  element= extract_minimum_from_list( &pending );
  put_at_end_of_list( &ordered, element );
} /* while */
/* ... */
```

The above program does not have many primitive C instructions and must therefore be refined. However its function can be easily understood. It is important to understand that the "address of" operators (the & sign) in the call parameters of the call functions indicate that they can modify the content of the same.

The most difficult aspect of the refining process is usually identifying the parts which should be described using primitive instructions, this means determining the different levels of abstraction the algorithm should have and therefore those of the corresponding program. We generally try to make sure that the program reflects the algorithm it came from to the greatest extent possible.

It is a common mistake to think that those operations which only require one or two primitive instructions can never be considered to be non-primitive instructions.

One rule which is easy to adopt is that all operations which are carried out with an abstract data type are equally abstract, meaning they are written using non-primitive instructions (functions).

6. Abstract data types and associated functions

The best way to implement top-down programming is to program all of the operations which can be performed with each of the abstract data types which are to be used. This in fact means creating a virtual machine to execute those instructions of the algorithm in such a way that a language will have all the operations needed for the machine to be able to process it, these are then obviously translated to operations in the language of the real machine which will carry out the processing.

In the example of the ordering algorithm above there are two abstract data types (`list_t` and `element_t`) for which we need, at least, the following operations:

```
void initialise_list( list_t *ref_list );
bool list_is_empty( list_t list );
element_t extract_minimum_from_list( list_t *ref_list );
void put_at_end_of_list( list_t *rlst, element_t e );
```

As we can see, there are no operations affecting the data type `element_t`. However, we can be sure that the program will make use of them (data reading, insertion of the same in the list, comparisons between elements, writing results etc.) in another section. We will therefore need to program the corresponding operations. In particular, if we look at the following code we can see that it contains functions to handle data of the type `element_t`:

```
element_t extract_minimum_from_list( list_t *ref_list )
{
  ref_node_t current, minimum;
  bool       is_lower;
  element_t small;
  start_of_list( ref_list );
  if( list_is_empty( *ref_list ) ) {
   initialise_element( &small );
  } else {
   minimum = ref_node_of_list( *ref_list );
   small = element_in_ref_node( *ref_list, minimum );
   advance_position_in_list( ref_list );
   while( !end_of_list( *ref_list ) ) {
    actual = ref_node_of_list( *ref_list );
    is_lower = compare_elements(
     element_in_ref_node( *ref_list, current ), small
    ); /* compare_elements */
    if( is_lower ) {
        minimum = current;
        small = element_in_ref_node( *ref_list, minimum );
       } /* if */
       advance_position_in_list( ref_list );
     } /* while */
     show_element( small );
     remove_from_list( ref_list, minimum );
   } /* if */
   return small;
} /* extract_minimum_from_list */
```

As we can see from the code above, at least two operations are required for data of the type `element_t`:

```
initialise_element();
compare_elements();
```

Four more operations are necessary for lists:

```
start_of_list();
end_of_list();
advance_position_in_list();
remove_from_list();
```

We have also added the data type `ref_node_t` to have the references of the nodes in the lists and two operations: `ref_node_of_list` to obtain the reference of a certain node in the list and `element_in_ref_node` to obtain the element which is stored in the node indicated.

This shows that progressive refinement serves to ascertain which operations are needed for each data type and also shows that lists have a different level of abstraction which is higher than that of the elements.

To complete the programming of the ordering algorithm we need to develop all the functions associated with the lists and then those associated with the elements. However the first thing we need to do is to determine the abstract data types which will be used.

Lists can be created using vectors or dynamic variables depending on the types of algorithms being used. In the ordering example it will depend in part on the general criteria applied for making the decision and also on the characteristics of the algorithm itself. Generally we will use vectors if they do not cause too much wastage, but we also need to bear in mind that the algorithm we are using can only be applied to the classification of modest amounts of data (a few hundred at most).

If we choose the first option, the data type `list_t` would be:

```
#define MAXIMUM_LENGTH 100
typedef struct list_e {
  element_t    node[ MAXIMUM_LENGTH ];
  unsigned short position; /* Current access position.  */
  unsigned short quantity; /* Length of the list.       */
} list_t;
```

We will also need to define the data type for node references:

```
typedef unsigned short ref_node_t;
```

The other operations required will be those corresponding to the following functions:

```
void initialise_list( list_t *ref_list )
{
  (*ref_list).quantity = 0;
  (*ref_list).position = 0;
} /* initialise_list */

bool list_is_empty( list_t list )
{
  return ( list.quantity == 0 );
} /* list_is_empty */

bool end_of_list( list_t list )
{
  return ( list_position == list.quantity );
} /* end_of_list */

void start_of_list( list_t *list_ref )
{
  list_ref->position = 0;
} /* start_of_list */

ref_node_t ref_node_of_list( list_t list )
{
  return list.position;
} /* ref_node_of_list */

element_t element_in_ref_node(
  list_t list,
  ref_node_t refnode)
{
  return list.node[ refnode ];
} /* element_in_ref_node */

void advance_position_in_list( list_t *list_ref )
{
  if( !end_of_list( *list_ref ) ) {
   (*list_ref).position = (*list_ref).position + 1;
  } /* if */
} /* advance_position_in_list */

element_t remove_from_list(
  list_t *ref_list,
  ref_node_t refnode)
{
  element_t removed;
  ref_node_t pos, last;

  if( list_is_empty( *ref_list ) ) {
  initialise_element( &removed );
  } else {
   removed = (*ref_list).node[ refnode ];
   last = (*ref_list).quantity - 1;
   for( pos = refnode; pos < last; pos = pos + 1 ) {
    (*ref_list).node[pos] = (*ref_list).node[pos+1];
   } /* for */
   (*ref_list).quantity = (*ref_list).quantity - 1;
  } /* if */
  return removed;
} /* remove_from_list */

element_t extract_minimum_from_list( list_t *ref_list );

void put_at_end_of_list(
  list_t *ref_list,
  element_t element )
{
  if( (*ref_list).quantity < MAXIMUM_LENGTH ) {
   (*ref_list).node[(*ref_list).quantity] = element;
   (*ref_list).quantity = (*ref_list).quantity +1;
   } /* if */
} /* put_at_end_of_list */
```

If we examine the above functions associated with the data type `list_t`, we will see that they only need one operation using the data type `element_t`: `compare_elements`. To complete the ordering program we now only need to define the data type and the comparison operation.

While list operations on vectors are generic, all those affecting elements will depend on the information whose data we wish to order.

For example, let us suppose that we want to order the results of an exam by a national identity number, then the data type for the elements could be as follows:

```
typedef struct element_s {
       unsigned int national_identity_number;
       float           score;
} data_t, *element_t;
```

The comparison function would be as follows:

```
bool compare_elements(
  element_t lower,
  element_t higher )
{
  return ( lower->national_identity_number < higher->national_identity_number );
} /* compare_elements */
```

The initialisation function would be as follows:

```
void initialise_element( element_t *ref_elem )
{
  *ref_elem = NULL;
} /* initialise_element */
```

Note that the elements are in fact pointers to dynamic variables. This will greatly simplify the code of the operations on higher levels of abstraction despite the fact that the program will need to construct and destroy them. However, it is important to remember that we always need to prepare functions for the creation, destruction, copying and duplication of each of the data types for which dynamic variables exist.

The copy and duplication functions are necessary as a simple assignment means copying the address of a variable to another pointer, meaning we would have two references to the same variable instead of two different variables with the same content.

Example

Look at the difference between these two functions:

```
/* ... */
element_t original, other, copy;
/* ... */
other = original; /* Copy of pointers. */
/* the address stored in 'other'
 is the same as that contained in 'original'
*/

/* ... */
copy_element( original, copy );
other = duplicate_element( original );
/* the addresses stored in 'copy' and in 'other'
 are different from that contained in 'original'
*/
```

Copying content should be done using a specific function and if the variable which should contain the copy has not been created, we will obviously need to create it first by making a duplicate.

To summarise, when we program an algorithm using top-down design, we will need to program the creation, destruction and copy functions for each of the abstract data types it contains. We will also need to program the functions for all operations used with each data type to reflect the different levels of abstraction present in the algorithm. This will allow us to write intelligible and easy-to-handle code although we may need to write a few more lines.

7. Header files

It is common that several programming teams collaborate on the writing of a program although this is sometimes more a wish than a reality. In small and medium-sized businesses the team will often be made up of just one person and sometimes the number of teams will be reduced from several to just one. However, the following sentence is true: programs must be created using parts of other programs if we are using good programming practices. The re-use of programs does not just reduce development time but also means we can be sure that the components of the program have been proven to work.

All this is especially true for free software in which programs are usually produced by groups of diverse programmers who are not formally coordinated: a programmer may have used code created by other programs for applications which it was not originally intended for.

To be able to use a certain code we will need to remove the implementation data, indicating only the data type for which it is intended and which operations can be performed with the corresponding variables. We therefore need to have a file in which the abstract data types are defined and where we declare the functions which are provided for the variables of the same. These files are known as header files as they appear at the beginning of the source code of the functions whose declarations or headers have been included in the same.

Header files will tell us all we need to know about the abstract data type and the functions used to manage it.

7.1. Structure

Header files have the extension ".h" and the content must be organised in such a way that it is easy to read. First we include a comments section which states the nature of the content and, above all, the functionality of the code in the corresponding ".c" file. We then follow the standard structure of a typical C program: the inclusion of header files, the definition of symbolic constants and, lastly, the declaration of the functions.

The definition must always be in the ".c" file.

The following header file is used to operate with complex numbers and is a good example:

```
/* File:   complex.h                                            */
/* Content: Functions for operating with complex                */
/*            numbers of type (X + iY) in which                 */
/*            X is the real part and Y the imaginary one.       */
/* Version: 0.0 (original)                                      */

#ifndef _COMPLEX_NUMBERS_H_
#define _COMPLEX_NUMBERS_H_

#include <stdio.h>
#define PRECISION 1E-10
typedef struct complex_s {
       double real, imaginary;
} *complex_t;
complex_t new_complex( double real, double imaginary );
void delete_complex( complex_t complex );
void print_complex( FILE *file, complex_t complex );
double complex_module( complex_t complex );
complex_t opposed_complex( complex_t complex );
complex_t sum_complexes( complex_t c1, complex_t c2 );
/* etcétera */

#endif /* _COMPLEX_NUMBERS_H_ */
```

The definitions of types and constants and function declarations have been included as the body of the preprocessor command `#ifndef` ... `#endif`. This command asks if a certain constant has been defined and, if it hasn't, the content of the file up to the end marker is transferred to the compiler. The first command of the body of this conditional command is specifically there to define the constant `_COMPLEX_NUMBERS_H_` to prevent a new inclusion of the same file from generating the same source code for the compiler (it is not required if it has already been processed once).

These so-called ***conditional compilation commands*** of the preprocessor allow us to decide if a certain tranche of source code is to be supplied to the compiler or not. These are summarised in the following table:

Table 8.

Command	Meaning
`#if expression`	The following lines are compiled if `expression` ≠ `0`.
`#ifdef SYMBOL`	The following lines are compiled if `SYMBOL` is defined.
`#ifndef SYMBOL`	The following lines are compiled if `SYMBOL` is not defined.

Command	Meaning
`#else`	Finalises the compiled block if the condition is fulfilled and initiates the block to be compiled if not.
`#elif expression`	Chains an `else` with an `if`.
`#endif`	Indicates the end of the conditional compilation block.

Note

A defined symbol (for example: `SYMBOL`) can be cancelled using:

```
#undef SYMBOL
```

and from this moment on it is considered to be not defined.

The forms:

```
#ifdef SYMBOL
#ifndef SYMBOL
```

are abbreviations of:

```
#if defined( SYMBOL )
#if !defined( SYMBOL )
```

respectively. The function `defined` can be used in more complex logical expressions.

To finish off this section, it only remains to say that the source code file should obviously include its header file:

```
/* File: complex.c                         */
/* ... */
#include "complex.h"
/* ... */
```

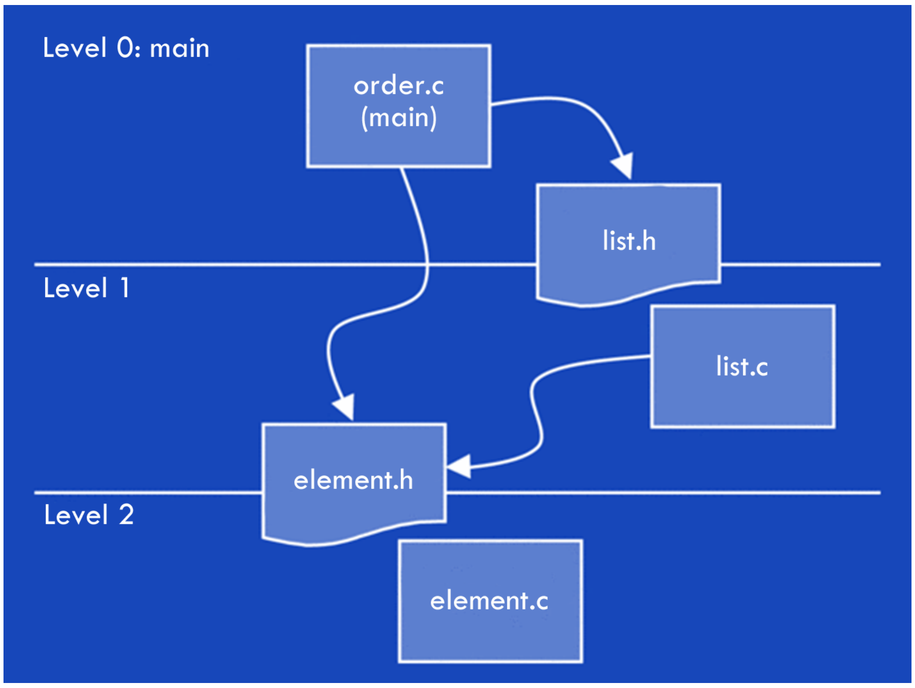
Note

Inclusion is done by indicating the file between inverted commas instead of angle brackets ("greater than" and "less than" symbols) as it is assumed that the included file is to be found in the same directory as the file which includes the inclusion directive. Angle brackets should be used if we want the preprocessor to examine the set of access paths to standard inclusion file directories, as is the case with `stdio.h,` for example.

7.2. Example

In the example of ordering by selection, we would have a header file for each abstract data type and the corresponding files containing the C code, obviously. The following figure shows a diagram of the relationships between these files:

Figure 10.



Each source code file can be compiled independently. This means in this example there would be three compilation units. Each of these could be developed independently of the others. The only aspect we must respect in terms of units which have header files is that the function declarations and data types are not modified. Therefore the other units which use them should not modify their calls and will therefore not need to be changed or compiled.

It is important to remember there can only be one unit with a `main` function (the one corresponding to `order.c`, in the above example). The others should be compendiums of functions associated with a certain type of abstract data.

The use of header files also allows us to change the code of a function without having to change the code of any of the programs that use it. It is clear that the change will not affect the "contract" established in the corresponding header file.

The content of the header file describes the functionality provided for a certain type of abstract data and, with this description, it is committed to maintain it independently of how it appears in the associated code.

To illustrate this idea we can imagine that, in this example, it is possible to change the code of the functions which work with the lists to make them dynamic, without changing the contract acquired in the header file.

We will now look at the header file for lists in the case of ordering:

```
/* File: list.h                                         */
```

```
#ifndef _LIST_VEC_H_
#define _LIST_VEC_H_

#include <stdio.h>
#include "bool.h"
#include "element.h"

#define MAXIMUM_LENGTH 100
typedef struct list_e {
  element_t    node[ MAXIMUM_LENGTH ];
  unsigned short position; /* Current access position.   */
  unsigned short quantity; /* Length of the list.        */
} list_t;

void initialise_list( list_t *ref_list );
bool list_is_empty( list_t list );
bool end_of_list( list_t list ):
void start_of_list( list_t *list_ref );
ref_node_t ref_node_of_list( list_t list );
element_t element_in_ref_node(
  list_t list,
  ref_node_t refnode
);
void advance_position_in_list( list_t *list_ref );
element_t extract_minimum_from_list( list_t *ref_list );
void put_at_end_of_list(
  list_t *ref_list,
  element_t element
);

#endif /* _LIST_VEC_H_ */
```

As we can see, we change the extraction method for the minimum element without needing to modify the header file nor the call in the main program. However, a change in the data type used to implement lists using dynamic variables, for example, would involve re-compiling all of the units which use them, although the headers of the functions are not modified. In these cases it is useful to maintain the contract with respect to these as this will make it unnecessary having to modify the source code of the units which use them.

8. Libraries

Libraries of functions are in fact just compilation units. As such, each one will have a header file and a source code file. To prevent the repeated compilation of libraries the source code has already been compiled (files with the extension ".o") and they only need to be linked to the main program.

Function libraries are different from compilation units in that they only incorporate those functions which are needed in the main program: those which are not used are not included. Compiled files which allow this option have the extension ".l".

8.1. Creation

To obtain a compilation unit in which only the functions used are included in the executable program, we need to compile it to obtain an object type file in which the executable code is not linked.

```
$ gcc -c -o library.o library.c
```

We can assume that the file `library.c` will include the appropriate header file as indicated.

Once the object file has been generated we need to include it in an archive (with the extension ".a") of files of the same type (it could be the only one if the library only has a single compilation unit). In this context "archives" are collections of object files contained in a single file with an index to locate them and, above all, to determine which parts of the object code correspond to which functions. To create an archive we need to execute the following command:

```
$ ar library.a library.o
```

To construct the index (the table of symbols and locations) we need to execute the command:

```
$ ar -s library.a
```

or:

```
$ ranlib library.a
```

The archive management command ar also allows us to list the object files it contains and to add or replace new or modified files, to update them (replacement is performed if the modification date is later than the date of inclusion in the archive) and also to remove them if they are not required. These are done using the following commands respectively:

```
$ ar -t library.a
$ ar -r new.o library.a
$ ar -u updateable.o library.a
$ ar -d obsolete.o library.a
```

With the information from the table of symbols the linker mounts an executable program using only those functions which are referenced. In other ways archives are similar to single object code files.

8.2. Use

Library functions are used in exactly the same way as the functions of any other compilation unit.

We only need to include the appropriate header file and incorporate the required function calls within the source code.

8.3. Example

The ordering example included a compilation unit for lists. As lists are a commonly-used dynamic data type it is useful to have a library of functions for operating with them. This will mean we do not need to program them again later on.

To transform the list unit into a function library we need to make sure that the data type does not depend on the application in any way. If it does we will need to compile the lists unit for each new program.

In the example, the lists contain elements of the type element_t, which was a pointer to data_t. In general, lists can have elements which are pointers to any data type. Whichever they are, they are all memory addresses. For this reason, in respect of lists, the elements will be a void type which the user of the function library will need to define. The compilation unit for lists will therefore include:

```
typedef void *element_t;
```

Therefore, to carry out the smallest element extraction function we will need to know which function to call to perform the comparison. We will therefore need to add another parameter for the comparison function:

```
element_t extract_minimum_from_list(
  list_t *ref_list,
  bool (*compare_elements)( element_t, element_t )
); /* extract_minimum_from_list */
```

The second argument is a pointer to a function which takes two elements as parameters (we do not need to name the formal parameters) which returns a logic value of the type `bool`. In the source code for the definition of the function the call will be made in the following way:

```
/* ... */
is_lower = (*compare_elements)(
         element_in_ref_node( list, current ),
         small
); /* compare_elements */
/* ... */
```

The rest does not need to be changed at all.

the decision to transform a compilation unit of a program in a library will therefore basically depend on two factors:

- The data type and the operations must be able to be used in other programs.
- Rarely will all the functions in a unit be used.

9. The tool *make*

The compilation of a program normally involves compiling some of its units and then linking them all to the library functions in order to mount the final executable program. Therefore, as well as carrying out a series of commands, we also need to know which files have been modified.

The *make* tools allow us to establish the relationships between files in order to determine which ones depend on others. When it detects that one of the files has a modification time and data which is earlier than one of the ones it depends on, the command is executed to regenerate them.

We therefore do not need to worry about which files we need to generate and which do not need to be updated. We also avoid the need to have execute a series of individual commands, which could be considerable in number in a large program.

The purpose of *make* tools is to automatically determine which parts of a program need to be recompiled and then to execute the pertinent commands.

The `gmake` tool (or simply `make`) is a *make* utility from GNU (www.gnu.org/software/make) which deals with the issues mentioned above. To use it we need to have a file which, by default, is called `makefile`. We can indicate a file with another name if we invoke it using the option `-f`:

```
$ make -f targets_file
```

9.1. File *makefile*

In the file *makefile* we need to specify the targets (usually files to be constructed) and the files they depend on (the pre-requisites for fulfilling the targets). For each target we will need to construct a rule whose structure will be as follows:

```
# Syntax of a rule:
target : file1 file2 ... fileN
  command1
  command2
  ...
  commandK
```

The # symbol is used to introduce a comments line. Every rule requires that we indicate what the target is, followed by a colon, followed by the files it depends on. The following lines include the commands which need to be executed. If you want to continue a line it should end with a line feed preceded by the escape character or backslash (\).

Note

The first of these lines must begin with a tab character.

Using the exam results ordering program we could construct a *makefile* such as the one shown below to simplify updating of the final executable code:

```
# Compilation of the ordering program:
classifies : orders.o marks.o list.a
  gcc -g -o classifies orders.o marks.o list.a
orders.o : orders.c
  gcc -g -c -o orders.o orders.c
marks.o : marks.c marks.h
  gcc -g -c -o marks.o marks.c
list.a : list.o
  ar -r list.a list.o ;
  ranlib list.a
list.o : list.c list.h
  gcc -g -c -o list.o list.c
```

The *make* tool processes the previous file revising the pre-requisites of the first target and, if these are also targets of other rules, it will proceed in the same way as for the others. Any terminal prerequisite (one which is not a target of another rule) which was modified after the target affected by it will cause a series of specific commands to be executed in the following lines of the rule.

It is possible to specify more than one target but `make` will only examine the rules of the first end target it finds. If we want it to process other targets we will need to indicate them in the invocation.

We can also have targets without prerequisites in which case the commands associated with the rule in which they appear will always be executed.

It is possible to have a target which deletes all the object files which are no longer needed.

Example

Using the above example it would be possible to clean any unnecessary files from the file directory by adding the following text to the end of the *makefile*:

```
# Cleaning the working directory:
clean :
  rm -f orders.o marks.o list.o
```

To achieve this target we just need to add the command:

```
$ make clean
```

We can also indicate several targets with the same prerequisites:

```
# Compilation of the ordering program:
all clean optimal : classifies
clean : CFLAGS := -g
optimal : CFLAGS := -O
classifies : orders.o marks.o list.a
  gcc $(CFLAGS) -o classifies orders.o marks.o list.a
# rest of the file ...
```

Note

From the above file we can see the following:

- If we invoke `make` without an argument, it will update the target `all` (the first one it finds), which depends on updating the secondary target `classifies`.
- If we specify the target `clean` or `optimal`, we will need to check that the two rules are consistent: the first indicates that to achieve this we need to update `classifies` and the second that we need to assign `CCFLAGS` with a value.

Given that *make* first analyses the dependencies and then executes the appropriate commands, the result will be that the mounting of *compiles* will be done with the content assigned to the variable `CCFLAGS` in the preceding rule: accessing the content of a variable is done using the corresponding operator which is represented by the dollar symbol.

In the above example we can appreciate that the usefulness of *make* goes much further than the scope we have looked at here and also that *makefile* has many ways of expressing rules more powerfully. All the same, we have still looked at their main characteristics from a programming point of view and we have also looked at some important examples.

10. Relation with the operating system. passing parameters to programs

Programs are translated into machine code so that they can be executed. However, many operations relating to input and output devices and storage devices (disks and memory among others) are translated to call functions of the operating system.

One of the functions of the operating system is precisely to allow the execution of other programs on the machine and software in particular. To do this, it provides the user with certain mechanisms for selecting the applications to be executed. Most of these are currently graphic user interfaces. Even so, we still have the text environments of the command interpreters, which are commonly known as *shells*.

When using *shells* to execute programs (some of these being utilities of the actual system and some of them applications) we need to supply the name of the corresponding executable file. It is also common for one program to be invoked by another.

In fact, the `main` function in C programs can have different arguments. In particular, there is a convention that the first parameter is the number of arguments passed to it through the second, which will be a character string vector. To clarify how the parameter passing procedure works, we can use the following program as an example when we invoke it from a *shell* with a certain number of arguments:

```
/* File: args.c */
#include <stdio.h>
int main( int argc, char *argv[] )
{
  int i;
  printf( "Num arguments, argc = %i\n", argc );
  for( i = 0; i < argc; i = i + 1 ) {
   printf( "Argument argv[%i] = \"%s\"\n", i, argv[i] );
  } /* for */
  return argc;
} /* main */
```

If we invoke it with the following command the result will be as shown below:

```
$ args -test number 1

Num arguments, argc = 4
```

```
Argument argv[0] = "args"
Argument argv[1] = "-test"
Argument argv[2] = "number"
Argument argv[3] = "1"
$
```

It is important to bear in mind that that the invocation of the argument is taken to be argument 0 of the command. Therefore, in the above program, the invocation in which three parameters are given to the program becomes a call to the function `main` which also includes the text used to invoke the actual program, this is the first character string of the arguments vector.

We can check the value returned by `main` to determine if an error occurred during the execution process or not. The general convention is that the value returned should be the corresponding error code, or 0 if no errors occurred.

Note

In the source code of the function we can use the constants `EXIT_FAILURE` and `EXIT_SUCCESS`, these are defined in `stdlib.h` and show the return value with or without errors respectively.

The following example shows a program which gives the sum of all the parameters present in the invocation. To do this, it uses the function `atof`, declared in `stdlib.h`, which converts text strings to real numbers. If the string does not represent a number, it will return a zero:

Note

In this case, the program will always return the value 0 as there are no execution errors.

```
/* File: sum.c */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
  float sum = 0.0;
  int   pos = 1;

  while( pos < argc ) {
   sum = sum + atof( argv[ pos ] );
   pos = pos + 1;
  } /* while */
  printf( " = %g\n", sum );
  return 0;
} /* main */
```

11. The execution of functions in the operating system

An operating system is a piece of software which allows the applications executed on a computer to be abstracted from the workings of the machine. This means that applications can run on a virtual machine which is able to perform operations which the real machine cannot understand.

What is more, the fact that the programs are defined in terms of routines (or functions) provided by the OS increases their portability, or their ability to be executed on different machines. This makes them fairly independent from the machine, but obviously not from the OS.

In C, many of the standard library functions use OS routines to perform their tasks. Among these functions are data and file input and output and memory management (dynamic variables above all).

In general, all the standard library functions in C have the same header and the same behaviour, independently of the operating system, however some of them do depend on the operating system: there are some differences between Linux and Microsoft. Fortunately, these are easy to detect as the functions which are related to a specific operating system are declared in specific header files.

It is almost always more appropriate to execute the commands of the *shell* of the operating system instead of directly executing the functions. Among other things, this allows the corresponding program to be described at a higher level of abstraction meaning that we can take advantage of the fact that it is the command interpreter itself which fills in the details needed for the task to be completed. In general these deal with the execution of internal commands of the *shell* itself or those which use its resources (search paths and environment variables among others) to execute other programs.

To be able to execute a *shell* command we only need to give the `system` function the character string which describes it. The value returned is the return code of the command executed, or 1 if there was an error.

In Linux, `system( command )` executes `/bin/sh –c command;` meaning it uses `sh` as the command interpreter. These will therefore have to follow the syntax of the same.

The following program shows the code returned by the execution of a command, which must be inserted between inverted commas as the argument of the program.

```
/* File: execute.c */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
  int code;

  if( argc == 2 ) {
   code = system( argv[1] );
   printf( "%i = %s\n", code, argv[1] );
  } else {
   printf( "Use: execute \"command\"\n" );
  } /* if */
  return 0;
} /* main */
```

Even though it is simple the above program gives us an idea of how to use the function `system`.

The set of routines and services provided by the operating system does more than just support data input and output functions and memory and file management, it also allows us to launch the execution of programs within others. In the following section we will look into the subject of program execution in more depth.

12. Process management

Today's operating systems, as well as being able to do all the things we have seen, are able to execute a variety of programs on the same machine at the same time. Obviously, this is only possible if they are executed on different processors or if they are executed either sequentially or alternately on the same processor.

Normally, despite the fact that operating systems are able to manage a machine with several processors, there will be more programs to execute than there are resources to do so. For this reason, we will always need to plan the execution of a specific set of programs on a single processor. Planning can be:

- Sequential. The following program is executed once the first has finished (also known as *batch execution*).

- Interleaved. Each program has a certain amount of time in which to carry out part of its execution flow of instructions, at the end of this period a part of another program is executed.

In this way, we can execute several programs during the same time period and it will look as if the programs were being executed simultaneously.

By using the services (functions) provided by the OS in respect of the execution of software, we can, among other things, execute it when it is disassociated from the standard input/output and/or part of the execution flow of instructions on several parallels.

12.1. Definition of a process

With respect to the operating system, each instruction flow it must manage is a process. It will therefore share the execution between the various processors in the machine and over the time required to carry out the tasks progressively. There will however be some processes which are split into two parallel processes, meaning that from this point on the program will consist of two different processes.

To start with, each process will have a standard data input and output associated with it, but which can be disassociated for the continuation of the program in the background, this is common in background processes which we will look at in the next section.

Processes which share the same environment or state, with the obvious exception of the reference to the next instruction, are called *threads*, while those which have different environments are simply called *processes*.

A program can therefore organise its code in such a way that it performs its task using several parallel instruction flows, whether these are single threads or complete processes.

12.2. Background processes

A background process is one which is executed indefinitely on a machine. They are usually processes which deal with the automated management of data input and output and therefore users do not really interact with them.

Many of the applications which work with the client-server model are constructed using background processes for the server and with interactive processes for the clients. Some clear examples of these applications are those relating to the Internet: The clients are programs, such as a browser or an email manager, and the servers are programs which deal with the requests of the corresponding clients.

In Linux, background processes are known, dramatically, as daemons because, although they are not visible to users as they do not interact with them (especially not through the standard terminal), they do exist: daemons are "spirits" of the machine, users can not see them but are able to perceive their effects.

Note

Literally speaking, a daemon is a malign spirit, although we are assuming that these processes are not.

To create a function we just need to call the function `daemon`, declared in `unistd.h`, using the appropriate parameters. The first argument indicates if the working directory has not changed and the second, if the standard input/output terminal has not been disassociated; in other words, a common call would have to be as follows:

```
/* ... */
if( daemon( FALSE, FALSE ) ) == 0 ) {
  /* body */
} /* if */
/* rest of the program, whether it has been created or not. */
```

Note

This call makes the body of the program a daemon which works in the root directory (as if we had done a `cd /`) and which is disassociated from the standard inputs and outputs (redirected to the null device in fact: `/dev/null`). The function returns the error code 0 if everything worked.

12.2.1. Example

To illustrate the function of daemons we will look at a program which tells the user that a certain amount of time has passed. To do this we will need to invoke the program with two parameters: one to indicate the hours and minutes which must have passed before advising the user and another containing the associated text. In this case the program becomes a non-disassociated daemon of the standard input/output terminal given that the message will appear on the same.

```
/* File: alarm.c                                    */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
  unsigned int hours;
  unsigned int minutes;
  unsigned int seconds;
  cha          *message, *separator;

  if( argc == 3 ) {
   separator = strchr( argv[1], ':' );
    if( separator != NULL ) {
     hours = atoi( argv[1] );
     minutes = atoi( separator+1 );
    } else {
     hours = 0,
minutes = atoi( argv[1] );
    } /* if */
    seconds = (hours*60 + minutes) * 60;
    message = argv[2];
    if( daemon( FALSE, TRUE ) ) {
     printf( "The alarm can not be installed :-(\n" );
    } else {
     printf( "Alarm in %i hours and %i minutes.\n",
      hours, minutes
     ); /* printf */
     printf( "Do $ kill %li to stop it.\n",
      getpid()
     ); /* printf */
    } /* if */
    sleep( seconds );
    printf( "%s\007\n", message );
```

```
      printf( "Alarm stopped.\n" );
     } else {
      printf( "Use: %s hours:minutes \"message\"\n", argv[0] );
     } /* if */
     return 0;
  } /* main */
```

The reading of the input parameters occupies a large part of the code. What we should look at more carefully is the extraction of the hours and minutes, to do this it looks for the colon (with `strchr`, declared in `string.h`) then it takes the integer string to determine the value for the hours and the string after the colon for the minutes.

The program then waits by calling the function `sleep`, the argument for which is the number of seconds for which the program should "sleep", or suspend execution.

Finally, in order to be able to stop the alarm, we give it the command which should be inserted in the *shell* to "kill" the process (meaning to end its execution). To do this, it uses the process number corresponding to the daemon installed. This identifier is obtained by calling the function `getpid()`, where PID stands for process identifier.

One of the fundamental uses of daemons is in the implementation of service provider processes.

12.2.3. Concurrent processes

Concurrent processes are those which are executed *simultaneously* in a single system. In this context, simultaneously refers to the fact that they are performed during the same time period in different processors or are distributed in time on a single processor, or both.

Distributing the execution of a program over several concurrent instruction flows has the following goals:

- Using all the resources of a multi-processor system. By executing each flow of instructions in a different processor, the program runs faster. In fact it is only in this example that the execution processes are genuinely simultaneous.

Note

When two or more processes are sharing the same processor, there is no option but to execute them in alternating sections with a specific duration, each program will advance a certain amount during each period.

- Increasing performance with respect to data input/output. To increase the I/O performance of a program, it is desirable that one of the processes deals with data input, another with the calculations needed to be performed on it and, lastly, another dealing with the output. In this way we can perform the calculations without stopping to output data or waiting for new data to be input. We do not always need to partition a program in this way and the number of processes will vary depending on the needs of the program. In this case we are separating processes which are slower (for example, those which need to communicate with others either to receive or to transmit data) from those which are faster, or those dealing more with calculations.

In the following sections we will look at several examples of concurrent programming using both "light processes" (threads) and complete or "heavy" processes (those which are not light).

13. Threads

A thread is a process which shares the environment with others from the same program, meaning that the memory space is the same. Therefore, the creation of a new thread only implies having the information on the state of the processor and the next instruction for the same. It is for this reason that they are known as "light processes".

Threads are individual instruction execution flows which are closely related.

To use them in C on Linux we need to make calls to the POSIX standard thread functions. This standard defines a portable interface for operating systems (originally for Unix) for computational environments, it takes its name from the acronym.

POSIX thread functions are declared in the file `pthread.h` and the corresponding library archive should be linked to the program. To do this we need to compile it using the following command:

```
$ gcc -o executable code.c -lpthread
```

Note

The option `–lpthread` tells the linker that it should also include the library of POSIX functions for threads.

13.1. Example

We will look at a program to determine if a number is prime or not and which has been unwound into two threads. The main thread is in charge of searching for possible divisors and the second acts as the "observer" for the user: this will read the data which manages the main thread and present it on the standard output terminal. This is possible because they share the main space in memory.

The creation of threads requires that their code be within a function which only allows for one parameter of the type `(void *)`. In fact, functions created to be POSIX threads must obey the following header:

```
(void *)thread( void *reference_parameters );
```

It is therefore necessary to place all the information to be made visible to the user in a tuple and pass its address as a parameter of the same. We will now define the tuple of elements which will show:

```
/* ... */
typedef struct s_visible {
  unsigned long number;
  unsigned long divisor;
  bool end;
} t_visible;
/* ... */
```

Note

The field end indicates to the child thread that the main thread (the parent) has finished its task. In this case it will have determined if the number is prime or not.

The function of the child thread will be as follows:

```
/* ... */
void *observer( void *parameter )
{
  t_visible *ref_view;

  ref_view = (t_visible *)parameter;
  printf( " ... checking %012lu", 0 );
  do {
   printf( "\b\b\b\b\b\b\b\b\b\b\b\b" );
   printf( "%12lu", ref_view->divisor );
  } while( !(ref_view->end) );
  printf( "\n" );
  return NULL;
} /* observer */
/* ... */
```

Note

The character '\b' corresponds to backspace and, given that the numbers are printed using 12 digits (the zeros to the left are shown as spaces), the printing of 12 backspaces will delete the number which was previously printed.

To create the observer thread we need to call `pthread_create()` using the appropriate parameters. From this moment on a new thread will execute concurrently with the program code.

```
/* ... */
int main( int argc, char *argv[] )
{
  int        error_code;     /* Error code to be returned.       */
  pthread_t  id_thread;      /* Thread identifier.               */
  t_visible  view;           /* Observable data.                 */
  bool       result;         /* Indicates whether it is prime.   */
{
  int        error_code;     /* Error code to be returned.       */
  pthread_t  id_thread;      /* Thread identifier.               */
  t_visible  view;           /* Observable data.                 */
  bool       result;         /* Indicates whether it is prime.   */

  if( argc == 2 ) {
   view.number = atol( argv[1] );
   view.end = FALSE;
   error_code = pthread_create(
   &id_thread,              /* Reference at which to place the ID.*/
   NULL,                    /* Reference to possible attributes.  */
   observer,                /* Function which executes the thread.*/
  (void *)&view             /* Argument of the function.          */
   ); /* pthread_create */
   if( error_code == 0 ) {
    result = is_prime( &view );
    view.end = TRUE;
    pthread_join( id_thread, NULL );
    if( result )printf( "It is prime.\n" );
    else printf( "It is not prime.\n" );
    error_code = 0;
    } else {
    printf( "I could not create an observer thread!\n" );
    error_code = 1;
   } /* if */
  } else {
   printf( "Use: %s number\n", argv[0] );
   error_code = -1;
  } /* if */
  return error_code;
} /* main */
```

Note

After creating the observer thread we check whether the number is prime and, on returning to the `is_prime()`function, we set the end field to `TRUE` so that the observer ends execution. In order to wait until it has actually finished we call `pthread_join()`. This function waits until the thread whose identifier has been taken as the first argument reaches the end of its execution and the two threads are joined (or *connected*). The second argument is used to collect any possible data returned by the thread.

To complete this example we will need to code the function `is_prime()`, which will need to have the following header:

```
/* ... */
bool is_prime( t_visible *ref_data );
```

```
   /* ... */
```

The programming has been left for an exercise. To resolve it correctly, we need to remember that we need to use `ref_data->divisor` as it is, given that the `observer()` function reads it to present it to the user.

In this case, the fact that the two threads have the same memory space, or the same environment or context, does not matter. However, it is fairly common that access to shared data by more than one thread is synchronised. In other words, a mechanism is enabled to prevent the two threads from accessing the data simultaneously, especially for modifying it, but also to ensure that the data read by the threads has been correctly updated. These mechanisms are mutually exclusive and are, in fact, conventions on calls to functions before and after accessing the data.

Note

We can imagine these to be like the mechanism of a traffic light controlling access to a car park: If there is space, the traffic light will be green and if it is full, the light will be red, until a space becomes free.

14. Processes

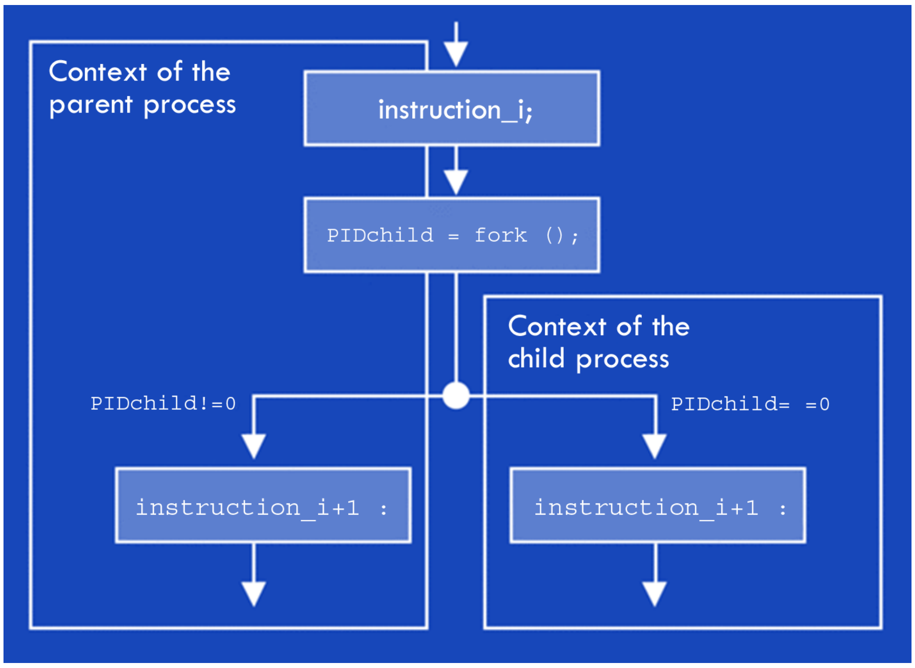
A process is a an instruction execution flow with its own environment and, therefore, with all the attributes of a program. The instruction execution flow can therefore be divided into other processes (light or not) if we deem it appropriate for reasons of efficiency.

In this case, the generation of a new process from the main process implies making a copy of the whole environment of the main one. The child process will therefore have a complete copy of the environment of the parent at the moment of separation. From then on, both the content of the variables and the indication of the next instruction may diverge. These therefore act as two distinct processes with environments which are obviously different from that of the executable code.

Given the strict separation of the environments of the processes, these are generally divided when we need to perform the same task using different data, and the task is performed by each of the child processes autonomously. There are also mechanisms to allow processes to communicate with each other: pipes, message queues, shared variables (in this case there are functions to implement mutual exclusivity) and any other type of communication which can be established between different processes.

To create a new process we just need to call `fork()`, the declaration for which is found in `unistd.h`, and this will return the identifier of the child process in the parent and 0 for the new process:

Figure 11.



Note

The fact that `fork()` returns different values for the parent and child processes allows the following instruction flows to determine whether they belong to one or the other.

The following program is a simple example of division in a dual process: the original, or parent and the copy, or child. To simplify matters we will assume that both the child and the parent perform the same task. In this example the parent waits for the child to end execution using `wait()`, this requires the inclusion of the `sys/types.h` and `sys/wait.h`:

```
/* File: ex_fork.c                                     */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* ... */

int main( void )

{
  pid_t              process;
  int                status;
  printf( "Parent process (%li) initialised.\n", getpid() );
  process = fork();
  if( process == 0 ) {
   printf( "Child process (%li) initialised.\n", getpid() );
   task( "child" );
```

```
     printf( "End of child process.\n" );
    } else {
     task( "parent" );
     wait( &status ); /* Waits for child to end.    */
    printf( "End of parent process.\n" );
    } /* if */
    return 0;
  } /* main */
```

To prove that these two processes are executed in parallel, the parent and child tasks should be different or they should use different input data.

To illustrate this situation we will look at the programming of a function in which the task employs repeated waits. In order to observe that the execution of the two processes may not always be interleaved in the same way, both the number of repetitions and the waiting time will be based on a random number provided by the function `random()`, this needs a "seed" value which is calculated using `srandom()` with an argument which will vary with each execution and the type of process (parent or child):

```
  /* ... */
  void task( char *name )
  {
    unsigned int counter;

    srandom( getpid() % ( name[0] * name[2]) );
    counter = random() % 11 + 1;
    while( counter > 0 ) {
     printf( "... step %i del %s\n", counter, name );
     sleep( random() % 7 + 1 );
     counter = counter -1;
    } /* while */
  } /* task */
  /* ... */
```

In the above example the parent waits for a single child and performs the same task. This is obviously not common. It is much more usual that the parent process generates a child process for each set of data to be processed. In this case, the main program is made slightly more complicated and we need to make the selection based on the value returned by `fork()` if the instructions belong to the parent or one of the children.

To illustrate the coding of these kinds of programs we will look at a program which uses an unknown number of natural numbers as arguments and checks to see if they are prime numbers or not. In this case the main program will create a child process for each natural number to be treated:

```
/* ... */
int main( int argc, char *argv[] )
{
  int counter;
  unsigned long int  number, divisor;
  pid_t              process;
  int                status;

  if( argc > 1 ) {
   process = getpid();
   printf( "Process %li initiated.\n", process );
   counter = 1;
   while( process != 0 && counter < argc ) {
    /* Creation of child processes: */
    number = atol( argv[ counter ] );
    counter = counter + 1;
    process = fork();
    if( process == 0 ) {
     printf( "Process %li for %lu\n",
      getpid(),
      number
     ); /* printf */
     divisor = is_prime( number );
     if( divisor > 1 ) {
      printf( "%lu is not prime.\n", number );
      printf( "Its first divisor is %lu\n", divisor );
     } else {
      printf( "%lu is prime.\n", number );
     } /* if */
    } /* if */
   } /* while */
   while( process != 0 && counter > 0 ) {
    /* Wait for end of child process:*/
    wait( &status );
    counter = counter -1;
   } /* while */
   if( process !=0 ) printf( "End.\n");
  } else {
   printf( "Use: %s natural_1 ... natural_N\n", argv[0] );
  } /* if */
  return 0;
} /* main */
```

Note

The creation loop is interrupted if `process==0` to prevent the child processes from being able to create "grandchildren" with the same data as some of their "brothers".

We need to bear in mind that the program code is the same for both the parent process and the child process.

However, the wait loop at the end should only apply to the parent, meaning the process in which the variable `process` is not zero. In this case we just need to deduct one unit from the process counter for each wait performed.

To check it is working we will need to design the `is_prime()`function, but we will leave this as an exercise. In order to observe the operation of the program it is useful to enter a large prime number and a smaller or non-prime number.

14.1. Communication between processes

As we have already mentioned, processes can communicate with each other (either if they belong to the same program or not) using the mechanisms of pipes, message queues and shared variables, among others. These mechanisms can also be applied in communications between different programs of the same application and even different applications. However, these will always be low-level communications and will require mutual exclusion when accessing data to prevent conflicts.

As an example, we will look at a program which breaks down any natural number into the sum of the powers of the prime divisors. To do this it uses a process to calculate the prime divisors and another, the parent, to show them once they have been calculated. Each factor of the sum is a piece of data of the type:

```
typedef struct factor_s {
  unsigned long int divisor;
  unsigned long int power;
} factor_t;
```

Communications between the processes are performed using a pipe.

Note

Remember that in the previous unit we defined a *pipe*. A pipe consists of two byte stream files, one for input and one for output, which are used by two processes to communicate.

As we can see from the following code, the function used to open a pipe is called `pipe()` and takes the argument of the address of a two-integer vector which will hold the descriptors of the *stream* files it has opened: The output at position 0 and the input at position 1. After the `fork()`, both processes have a copy of the descriptors and, therefore, can access the same files both for the input and the output of data. In this case, the child process will close the input file and the parent will close the output file, given that the pipe only allows the processes to communicate in a single direction: from child to parent. (If we would like to communicate in both directions we would need to establish a data access protocol to avoid conflicts):

```
/* ... */
int main( int argc, char *argv[] )
{
  unsigned long int number;
  pid_t             process;
  int               status;
  int              desc_pipe[2];

  if( argc == 2 ) {
   printf( "Prime divisors.\n" );
   number = atol( argv[ 1 ] );
   if( pipe( desc_pipe ) != -1 ) {
    process = fork();
    if( proceso == 0 ) { /* Child process:               */
     close( desc_pipe[0] );
     divisors_of( number, desc_pipe[1] );
     close( desc_pipe[1] );
    } else { /* Main or parent process:                  */
    close( desc_pipe[1] );
    show_divisors( desc_pipe[0] );
    wait( &status );
    close( desc_pipe[0] );
    printf( "End.\n" );
   } /* if */
  } else {
   printf( "I cannot create the pipe!\n" );
  } /* if */
 } else {
  printf( "Use: %s natural_number\n", argv[0] );
 } /* if */
 return 0;
} /* main */
```

Including everything, the code of the `show_divisors()` function in the parent process could be as the one shown below. It uses the reading function `read()`, which tries to read a certain number of bytes from the file whose descriptor is sent to it as the first argument. It returns the number of bytes effectively read and their content and deposits this at the indicated memory address.

```
/* ... */
void show_divisors( int desc_input )
{
  size_t nbytes;
  factor_t factor;

  do {
   nbytes = read( desc_input,
    (void *)&factor,
    sizeof( factor_t )
   ); /* read */
   if( nbytes > 0 ) {
    printf( "%lu ^ %lu\n",
     factor.divisor,
     factor.power
    ); /* printf */
  } while( nbytes > 0 );
} /* show_divisors */
/* ... */
```

To complete the example we will look at a possible programming of the function `divisors_of()` in the child process. This function uses `write()` to deposit the recently calculated factors in the output file of the pipe:

```
  /* ... */
  void divisors_of(
    unsigned long int number,
    int           desc_output )
  {
    factor_t f;

    f.divisor = 2;
    while( number > 1 ) {
     f.power = 0;
     while( number % f.divisor == 0 ) {
      f.power = f.power + 1;
      number = number / f.divisor;
     } /* while */
     if( f.power > 0 ) {
      write( desc_output, (void *)&f, sizeof( factor_t ) );
     } /* if */
     f.divisor = f.divisor + 1;
    } /* while */
  } /* divisors_of */
  /* ... */
```

In this example we have looked at one of the possible ways processes can communicate with each other. In general, each communication mechanism will have preferred uses.

Note

Pipes are suitable for passing relatively large amounts of data between processes while message queues are better if the process communicates infrequently or in an irregular way.

We always need to remember that distributing the tasks of a program over several processes will involve an increase in its complexity due to the need to insert communications mechanisms among them. It is therefore important to evaluate the benefits which such a division may bring to the development of the program.

Summary

The algorithms which are used to process the information may be more or less complex depending on how they are represented. As a consequence, the efficiency of the programming is directly related to the data structures it uses.

For this reason we have introduced dynamic data structures that allow us to make better use of the memory and to change the relationships they have as part of the processing of the information.

Dynamic data structures are those in which the number of pieces of data can vary during the execution of a program and whose relationships may also change. This is done through the creation and destruction of dynamic variables and in the mechanisms used to access them. Basically, accessing these variables should be done using pointers given that dynamic variables do not have names with which they can be identified.

We have also looked at common examples of dynamic data structures such as character strings and node lists. For the second of these we reviewed the possible programming of the management funcions for the nodes in a list and we took a look at how they can be treated in a special way, in that they can be used to represent queues.

Given that many of these common dynamic data structure functions are used often, we can group them into object file archives: function libraries. In this way we can use the same functions in diverse programs without having to worry about programming them. Even so, we will need to include the header files in order to tell the compiler how to invoke these functions. We looked at the mechanism for creating function libraries and also introduced the *make* utility which is used for generating executables resulting from the compilation of several units of the same program and the library archives required.

We have seen how the relationships between the different abstract data types of a program can facilitate modular programming. In fact, these types are classified according to their level of abstraction, or, depending on how you look at it, by their dependency on other data types. It is therefore abstract data types which are defined in terms of primitive data types which have the lowest level of abstraction.

As such, it will be the main program which operates with the data types of the highest level of abstraction. The rest of the program modules will be those providing the main program with the functions it needs to perform these operations.

We can therefore use top-down algorithm design, based on the hierarchy established between the different data types used, as a technique to produce an efficient modular program.

In practice, each abstract data type should be accompanied by the functions for basic operations such as creation, data access, copying, duplication and destruction for the corresponding dynamic variables. What is more, they should be contained in an independent compilation unit together with the correct header file.

In the last chapter we looked at the organisation of code, not just in terms of the information to be processed but also in relation to how this should be done. It therefore makes sense to take maximum advantage of the facilities provided by the C programming language for using the service routines of the operating system.

When information needs to be processed by another program, we can execute it from the instruction execution flow of the one under execution. In this case however, there will be minimal communication between the called program and the caller. As a consequence, it should be the called program which handles most of the information and which produces the result.

We have also looked at the possibility of dividing the instruction execution flow into several different flows which are executed concurrently. In this way it is possible to specialise each flow for a certain aspect of the treatment of the information and, in other cases, to perform the same treatment on different parts of the information.

Instruction execution flows can be divided into threads (*threads*) or processes. Threads are also known as light processes as they share the same context of execution (environment). We would determine the best type of division based on the type of treatment of the information. We can use the level of sharing of the information as a rule: if it is high, it will be better to use a thread, if it is low, a process (there are several communications methods depending on the level of relationship they have).

Part of the content of this unit will re-examine how both C++ and Java facilitate programming using abstract data types, modular design and the distribution of the execution over several instruction flows.

Self-evaluation

1. Write a search engine for words in files similar to the last exercise of the previous unit. The program should ask for the name of the file and the word to be searched. In this case, the main function should be as follows:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
typedef char *word_t;
word_t next_word(
  char         *phrase,
  unsigned int start
) { /* ... */ }
int main( void )
{
  FILE         *input;
  char         name[BUFSIZ];
  word_t    word, word2;
  unsigned int numlin, pos;
  printf( "Search words.\n" );
  printf( "File: ");
  gets( name );
  input = fopen( name, "rt" );
  if( input != NULL ) {
   printf( "Word: ");
   gets( name );
   word = next_word( name, 0 );
   printf( "Searching %s in file...\n", word );
   numlin = 1;
   while( fgets( name, BUFSIZ-1, input ) != NULL ) {
    numlin = numlin + 1;
    pos = 0;
    word2 = next_word( name, pos );
    while( word2 != NULL ) {
     if( !strcmp( word, word2 ) ) {
      printf( "... line %lu\n", numlin );
     } /* if */
```

```
      pos = pos + strlen( word2 );
      free( word2 );
      word2 = next_word( name, pos );
     } /* while */
    } /* while */
    free( word );
    fclose( input );
    printf( "End.\n" );
   } else {
    printf( "Can not open %s!\n", name );
   } /* if */
   return 0;
 } /* main */
```

We then need to program the function `next_word()`.

2. Using the functions provided in section 4.2, write a function to delete the nth element from a list of integers. The main program should be as follows:

```
int main( void )
{
  list_t       list;
  char     option;
  int      data;
  unsigned int n;

  printf( "Integer list manager.\n" );
  list = NULL;
  do {
   printf(
    "[I]nsert, [D]elete, [S]how or [E]xit? "
   ); /* printf */
   do option = getchar(); while( isspace(option) );
   option = toupper( option );
   switch( option ) {
    case 'I':
     printf( "Data =? ");
     scanf( "%i", &data );
     printf( "Position =? " );
     scanf( "%u", &n );
     if( !insert_nth_list( &list, n, data ) ) {
      printf( "Was not inserted.\n" );
     } /* if */
     break;
    case 'E':
     printf( "Position =? " );
     scanf( "%u", &n );
     if( delete_nth_list( &list, n, &data ) ) {
      printf( "Data = %i\n", data );
      } else {
      printf( "Was not deleted.\n" );
      } /* if */
      break;
     case 'S':
      show_list( list );
      break;
    } /* switch */
  } while( option != 'E' );
  while( list != NULL ) {
   delete_nth_list( &list, 0, &data );
  } /* while */
  printf( "End.\n" );
  return 0;
} /* main */
```

We will also need to program the function `show_list()` to be able to see its contents.

3. Write a program which allows us to insert and delete elements from an integer queue. The functions which should be used can be found in the section referring to queues in section 4.2. We therefore only need to develop the main function for this program, you could use the one from the previous exercise as the basis.

4. Program the ordering by selection algorithm seen in section 5 for classifying a text file in which each line has the following format:

National identity number mark

The elements will be of the same data type as that seen in the example. The main program will be:

```
int main( void )
{
  FILE          *input;
  char          name[BUFSIZ];
  list_t    pending, ordered;
  ref_node_t  refnode;
  element_t   element;

  printf( "Order list of names.\n" );
  printf( "File =? " ); gets( name );
  input = fopen( name, "rt" );
  if( input != NULL ) {
   initialise_list( &pendings );
   while( fgets( name, BUFSIZ-1, input ) != NULL ) {
    element = read_element( name );
    put_at_end_of_list( &pendings, element );
   } /* if */
   initialise_list( &ordered );
   while( ! list_is_empty( pending ) ) {
    elemento= extract_minimum_from_list( &pending );
    put_at_end_of_list( &ordered, element );
    } /* while */
    printf( "List ordered by national identity number:\n" );
    start_of_list( &ordered );
    while( !end_of_list( ordered ) ) {
     refnode = ref_node_of_list( ordered );
     element = element_in_ref_node(ordered, refnode);
     show_element( element );
     advance_position_in_list( &ordered );
    } /* while */
    printf( "End.\n" );
   } else {
    printf( "Can not open %s!\n", name );
   } /* if */
   return 0;
  } /* main */
```

We will also need to program the following functions:

- `element_t create_element( unsigned int national identity number, float marks );`
- `element_t read_element( char *phrase );`
- `void show_element( element_t element );`

Note

In this case, the elements of the list are not destroyed before the end of the execution of the program because it is simpler and we also then know that all the memory will have been released. However, it is still not good programming practice and we therefore suggest you do another exercise and incorporate a function to delete the dynamic variables for each element before the execution of the program has finished.

5. Write the above program in three different compilation units: One for the main program, which can also be divided into more manageable functions, one for the elements and another for the lists, which can be transformed into a library.

6. Write a program which accepts a fiscal identification number as an argument and validates it. You can use exercise number 7 from the previous unit as a reference.

7. Transform the utility for searching words in text files from the first exercise so that it uses both the search word and the name of the text file in which the search is to be performed as arguments in the command line.

8. Create a command which shows the content of the directory as if it was a `ls -als | more` . In order to do this, we will need to write a program that executes this mandate and returns the corresponding error code.

9. Program an "alarm clock" which displays a message after a certain period or at a certain time. Use the example program we looked at in the section on permanent processes as a reference.

The first argument of the program should be the hour and minutes when the message is displayed, the message will be the second argument. If the hours and minutes are preceded by the +" sign, it will be treated in the same way as in the example, as the time period which should pass before the message is shown.

We should remember that the reading of the first value of the first argument can be done in the same way as in the "alarm" program, given that the +" sign is interpreted as a sign indicator for the same number. This means we specifically need to read `argv[1][0]` to know if the user has inserted the sign or not.

To know the current time we need to use the standard library functions for time, which are declared in `time.h`, the use of which is shown in the following program:

```
/* File: hourmin.c                              */
#include <stdio.h>
#include <time.h>
int main( void )
{
  time_ttime;
  struct tm *time_desc;
  time( &time );
  time_desc = localtime( &time );
  printf( "It is %2d and %2d minutes.\n",
   time_desc->tm_hour,
   time_desc->tm_min
  ); /* printf */
  return 0;
} /* main */
```

10. Try out the prime number detection programs using threads and processes. To do this we need to define the function `is_prime()` correctly. The following program is a demonstration of this function and takes advantage of the fact that no integer divisor will be larger than the square root of the number (it will be approximated by the closest power of 2):

```
/* File: is_prime.c                                  */
#include <stdio.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
  unsigned long int number, maximum, divisor;
  bool              prime;

  if( argc == 2 ) {
   number = atol( argv[ 1 ] );
   prime = number < 4; /* 0 ... 3, considered prime.  */
   if( !prime ) {
    divisor = 2;
    prime = number % divisor != 0;
    if( prime ) {
     maximum = number / 2;
     while( maximum*maximum > number ) maximum = maximum/2;
     maximum = maximum * 2;
     divisor = 1;
     while( prime && (divisor < maximum) ) {
      divisor = divisor +2;
      prime = number % divisor != 0;
     } /* while */
    } /* if */
   } /* if */
   printf( "... %s prime.\n", prime? "is" : "is not" );
  } else {
   printf( "Use: %s natural_number\n", argv[0] );
  } /* if */
  return 0;
} /* main */
```

Answer key

1. As we already have the main program, it is sufficient to show the function `next_word`:

```
word_t next_word(
   char         *phrase,
   unsigned int start)
{
   unsigned int end, length;
   word_t    word;
   while( phrase[start]!='\0' && !isalnum(phrase[start])
    start = start + 1;
   } /* while */
   end = start;
   while( phrase[end]!='\0' && isalnum( phrase[end] ) )
    end = end + 1;
   } /* while */
   length = end - start;
   if( length > 0 ) {
    word = (word_t)malloc((length+1)*sizeof(char));
    if( word != NULL ) {
     strncpy( word, &(phrase[start]), length );
     word[length] = '\0';
    } /* if */
   } else {
    word = NULL;
   } /* if */
   return word;
} /* next_word */
```

2.

```
bool delete_nth_list(
  list_t *listref, /* Pointer to reference 1st node.*/
  unsigned int n,    /* Position of deletion.              */
  int      *dataref)      /* Reference of deleted data.      *
{ /* Returns FALSE if it can not.
  node_t *p, *q, *t;
  bool retval;
  nth_pq_node( *listref, n, &p, &q );
  if( q!= NULL ) {
   *dataref = destroy_node( listaref, p, q );
   retval = TRUE;
  } else {
   retval = FALSE;
  } /* if */
  return retval;
} /* delete_nth_list */

void show_list( list_t list )
{
  node_t *q;
  if( list != NULL ) {
   q = list;
   printf( "List = " );
   while( q != NULL ) {
    printf( "%i ", q->data );
    q = q->next;
   } /* while */
   printf( "\n" );
  } else {
   printf( "List empty.\n" );
  } /* if */
} /* show_list */
```

3.

```
int main( void )
{
  queue_t   queue;
  char      option;
  int       data;

  printf( "Integer queue manager.\n" );
  queue.first = NULL; queue.last = NULL;
  do {
   printf(
    "[E]nqueue, [D]equeue, [S]how or [E]xit? "
   ); /* printf */
   do option = getchar(); while( isspace(option) );
   option = toupper( option );
   switch( option ) {
    case 'E':
     printf( "Data =? ");
     scanf( "%i", &data );
     if( !enqueue( &queue, data ) ) {
      printf( "Was not inserted.\n" );
     } /* if */
     break;
    case 'D':
     if( dequeue( &queue, &data ) ) {
      printf( "Data = %i\n", data );
     } else {
      printf( "Was not deleted.\n" );
     } /* if */
     break;
    case 'S':
     show_list( queue.first );
     break;
   } /* switch */
  } while( option != 'E' );
  if( dequeue( &queue, &data ) ) { ; }
  printf( "End.\n" );
  return 0;
} /* main */
```

4.

```
element_t create_element( unsigned int national identity number, float score )
{
  element_t   element;
  element = (element_t)malloc( sizeof( data_t ) );
  if( element != NULL ) {
   element->national identity number = national identity number;
   element->score = score;
  } /* if */
  return element;
} /* create_element */

element_t read_element( char *phrase )
{
  unsigned int national identity number;
  double       score;
  int          read_ok;
  element_t   element;
  read_ok = sscanf( phrase, "%u%lf", &national identity number, &score );
  if( read_ok == 2 ) {
   element = create_element( national identity number, score );
  } else {
   element = NULL;
  } /* if */
  return element;
} /* read_element */
void show_element( element_t element )
{
  printf( "%10u %.2f\n", element->national identity number, element->score );
} /* show_element */
```

5. See section 5

6.

```
char letter_of( unsigned int national identity number )
{
  char code[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;
  return code[ national identity number % 23 ];
} /* letter_of */

int main( int argc, char *argv[] )
{
  unsigned int national identity number;
  char         letter;
  int error_code;

  if( argc == 2 ) {
   sscanf( argv[1], "%u", &national identity number );
   letter = argv[1][ strlen(argv[1])-1 ];
   letter = toupper( letter );
   if( letter == letter_of( national identity number ) ) {
    printf( "fiscal identity number valid.\n" );
    error_code = 0;
   } else {
    printf( "Letter %c not valid!\n", letter );
    error_code = -1;
   } /* if */
  } else {
   printf( "Use: %s national identity number-letter\n", argv[
   error_code = 1;
  } /* if */
  return error_code;
} /* main */
```

7.

```
int main( int argc, char *argv[] )
{
  FILE         *input;
  char         name[BUFSIZ];
  word_t    word, word2;
  unsigned int numlin, pos;
  int error_code;
  if( argc == 3 ) {
   word = next_word( argv[1], 0 );
   if( word != NULL ) {
    input = fopen( argv[2], "rt" );
    if( input != NULL ) {
     /* (See: Statement in exercise 1.)
    } else {
     printf( "Can not open %s!\n", argv[2] );
     error_code = -1;
    } /* if */
   } else {
    printf( "Word %s invalid!\n", argv[1] );
    error_code = -1;
   } /* if */
  } else {
   printf( "Use: %s word file\n", argv[0] );
   error_code = 0;
  } /* if */
  return error_code;
} /* main */
```

8.

```
int main( int argc, char *argv[] )
{
  int error_code;

  error_code = system( "ls -als | more" );
  return error_code;
} /* main */
```

```
int main( int argc, char *argv[] )
{
  unsigned int hours;
  unsigned int minutes;
  unsigned int seconds;
  char          *message, *separator;
  time_t       time;
  struct tm *time_desc;
  if( argc == 3 ) {
   separator = strchr( argv[1], ':' );
   if( separator != NULL ) {
    hours = atoi( argv[1] ) % 24;
    minutes = atoi( separator+1 ) % 60;
   } else {
    hours = 0,
    minutes = atoi( argv[1] ) % 60;
   } /* if */
     if( argv[1][0]!='+' ) {
      time( &time );
      time_desc = localtime( &time );
      if( minutes < time_desc->tm_min ) {
       minutes = minutes + 60;
       hours = hours - 1;
      } /* if */
      if( hours < time_desc->tm_hour ) {
       hours = hours +24;
      } /* if */
      minutes = minutes - time_desc->tm_min;
      hours = hours - time_desc->tm_hour;
     } /* if */
     seconds = (hours*60 + minutes) * 60;
     message = argv[2];
     if( daemon( FALSE, TRUE ) ) {
      printf( "The alarm can not be installed :-(\n" );
    } else {
     printf( "Alarm in %i hours and %i minutes.\n",
     hours, minutes
    ); /* printf */
    printf( "Do $ kill %li to stop it.\n",
     getpid()
    ); /* printf */
   } /* if */
   sleep( seconds );
   printf( "%s\007\n", message );
   printf( "Alarm stopped.\n" );
  } else {
   printf( "Use: %s [+]HH:MM \"message\"\n", argv[0] );
   printf( "(with + is with respect to the current time)\n" );
   printf( "(without + is the time of the day)\n" );
  } /* if */
  return 0;
 } /* main */
```

10. This means repeating the given code within a function which has the correct header for each case.

Object oriented programming in C++

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Josep Anton Pérez López
Lluís Ribas i Xirgo

PID_00148430

Index

Introduction

Up until now we have mainly looked at how to tackle a problem using the modular programming and top-down design paradigms for algorithms. We can use these to overcome complex problems by breaking them down into simpler problems, progressively reducing their level of abstraction until we get a more manageable level of detail. The problem will be reduced to data structures and functions or procedures.

In order to work more efficiently, good programming practice tells us to group sets of routines and interrelated structures into compilation units which are then linked to the main archive. This achieves the following:

- The rapid localisation of the source code which performs a specific task and limiting the impact of the modifications on certain archives.
- It improves the legibility and understandability of the source code as the parts are separated.

However, organising the project documents like this will only provide a certain separation between the different archives and will not reflect the close relationship which often exists between the data and the functions.

In reality we will often want to implement entities in such a way that they have a few general properties: awareness of the inputs they need, a general idea of their functionality and the outputs they generate. In general, the specific details of the implementation are not important: there will clearly be dozens of ways to do it.

We can use a television as an example. Its properties could be the brand, the model, its measurements and number of channels and the actions to be taken could be turning it on or off, changing the channel, tuning to a new channel etc. When we are using a television we see it as a closed box with a set of properties and connections. We have no interest in its internal workings, we just want it to do something when we press the appropriate button. What is more, it can be used anywhere and the functions will always be the same. If it breaks down, it can be replaced by a new one and the basic characteristics (having a brand, turning on and off, changing channels etc) will remain the same despite the fact that the television is more modern. The television is treated as an individual object and not as a set of component parts.

When we apply this principle to programming it is called **encapsulation**. Encapsulation consists of implementing an element (the details of which we will look at later) which will act as a "black box" where we have inputs, a general idea of its operation and some outputs. This provides the following:

- The re-use of code. If we already have a "black box" whose characteristics match the needs defined, it can be incorporated without interfering with the rest of the project.

- The maintenance of code. We can make changes without affecting the project as a whole and still complying with the specifications of said "black box".

We will call each of these elements *objects* (in reference to the objects they represent in real life). When working with objects, which supposes a higher level of abstraction, we need to design an application without thinking about the sequence of instructions, but focusing on the definition of the objects involved and the relationships which have to be established between them.

In this unit we will be looking at a new language which allows us to look at things from the perspective of object oriented programming: C++.

This new language is based on the C language but has new attributes. So, firstly we should compare the two languages and their common aspects to allow us to learn it more quickly. Below we have set out the new paradigm and the tools which the new programming language provides for the implementation of objects and their relationships. Lastly we will look at how this change in philosophy affects the design of applications.

In this unit we will try to give readers a basic understanding of object oriented programming using C++ and the design of applications using this paradigm based on their knowledge of C. At the end of this unit the reader should have achieved the following objectives:

1) An understanding of the principal differences between C and C++, although without having explored object technology.

2) An understanding of the object oriented programming paradigm.

3) Knowledge of implementing classes and objects in C++.

4) An understanding of the main properties of objects: inheritance, homonymity and polymorphism.

5) To be able to design a simple application in C++ applying the principles of object oriented design.

1. From C to C++

1.1. Our first C++ program

Choosing the C++ programming environment for the implementation of the new paradigm of object oriented programming provides many advantages due to the many similarities with C. However it could become a limiting factor if the programmer does not explore the new attributes of this language and the fact that it has some very interesting improvements.

Traditionally, in the world of programming the first contact with a new language takes the form of displaying the classic message "Hello world" and we will make no exceptions in this case.

You should therefore write the following text in your text editor and save it with the name *example01.cpp*:

Note

The extension ".cpp" tells the compiler that the source code type is C++.

```
#include <iostream>
int main()
{
  cout << "hello world \n" ;
  return 0;
}
```

Comparing this program with our first program in C we can see that the structure is similar. In fact, as we have already mentioned, C++ can be seen as an evolution of C for the implementation of object oriented programming and therefore a large percentage of it is compatible.

The only observable difference is to be found in that the output is managed by an object called `cout`. We will look at the nature of objects and classes in more depth later on, but for now we can get an idea of how they work by considering a class as a new data type which includes attributes and associated functions and the object as being a variable of said data type.

The definition of the object `cout` is contained in the <iostream> library which is included in the first line of the source code. We should also notice the way that the text "Hello world" is addressed to the object using the << symbol, the `cout` function generates the output of this message on the screen.

As the handling of the input/output functions is one of the main new areas in C++, we will begin by looking at the differences between one language and the other.

1.2. Data input and output

Although the input/output operations are not actually defined within the programming languages of C or C++, they are obviously required for programs to be able to work. The operations allowing communication between users and the programs are to be found in libraries provided by the compiler. In this way we can translate a source code written for a Sun environment to our home PC, making it independent of the platform used. At least in theory.

As mentioned in previous units, input/output functionality in C is provided through function libraries, the most important of which is <stdio.h> or <cstdio> (standard input/output). The functions (`printf, scanf, fprint, fscanf` etc.) still work in C++ although we do not recommend using them as they do not take advantage of the advances in the new programming environment.

Note

Both ways of expressing the name of the library <xxxx.h> or <cxxxx> are correct although the second is considered to be the standard way of including C libraries within the C++ language and the only one recommended for use in new applications.

C++, as with C, understands the communication of data between the program and the screen as a data stream: the program progressively sends data and the screen receives it and displays it. It also understands communications between the keyboard (or other input devices) and the program.

In order to manage these data streams, C++ includes the iostream class which creates and initialises four objects:

- `cin`. Manages data input streams.
- `cout`. Manages data output streams.
- `cerr`. Manages the output to the default error device: the screen.
- `clog`. Manages error messages.

Some simple examples of their use are shown below.

```
#include <iostream>
int main()
{
  int number;
  cout << "Write a number: ";
```

```
    cin >> number;
  }
```

In this block of code we can observe the following:

- The declaration of an integer variable we want to work with.
- The text "Write a number" (which we consider as a literal data stream) which we wish to send to our output device.

To achieve our goal we direct the text to the object `cout` using the >> operator. The result is that the message is shown on the screen.

- A variable in which we wish to store the input from the keyboard. Once again we can achieve this by directing the input stream received from the keyboard (represented/managed by the object `cin`) to the variable.

The first surprise for C programmers who are used to `printf` and `scanf` is that the format of the data we want to print or receive is not indicated in the instruction. This is in fact one of the main advantages of C++: the compiler recognises the data type of the variables and handles the data stream correspondingly. Simplifying this idea slightly, we could say that the objects `cin` and `cout` adapt themselves to the data type. This attribute allows us to adapt the objects `cin` and `cout` for the handling of new data types (for example, `structs`), which would be unthinkable with the earlier system.

If we want to show or collect several variables we simply chain the data streams:

```
  #include <iostream>
  int main()
  {
    int i, j, k;
    cout << "Enter three numbers";
    cin >> i >> j >> k;
    cout << "The numbers are: "
    cout << i << ", " << j << " and " << k;
  }
```

In the last line we can see how, firstly, the data stream is sent to `cout` , this corresponds to the text "The numbers are: "; followed by the data for the variable i; followed by the literal text " , ", and so on till the end.

When entering data using the keyboard, `cin` will read the characters until a line feed character is entered (return or "`\n`"). It will then extract the data stream of characters until it finds the first space and will store the result in the variable i. The result of this operation will also be a data stream (without the first number that has already been extracted) that will receive the same treatment: characters of the data stream will be extracted up to the next separator to be sent to the next variable. This process is repeated for all three variables.

The read line could therefore have been written in the following way and would be equivalent, but less clear:

```
( ( ( cin >> i ) >> j ) >> k )
```

If we want to show the variable in a certain format we must use an object manipulator indicating the desired format. We can see how this works in the following example:

```
#include <iostream>
#include <iomanip>
// Must be included for the definition of the
// cout object manipulators with parameters

int main()
{
  int i = 5;
  float j = 4.1234;

  cout << setw(4) << i << endl;
  //shows i with a length of 4 char.
  cout << setprecision(3) << j << endl;
  // shows j to three decimal places
}
```

There are many other formatting possibilities but they are not the object of this course. This information is available on the compiler's help page.

1.3. Using C++ like C

As we have already mentioned, C++ evolved from C, meaning that it is fairly easy for C programmers to adapt to the new environment. However, as well as introducing object oriented programming, C++ also includes other improvements over classical programming which can be useful to learn and which allow us to fully take advantage of the object oriented programming paradigm.

We will now look more in depth at the various aspects of the language.

1.4. Basic instructions

In this respect, C++ has remained faithful to C. The instructions keep their basic appearance (ending in a semi-colon, blocks of code between bracers etc.) and the basic flow control instructions, both selection and iterative, keep their syntax (`if, switch, for, while, do ... while`). These attributes make leaning the new language a fairly quick process.

We can include the input/output functions within the basic instructions. However, these do present significant changes in C++ which we looked at in the last section.

We also need to underline the fact that it includes a new way of adding comments within the source code to make reading and maintaining it easier. It still keeps the comment system where we can include text between the characters /* (start of comment) and */ (end of comment), but a new way of doing it has also been added: the sequence // allows us to add a comment up to the end of the line.

Example

```
/*
This text has been included using the classic method used in C.
It can contain as many lines as we want.
*/

//
// This text uses the new comment format
// up to the end of line incorporated in C++
//
```

1.5. Data types

The fundamental data types in C (`char, int, long int, float` and `double`) are still in C++ with the addition of `bool` (boolean or logic type), which can acquire two possible values: false or true, these have now been defined in the language.

```
// ...
{
  int i = 0, num;
  bool continue;

  continue = true;
  do
  {
   i++;
   cout <<  "To end this loop that has run";
  cout << i << "times, press 0 ";
   cin >> num;
   if (num == 0) continue = false;
  } while (continue);
}
```

Although we often saw logic or boolean variables in C using integer values (0 is false and any other value is true), the new implementation simplifies this and helps to prevent errors. The new data type also only occupies one byte of memory instead of the two bytes used when simulating it with the type `int`.

We also need to point out other new aspects relating to structured type (`struct`, `enum` or `union`). In C++ these are now used to describe complete data types so avoiding the need for the use of the instruction `typedef` to define new data types.

In the following example we can see that the definition part of the new type does not change:

```
struct date {
  int day;
  int month;
  int year;
};
enum daysWeek {MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                FRIDAY, SATURDAY, SUNDAY};
```

The declaration of a variable of this type is simplified given that we do not need to repeat the terms `struct`, `enum` or `union`, or define new types using the instruction `typedef`:

```
date birthday;
daysWeek holiday;
```

Referencing data does not change either.

```
// ...
birthday.day = 2;
birthday.month = 6;
birthday.year = 2001;
holiday = MONDAY;
```

In the case of declaring variables of the type `enum` , two functions are fulfilled:

- Declaring `daysWeek` as a new type.
- Making `MONDAY` correspond to the constant 0, `TUESDAY` to the constant 1 and so on.

Therefore each enumerated constant will be an integer value. If none are specified, the first value taken will be 0 and the successive constants will increase by one unit. However, C++ also allows us to change this and we can assign a specific value to each constant:

```
enum behaviour {HORRIBLE = 0, BAD, FAIR = 100,
  GOOD = 200, VERY_GOOD, EXCELLENT};
```

In this way, `HORRIBLE` will take the value 0, `BAD` the value 1, `FAIR` the value 100, `GOOD` the value 200, `VERY_GOOD` the value 201 and `EXCELLENT` the value 202.

Another aspect to bear in mind is the recommendation for the new version on typecasting. For greater legibility C++ recommends using:

```
int i = 0;
long v = (long) i; // typecasting in C
long v = long (i); // typecasting C++
```

1.6. The declaration of variables and constants

The declaration of variables in C++ still has the same format as in C, but we also insert a new element which provides a safer way of working. The use of the specifier `const` for the definition of constants.

In programming, we use a constant when we know for sure that a certain variable will not change during the execution of the application.

```
const float PI = 3.14159;
```

Once the constant has been defined, it can not be assigned another value and will therefore always appear on the right hand side of expressions. However, a constant must always be initialised:

```
const float radius;// ERROR!!!!!!!!
```

The use of constants is not new in C. The classic way of defining them using the preprocessor instruction `#define`.

```
#define PI 3.14159
```

In this case the actual action will be to replace each appearance of the text `PI` by its value in the text preprocessing phase. Therefore, when it sees this text the compiler only sees the number 3.14159 and not `PI`.

In spite of that, while the second of these cases corresponds to special treatment during the process before compilation, the use of `const` standardises this procedure and makes it similar to a variable but with limited capabilities. It receives the same treatment as variables with respect to the scope of action (only in the working file, unless indicated otherwise using the reserved word `extern`) and it has an assigned type, this means that all the types can be checked during the compilation phase making the source code more robust.

1.7. The management of dynamic variables

Direct memory management is one of the most powerful tools available in C++, but also one of the most dangerous: Accidentally accessing memory locations which do not correspond to the required data can have unpredictable results, in the worst case they may be disastrous.

In the previous chapters we have seen how operations using memory addresses in C are based on the use of pointers (`*point`) to access a variable from its memory address. They use the indirection or dereference operator (`*`), and their attributes are:

- `pointer` contains a memory address.
- `*pointer` indicates the content residing at that memory address.
- To access the memory address of a variable we use the address operator (&) preceding the name of the variable.

```
// Example of the use of pointers
int i = 10;
int *point_i = &i;
     // point_i takes the address
     // of the integer type variable i
     // If it is not assigned here, it would
     // be advisable to initialise it to NULL

*point_i = 3;
     // The value 3 is assigned to the memory
     // position point_i
     //This therefore modifies the value of i.

cout << "Original Value        : " << i << endl ;
cout << "Using the pointer : "
           << *point_i << endl ;
           // The output will show:
           // Original value: 3
```

```
            // Using the pointer: 3
```

1.7.1. The operators `new` and `delete`

Pointers are mainly used in relation to dynamic variables. The two main functions defined in C for performing these operations are `malloc()` and `free()`, which are used to reserve space and release it, respectively. Both of these functions are still used in C++. However, C++ also includes two new operators which provide more robustness. These are `new` and `delete`. As they are included in the language, we do not need to add any specific library.

The `new` operator reserves the memory. Its format is `new` followed by the data type. Unlike with `malloc` we do not need to indicate the size of memory to be reserved as the compiler calculates it from the data type being used.

To free the memory we use the `delete` operator, that is also more robust than the `free()`function as it protects against an attempt to free the memory pointed to by a null pointer.

```
date * birthday = new date;
...
delete birthday;
```

If we want to create several elements they can be specified in a vector or matrix. The result is to declare the variable as a pointer to the first element of the vector.

Similarly, we can free all of the memory reserved for the vector (each of the objects created and the vector itself) using the format `delete []`.

```
date * fullMoons = new date[12];
...
delete [] fullMoons;
```

If we omit the square brackets, the result would be to only delete the first object of the vector but not the others, this creates a memory leak as we then have reserved memory which can not be accessed in the future.

1.7.2. Pointers `const`

When declaring pointers in C++ we can use the reserved word `const`. There are also other possibilities.

```
const int * ap_i;         // *ap_i remains constant
int * const ap_j;
         // The address ap_j is constant but its value is not
const int * const ap_k;
```

```
          // Both the address ap_k and its value *ap_k will remain constant
```

This means we can make the value of pointers (`*ap_i`) constant or their memory address (`ap_i`) constant, or both of them. So as not to confuse the issue we can put the reserved word in the subsequent text `const`.

With the declaration of constant pointers the programmer is telling the compiler that the value or the address of the pointer will not change. This means that any attempt to assign a value will be detected by the compiler in time. This reduces the risk of programming errors.

1.7.3. References

C++ includes a new element to make it easier to manage dynamic variables: references. A reference is an alias or synonym. When a reference is created it is initialised with the name of another variable and acts like an alternative name for it.

To create it we first include the type of the destination variable followed by the reference operator (&) and the name of the reference. For example,

```
int i;
int & ref_i = i;
```

The above expression can be read as: the variable `ref_i` is a reference to the variable `i`. References must always be initialised at the time of their declaration (as if it were a `const`).

We should note that, although the reference operator and the address operator are written in the same way (&), they refer to different operations, even though they are related. The main attribute of references is that if we ask for its address they return that of the destination variable.

```
#include <iostream>

int main()
{
  int i = 10;
  int & ref_i = i;

ref_i = 3; //The value 3 is assigned to the position

  cout << "value of i       " << i << endl;
  cout << "address of i     " << &i << endl;
  cout << "address of ref_i " << &ref_i <<endl;
}
```

In this example we can see that the addresses are identical and that the assignment to `ref_i` has the same effect as an assignment to `i`.

References also have the following attributes:

- They can not be reassigned. An attempt to reassign them will become an assignment to the synonymous variable.
- A null value may not be assigned.

References are mainly used for calling functions as we will see later.

1.8. Functions and their parameters

The use of functions, a basic element in modular programming, still has the same format: the return value type, the name of the function and the number of parameters preceded by their type. The parameters list of a function is also known as the ***function signature***.

1.8.1. Using parameters by value or by reference

As we have mentioned earlier, there are two ways of passing parameters to a function in C. By value or by variable. In the first case, the function receives a copy of the original value of the parameter while in the second it receives the address of the variable. We can therefore directly access the original variable, which can also be modified. In C, the traditional method is to pass the pointer to a variable to the function as a parameter.

Note

Here we will not use the term by reference so as not to confuse with C++.

We will now look at a function which will allow us to exchange the content of two variables:

```
#include <iostream>

void exchange(int *i, int *j);

int main()
{
  int x = 2, y = 3;
  cout << " Before. x = " << x << " y = " << y << endl;
  exchange(&x , &y);
  cout << " After. x = " << x << " y = " << y << endl;
}
void exchange(int *i, int *j)
{
  int k;
  k = *i;
  *i = *j;
  *j = k;
}
```

We can see that the use of dereferences (*) makes it harder to understand. However, in C++ we have a new concept that we mentioned before: references. This new concept consists of receiving the parameter as a reference instead of a pointer:

```
#include <iostream>

void exchange(int &i, int &j);

int main()
{
  int x = 2, y = 3;
  cout << " Before. x = " << x << " y = " << y << endl;
  exchange(x , y); //NO exchange(&x , &y);
  cout << " After. x = " << x << " y = " << y << endl;
} void exchange(int & i, int & j)
{
  int k;
  k = i;
  i = j;
  j = k;
}
```

The operation of this concept is identical to the previous one, but it is far easier to read the code when we use the reference operator (&) to collect the memory addresses of the parameters.

However, we must remember that references also have limitations (they can never take a null value and they can not be reassigned). As such, references can not be used for passing parameters when we want to pass a pointer as a parameter and it can be modified (for example, obtaining the last element of a queue data structure). Neither can they be used for those parameters which we wish to consider as optional, as there will always be the possibility that they could not be assigned to the parameter of a function which calls them, meaning they will have to take the value `null`, (which is not possible).

In these cases, passing a parameter by variable must still be done using pointers.

1.8.2. The use of `const`

In practice, when programming in C, we sometimes use a pass by variable, this is more efficient due to the fact that we do not need to make a copy of the data within the function. With large data structures (structures etc.), this operation safeguards the original values but it may take some time, we also run the risk of modifying the data due to an error.

To mitigate these risks, C++ allows us to use the specifier `const` before the parameter (as we mentioned in the section on `const`).

If, in the above exchange function, we had defined the parameters `i` and `j` as `const` (which would make no practical sense and is just for explanation), we would get compilation errors.

```
void exchange(const int & i, const int & j);
{
  int k;
  k = i;
  i = j; // Compilation error. Value i is constant.
  j = k; // Compilation error. Value j is constant.
}
```

We can therefore get extra efficiency benefits as we avoid the unwanted copying processes, but we are still protected against unwanted modifications.

1.8.3. Function overloading

C is fairly flexible in the use of function calls as it allows the use of a variable number of parameters in the call to a function, as long as these are the final parameters and, in the definition of the function, they have been assigned a value in case this parameter is not used.

C++ has incorporated a much more flexible option and this is one of the most notable new aspects with respect to C: it allows the use of different functions with the same name (*function homonymy*). This property is also known as *function overloading*.

Functions may have the same name but there must be differences in their parameter lists, either in their number or in their types.

We should note that the return value type of the function is not considered to be a difference of the function, the compiler will therefore display an error if we try to define two functions with the same name and an identical number and type of parameter but the return values are different. The reason for this is that the compiler can not tell which function you are trying to call.

We will now look at a program which squares different types of numbers:

```
#include <iostream>

int ToSquare (int);
float ToSquare (float);

int main()
{

  int numInteger = 123;
  float numReal = 12.3;
  int numIntegerSquared;
  float numRealSquared;

  cout << "Example for squaring numbers\n";
  cout << "Original numbers \n";
  cout << "Integer: " << numInteger << "\n";
  cout << "Real number: " << numReal << "\n";

  numIntegerSquared = ToSquare (numInteger);
  numRealSquared = ToSquare (numReal);

  cout << "Numbers squared \n";
  cout << "Integer:"<< numIntegerSquared << "\n";
  cout << "Real number: " << numRealSquared << "\n";

  return 0;
}

int ToSquare (int num)
{
  cout << "Squaring an integer \n";
  return ( num * num);
}
float ToSquare (float num)
{
  cout << "Squaring an integer \n";
  return ( num * num);
}
```

Overloading the function `ToSquare` has allowed us to perform what is essentially the same operation using a function with the same name. This means that we do not need to define functions with two different names:

```
 - ToSquareIntegers
 - ToSquareRealNumbers
```

In this way the compiler will identify the function you want to execute by the type of parameter and will make the correct call.

2. The object oriented programming paradigm

In the previous units we have analysed two programming paradigms (modular and top-down) which are based on the progressive organisation of data and the resolution of problems by dividing them into sequential sets of instructions. The execution of these instructions only depends on the data defined beforehand.

This allows to overcome many problems but it also has limitations:

- The sharing of data makes it difficult to modify and expand programs, as they are interrelated.
- Maintenance of large programs becomes extremely complicated as it is very hard to track the consequences of any changes made to the code.
- The re-use of code can also cause surprises as we may not be aware of all its implications.

Note

How can this present so many difficulties if people are able to perform complex actions in their daily lives? The reason is very simple: in our day-to-day lives we do not use the same criteria. Our environment is described in terms of objects: doors, computers, cars, lifts, people, buildings etc. and these objects are related in fairly simple ways: if a door is open you can go through it and if it is closed you can not. If a car has a flat tyre you can replace the wheel and get back on the road. We do not need to know everything about the mechanics of a car to be able to perform this operation! However, can we imagine a world in which replacing a wheel made the windscreen wipers stop working? It would be chaos. With the previous paradigms this is almost the case, at least we can not be completely sure it will not happen.

The object oriented programming paradigm proposes a different way of programming based on the definition of objects and the relationships between them.

Each object is represented by an **abstraction** which contains its essential information without worrying about the other attributes.

This information is made up of data (variables) and actions (functions) and, unless specifically indicated otherwise, the scope of operation is limited to this object (**Information hiding**). We can use this to limit the scope of its programming code and therefore the repercussions on the environment surrounding it. This attribute is called **encapsulation**.

The relationships between different objects can be very diverse and will normally involve one object acting on another using messages travelling between the objects.

Example

Examples of actions on objects in everyday life: making a telephone call, answering the telephone, speaking, replying etc.

One of the most important relationships between objects is the act of belonging to a more general type. In this case the more specific object shares a series of attributes (information and actions) with the generic type due to the fact that they are included in this group. The new paradigm provides a tool which allows us to re-use all these attributes in a simple way: **inheritance**.

Example

A dog is an animal, a lorry is a vehicle etc.

Lastly, one additional characteristic is the fact that they can behave differently depending on the context they are set in. This is known as **polymorphism** (one object, many forms). This property acquires its full power in that it is able to adapt this behaviour at the time of execution and not during compilation.

The object oriented programming paradigm is based on these four pillars: abstraction, encapsulation, inheritance and polymorphism.

2.1. Classes and objects

At the implementation level, a class corresponds to a new data type which contains a collection of data and functions which can be manipulated.

Example

Imagine we want to describe a video recorder.

The description might include such attributes as the brand, model, number of heads etc. or it could include its functions, such as the reproduction of video cassettes, recording, rewinding etc. This means we can look at the same unit from different perspectives.

The first perspective corresponds to a set of variables and the second to a set of functions.

The use of classes allows us to integrate both data and functions within the same entity.

The act of joining attributes and functions together in the same container allows them to be more easily related and also allows us to isolate them from the rest of the source code. In the example of the video recorder, the reproduction of a tape involves motors moving the tape over a set of heads which read the information.

The details of the operation of the unit are not that interesting to the user, we see the video recorder as a box with a slot and some buttons and we are aware that the innards contain some fairly complicated mechanics. However, we know that we only have to press the "play" button.

This concept is known as **the encapsulation of data and functions in a class**. Variables within the class are known as *member variables* or *member data* and functions are known as *member functions* or *class methods*.

But classes correspond to abstract elements or generic ideas, and the video recorder everyone has at home is not an idea but a real object with a set of attributes and specific functions. Similarly, in C++ we need to work with specific elements. We will call these elements *objects*.

2.1.1. Declaring a class

The syntax for declaring a class includes the reserved word `class` followed by the name of a class and, between bracers, the list of the member variables and the member functions

```
class Dog
{
  // list of member variables
  int age;
  int height;

  // list of member functions
  void bark();
};
```

The declaration of the class does not imply reserving any memory although it does indicate the amount of memory that each of the objects in this class will need.

Example

In the class Dog, each of the objects will occupy 8 bytes of memory: 4 bytes for the member variable `age` , which is an integer type, and 4 bytes for the member variable `height`. The definitions of the member functions, in our case `bark`, do not imply the reservation of space.

2.1.2. Implementing the member functions of a class

Up to now we have only declared two variables as members of the class Dog (`age` and `height`) and one function (`bark`). But we have not specified the implementation of the function.

The definition of a member function is done using the name of the class followed by the scope operator (`::` , the name of the member function and its parameters.

```
class Dog
{
  // list of member variables
```

```
    int age;
    int height;

    // list of member functions
    void bark();
  };

  Dog:: bark()
  {
    cout << "Woof";
  }
```

Note

Although this is the usual way, we can also implement the member functions as *inline*. To do this we insert the declaration of the method after the declaration of the method and before the semi-colon (;):

```
class Dog
{
  // list of member variables
  int age;
  int height;

  // list of member functions
  void bark()
   { cout << "Woof"; };
};
```

These types of calls are only useful when the body of the function is very small (one or two instructions).

2.1.3. The member functions `const`

In the previous chapter we looked at the usefulness of variables which are not modified during the course of the execution of the program and their declaration using the specifier `const`. We also looked at the added safety provided by `const`. We can also define member functions as being `const`.

```
void bark() const;
```

To indicate that a member function is `const`, we need to put the reserved word `const` between the closing bracket after the parameter and the semi-colon (;) at the end.

This tells the compiler that this member function cannot modify the object. Any attempt to assign a member variable or to call a non-constant function will generate an error in the compiler. This makes it a useful tool for the programmer to ensure the coherence of the lines of source code.

2.1.4. Declaring an object

In the same way that a class is similar to a new data type, an object only corresponds to the definition of an element of said type. The declaration of an object therefore uses the same model:

```
unsigned int numberFleas; // Unsigned int type variable
Dog sultan;               // object in the class Dog.
```

An object is an individual instance of a class.

2.2. Accessing objects

To access the member variables and functions of an object we use the point operator (.), we write the name of the object followed by a point and then the name of the desired variable or function.

In the previous section we defined an object called `sultan` in the class Dog, if we want to initialise the age of `sultan` to 4, or call the function `bark()`, we only need to refer to it as follows:

```
sultan.age = 4;
sultan.bark();
```

However, one of the main advantages provided by classes is that only the members we are interested in (both data and variables) are shown. For this reason, the members of a class are only visible from the functions of that class, unless indicated otherwise. In this case we say that the members are **private**.

In order to control the scope of the members of a class, we can use the following reserved words: `public`, `private` and `protected`.

Note

`protected` is a more specific case which we will look at in the section on "Inheritance".

When we declare a member (either a variable or a function) as `private` we are telling the compiler that the use of this variable is private and is restricted to within this class. However, if we declare it as `public`, it will be accesible from any place where objects of this class are used.

In the source code these reserved words are used in the form of labels before members of the same scope:

```
class Dog
{
  public:
   void bark();
  private:
   int age;
   int height;
};
```

Note

We have declared the function `bark()` to be public allowing access from outside the class, but we have kept the values for `age` and `height` hidden by declaring them private.

Now we will look at their implementation in a complete program:

```
  #include <iostream>
  class Dog
  {
    public:
     void bark() const
     { cout << "Woof"; };

    private:
     int age;
     int height;
  };

  int main()
  {
    Dog sultan;

    //Compilation error. Use of a private variable
    sultan.age = 4;
    cout << sultan.age;
  }
```

In the `main` block we have declared an object `sultan` of the class Dog. We then tried to assign the value of 4 to the member variable age. As this variable is not public the compiler will show an error indicating that it does not have access to it. We will also get a similar compilation error for the next line. One solution in this case would be to declare the member variable `age` as `public`.

2.2.1. Member data privacy

The act of declaring a member variable as being public limits the flexibility of classes, as modifying the type of the variable will also affect the various parts of the code where these values are used.

A general design recommendation is to keep the member data private and to manipulate them using public access functions for obtaining or assigning a value.

In our example we could use the functions `getAge()` and `assignAge()` as methods of accessing the data. Declaring the age to be private allows us to change the type from integer to byte, or even to replace the data by the date of birth. The modification will be limited to changing the source code in the access methods, but it will continue to be transparent outside of the class, the age can be calculated from the date of birth and the current data or we can assign an approximate date of birth using the value for age.

```
class Dog
{
  public:

   Dog(int, int);   // constructor method
   Dog(int);        //
   Dog();           //
   ~Dog();          // destructor method

   void assignAge(int);   // access methods
   int getAge();       //
   void assignHeight(int); //
   int getHeight();     //

   void bark();           // class methods
   private:
    int age;
    int height;
};

Dog:: bark()
{ cout << "Woof"; }

void Dog:: assignHeight (int nHeight)
{ height = nHeight; }

int Dog:: getHeight (int nHeight)
{ return (height); }

void Dog:: assignAge (int nAge)
{ age = nAge; }

int Dog:: getAge()
```

```
{ return (age); }
```

2.2.2. The pointer `this`

Another important aspect of member functions is that they always have access to the object itself using the pointer `this`.

In fact, the member function `getAge` can also be expressed in the following way:

```
int Dog:: getAge()
{ return (this->age); }
```

It does not seem very important if we look at it in this way. However, being able to refer to the object either using the `this` pointer or in its dereferenced form (`*this`) makes it very powerful. We will look at more advanced examples of this when we look at operator overloading.

2.3. Object constructors and destructors

In general, we always initialise a variable after defining it. This is good practice in that we are trying to prevent unpredictable results caused by using a variable without having assigned a value to it beforehand.

Classes can also be initialised. Each time we create a new object, the compiler calls on a specific class method for initialising its values, this is called the **constructor**.

The constructor always receives the name of the class, without a return value (not even `void`), and it can have initialisation parameters.

```
Dog::Dog()
{
  age = 0;
}

or

Dog::Dog(int nAge)// New age for the dog
{
  age = nAge;
}
```

If the constructor is not specifically defined in the class, the compiler uses the **predetermined constructor**, this consists of the name of the class with no parameters and the instruction block is empty. It therefore does nothing.

```
Dog::Dog()
{ }
```

This attribute sounds disconcerting but it allows us to keep the same criteria for the creation of all objects.

In our case, if we want to initialise an object of the class Dog with an initial age of 1, we will use the following definition of the constructor:

```
Dog(int newAge);
```

The call to the constructor would therefore take the following form:

```
Dog sultan(1);
```

If there are no parameters, the declaration of the constructor to be used within the class would be as follows:

```
Dog();
```

The declaration of the constructor in `main` or any other function of the body of the program would be done as follows:

```
Dog sultan();
```

However, this is a special case and we can apply an exception to the rule stating that all calls to functions must be followed by brackets even if they do not have parameters. The final result will be as follows:

```
Dog sultan;
```

The above fragment is a call to the constructor `Dog()` and matches the form of declaration presented at the beginning and that we have already looked at.

Whenever we declare a constructor method, we must also declare a **destructor** method which will clean up the object and free the memory when it is not going to be used anymore.

The destructor is always preceded by a wavy line (~), it takes the name of the class and does not have parameters or a return value.

```
~Dog();
```

If we do not define a destructor, the compiler will define a **predetermined destructor**. The definition is exactly the same but the instruction body will always be empty:

```
Dog::~Dog()
{ }
```

If these new definitions are incorporated into the program, the final result will be as follows:

```
#include <iostream>

class Dog
{
  public:
   Dog(int age);    // constructor with a parameter
   Dog();           // predetermined constructor
   ~Dog();          // destructor
   void bark();

  private:
   int age;
   int height;
};

Dog:: bark()
{ cout << "Woof"; }

int main()
{
  Dog sultan(4);   // Initialisation of object
                   // with the age of 4.
sultan.bark();
}
```

2.3.1. The copy constructor

As well as providing predetermined constructor and destructor methods for classes, the compiler also provides a copy constructor.

Each time we create a copy of an object we call the copy constructor. This includes cases in which an object is passed as a value parameter to a function or the object is returned by a function. The purpose of the copy constructor is to copy the member data of the object to a new one. This process is also known as *shallow copying*.

This procedure is generally correct but it may cause serious conflicts if there are pointers between the member variables. The result of a *shallow copy* would be that two pointers (that of the original object and the copy) both point to the same memory address: if one of them frees this memory the other will not know this and it will be pointing to a lost memory position, this situation can have unpredictable results.

The solution in these cases is to replace the shallow copy with a *deep copy* in which new memory positions are reserved for pointer type elements and they are assigned the content of the variables pointed to by the original pointers.

This constructor is declared as follows:

```
Dog :: Dog (const Dog & adog);
```

In this declaration we can see the advantage of passing the parameter as a constant reference as the constructor does not need to alter the object.

The usefulness of the copy constructor can be seen more clearly when one of the attributes is a pointer. For this reason, in this test we will change the age type to an integer pointer. The final result will be as follows:

```
class Dog
{
  public:
   Dog();                      // predetermined constructor
   ~Dog();                     // destructor
   Dog(const Dog & rhs);       // copy constructor
   int getAge();               // access method

  private:
   int *age;
};

Dog:: getAge()
{ return (*age) }

Dog:: Dog ()                   // Constructor
{
age = new int;
* age = 3;
}

Dog:: ~Dog ()                  // Destructor
{
  delete age;
  age = NULL;
}
Dog:: Dog (const Dog & rhs)    // Copy constructor
{
  age = new int;               // New memory reserved
  *age = rhs.getAge();         // Copies the age value
                               // to the new position

}
int main()
{
  Dog sultan(4);               // Initialised with age 4.
}
```

2.3.2. Initialise values in the constructor methods

There is another way of initialising values in a constructor method which is cleaner and more efficient, consisting of inserting this initialisation between the definition of the parameters of the method and the bracer indicating the start of the block of code.

```
Dog :: Dog () :
  age (0),
  height (0)
{ }

Dog:: Dog (int nAge, int nHeight):
  age (nAge),
  height(nHeight)
{ }
```

As we can see in the above fragment, initialisation consists of a colon symbol (:) followed by the variable to be initialised and, between brackets, the value that we want to assign. This value may be either a constant or a parameter of the aforementioned constructor. If more than one variable is to be initialised, they should be separated by commas (,).

2.3.3. Static member variables and member functions

Up until now, when we have referred to classes and objects, we have situated them on different levels: classes describe abstract entities and objects describe created elements with specific values.

However, there are times when objects need to refer to an attribute or a method they have in common with the other objects in the same class.

Example

If we are creating the class of Animals, we might want to keep the total number of dogs created up to now in a variable, or write a function which allows us to count the dogs even if none have been created yet.

The solution is to precede the declaration of the member variables or the member functions with the reserved word `static`. By doing this, we are telling the compiler that this variable or this function refers to the class in general and not to a specific object. We can also consider that this data or function is being shared with all the instances of said object.

The following example defines a static member variable and a static member function:

```
class Dog {
// ...
static int numberOfDogs; //would normally be private
static int howmanyDogs() { return numberOfDogs; }
};
```

Accessing them can be done in two ways:

- From an object in the Dog class.

```
Dog sultan = new Dog();
sultan.numberOfDogs; //would normally be private
sultan.howmanyDogs();
```

- Using the class identifier without defining any objects.

```
Dog::numberOfDogs; //would normally be private
Dog::howmanyDogs();
```

There is an important aspect we need to remember: member variables and functions which are `static` always refer to the class and not to a specific object, meaning that the object `this` does not exist.

One consequence of this is that in static member functions we can not make either a direct or an indirect reference to the object `this` and:

- They can only make calls to static functions as non-static functions will always implicitly wait for said object as a parameter
- They can only access static variables because non-static variables are always accessed through said object.
- These functions can not be declared as `const` as this would make no sense.

2.4. The organisation and use of libraries in C++

Up to now we have always included the class definition and the code which uses it in the same archive, however this is not the best layout if we want to re-use the information.

We recommend dividing the source code of the class into two files, so separating the definition of the class and its implementation.

- The header file incorporates the definition of the class. The extension of these files can vary between several possibilities and the final decision is an arbitrary one.

- The implementation file for the class methods containing an "include" for the header file. The extension used for these files may also vary.

Note

The standard extensions used are .hpp (used more in Windows environments), .H and .hxx (used more in Unix environments) and even .h (the same as in C).

Note

The most standard extensions used are .cpp (more frequently used in Windows environments) .C and .cxx (more frequently used in Unix environments).

When we want to use this class later on we only need to include a call to the class header file in the source code.

In our example, the implementation is found in the file dog.cpp, which has an `include` for the header file dog.hpp.

File dog.hpp (class header file)

```
class Dog
{
  public:

  Dog(int age);   //constructor methods
  Dog();
  ~Dog();          // destructor method
  int getAge(); // access methods
  int assignAge(int);
  int assignHeight(int);
  int getHeight();
  void bark();     // actual methods

  private:
  int age;
  int height;
};
```

File dog.cpp (implementation file for the class)

```
  #include <iostream>   //needed for cout
  #include <dog.hpp>

  Dog:: bark()
  { cout << "Woof"; }

  void Dog:: assignHeight (int nHeight)
  { height = nHeight; }

  int Dog:: getHeight (int nHeight)
  { return (height); }

  void Dog:: assignAge (int nAge)
  { age = nAge; }

  int Dog:: getAge()
  { return (age); }
```

File example.cpp

```
  #include <dog.hpp>
  int main()
  {
    //Initialising the object with age 4.
    Dog sultan(4);
     sultan.bark();
  }
```

2.4.1. Standard libraries

Compilers usually include a set of additional functions for the programmer to use. GNU provides a standard library of functions and objects for C++ programmers, this is called libstdc++.

This library provides input/output operations using *streams, strings*vectors, lists, comparison algorithms, mathematical operations and ordering algorithms, among many others.

2.4.2. Using STL libraries

C++ has incorporated a new level of abstraction with the introduction of **templates**, also known as ***parameterised types***. This subject does not fall under the scope of this course but the inclusion of the STL (Standard Template Library) in C++ is a very powerful feature and we should take a brief look at how it is used.

The basic idea of templates is simple: When we implement a general operation using an object (a list of dogs for example) we define the various operations for manipulating a list based on the class Dog. If we then want to perform a similar operation with other objects (a list of cats), the resulting code for the maintenance of the list is similar but the elements are defined based on the class Cat. The best way to go about this will be to make a copy and paste and modify the copied block so that it works with the new class. However, this process must be repeated each time we want to implement a new list with another type of object (a list of horses for example).

What is more, each time we want to modify an operation for the lists, we will have to change each of the customisations. We can therefore quickly see that this implementation will not be efficient.

A more efficient way to do it would be to generate a generic code that performs the operations of the lists for a type which can be indicated later on. This generic code is what we call a template, or a parameterised type.

From this brief summary we can see the efficiency and power of this new attribute, and also its complexity, but, as we have mentioned, this goes beyond the scope of this course. However, this subject is essential for a deeper understanding of C++ and we recommend you read other textbooks in order to learn about it.

While the definition and implementation of a template library may become extremely complicated, the use of Standard Template Libraries (STL) is somewhat easier.

In the following example we will work with the class `set` which defines a set of elements. As such, we will include the `set` library which is contained in the STL.

```
#include <iostream>
#include <set>

int main()
{
// define a set of integers <int>
set<int> setNumbers;
//add three numbers to the set of numbers
setNumbers.insert(123);
setNumbers.insert(789);
setNumbers.insert(456);

// show how many numbers the
// set of numbers has
cout << "Number set: "
     << setNumbers.size() << endl;

// repeat the process with a set of letters
//define the set of characters <char>
set<char> setLetters;

setLetters.insert('a');
setLetters.insert('z');
cout << "Letter set: "
     << setLetters.size() << endl;
return 0;
}
```

In this example we have defined a set of numbers and a set of letters. For the number set we have defined the variable `setNumbers` using the template `set` indicating that we will use elements of the type `<int>`. This set defines several methods among which is that of inserting an element in the list (`.insert`) and counting the number of elements (`.size`). For the second of these we have defined the variable `setLetters` also using the same template `set` but now the elements are of type `<char>`.

The output of the program will show the number of letter entered into the set of numbers and then the number of elements entered in the set of letters.

3. Designing object oriented programs

The power of the object oriented programming paradigm does not only lie in the definition of classes and objects but in the consequences they imply which can also be used in the programming language.

In this unit we will look at the main properties of this paradigm:

- Homonymy
- Inheritance
- Polymorphism

Once we understand the scope of this paradigm shift, we can apply new rules for the design of applications.

3.1. Homonymy

As its name indicates, homonymy refers to the use of two or more meanings (in our case, operations) having the same name.

In object oriented programming, homonymy refers to the use of the same name to define the same operation several times in different situations, although the underlying idea is usually the same. The easiest example would be to define operations which basically have the same objective but for different objects, with the same name.

In our case we can distinguish between two main forms: what is **homonymy** (or **the overloading**) **of functions** and **homonymy of operators**.

3.1.1. Overloading of functions and methods

We have already seen how function overloading is one of the improvements which makes C++ much more flexible than C, it is one of the most frequently used attributes within the definition of classes.

Constructors are a practical example of method overloading. Each class will have a default constructor which has no parameters and initialises the objects for this class.

In our example,

```
Dog::Dog()
{ }
```

As we have also seen, we could have the situation in which we always want to initialise this class using a certain age, or a certain age and a certain height.

```
Dog::Dog(int nAge) // New age of the dog
{ age = nAge; }

Dog::Dog(int nAge, int n:height) // New defin.
{
  age = nAge;
  height = nHeight;
}
```

In all three cases we are creating an instance of the object `Dog`. They are therefore basically performing the same operation although the final result will be slightly different.

Any other method or function of a class can be overloaded in the same way.

3.1.2. Overloading operators

In the end, an operator is really nothing more than a simplified way of expressing an operation on one or more operandi, while a function allows us to perform more complex operations.

Operator overloading is therefore a way of simplifying expressions for operations between objects.

In our example we could define a function to increase the age of an object `Dog`.

```
Dog Dog::increaseAge()
{
  ++age;
  return (*this);
}
// the resulting call would be Sultan.IncreaseAge()
```

Although the function is very simple, it may be a bit unwieldy to use. In this case we could consider overloading the operator ++ so that, when applying it to a `Dog` object, it automatically increases the age.

Overloading an operator is declared in the same way as for a function. We use the reserved word `operator` followed by the operator to be overloaded. Single-operator functions do not have parameters (with the exception of the increase or decrease postfix which uses an integer to distinguish it).

```
#include <iostream>
class Dog
```

```
{
  public:
   Dog();
   Dog(nAge);
   ~Dog();
   int getAge();
   const Dog & operator++();     // Operator ++i
   const Dog & operator++(int);  // Operator i++

  private:
   int age;
};

Dog::Dog():
   age(0)
{ }

Dog::Dog(int nAge):
   age(nAge)
{ }

int Dog::getAge()
{ return (age); }

const Dog & Dog::operator++()
{
  ++age;
  return (*this);
}

const Dog & Dog::operator++(int x)
  {
   Dog temp = *this;
   ++age;
   return (temp);
}

int main()
{
  Dog sultan(3);
  cout << "The age of Sultan at the start of the program \n " ;
  cout << sultan.getAge() << endl;

  ++sultan;
  cout << "The age of Sultan after one birthday \n " ;
  cout << sultan.getAge() << endl;
```

```
    sultan++;
    cout << "The age of Sultan after another birthday \n " ;
    cout << sultan.getAge();
  }
```

In the operator overloading declaration we can see how it returns a `const` reference to an object of the type `Dog`. This protects the address of the original object from undesired modification.

We can also see how the declarations for the postfix and prefix operator are practically the same and how only the argument type changes. To distinguish between these cases, a convention has been established that the postfix operator should have a parameter of the `int` kind in the declaration (although this parameter is not used).

```
  const Dog & operator++();     // Operator ++i
  const Dog & operator++(int);  // Operator i++
```

There are also significant differences in the implementation of both these functions:

- In the case of the prefix operator, the age value of the object is incremented and the modified object is returned through the pointer `this`.

```
  const Dog & Dog::operator++()
  {
    ++age;
    return (*this);
  }
```

- In the case of the postfix operator, we want to return the value of the object before it has been modified. For this reason we establish a temporary variable which collects the original object, modifies it and returns the temporary variable.

```
  const Dog & Dog::operator++(int x)
  {
    Dog temp = *this;
    ++age;
    return (temp);
  }
```

The definition of the overloading of the sum operator, which is a binary operator, would be as follows:

```
  // In this case, the summing of two Dog type objects
  // makes NO LOGICAL SENSE.
```

```
// EXCEPT to show how the
// declaration of the operator would be, one
// "possible" result would be to
// return the object Dog on the left hand side of
// the summing operator with the age corresponding to
// the sum of the ages of the two dogs.

const Dog &Dog::operator+(const Dog & rhs)
{
  Dog temp = *this;
  temp.age = temp.age + rhs.age;
  return (temp);
```

Note

Given the disconcerting logic used in the above example, it becomes clear that we should not abuse operator overloading. It should only be used in those cases in which it is more intuitive and makes the program more legible.

3.2. Simple inheritance

Objects are not isolated elements. When we study objects we establish relationships between them which help us to understand them better.

Example

A dog and a cat are different objects but they have one thing in common: they are both mammals. Dolphins and whales are also mammals, but they live in a very different environment, sharks however are not, they fall under the category of fish. What do all these objects have in common? They are all animals.

We can establish a hierarchy of objects in that a dog is a mammal, a mammal is an animal, an animal is a living thing etc. We are establishing a relation between them: *is a*. This type of relationship is very common: a pea is a seed, which is a type of vegetable; a hard disk is a storage unit which is, in turn, a component of a computer.

When saying that one element is a type of another we are establishing a specialisation: We are saying that the element has some general attributes and also others which are its own.

Inheritance is a way of representing attributes which are received from the more general level.

The dog concept inherits all the attributes of mammal, meaning it produces milk, it breathes using lungs, it moves etc, but it also has its own specific attributes such as barking or wagging its tail. Dogs can also be divided up into breeds: German shepherd, poodle, doberman etc. Each one has its own peculiarities but they inherit all the attributes of dogs.

In order to represent these relationships, C++ allows us to derive one class from another. In our case, the Dog class is derived from the Mammal class. We therefore do not need to indicate that the Dog class suckles, breathes using lungs, or that it moves. As a mammal, it inherits these properties on top of its own data or functions.

In the same way a Mammal can be implemented as a class deriving from the Animal class which in turn inherits information from the class of living and moving things.

Given the relationship between the Dog class and the Mammal class, and between the Mammal class and the Animal class, the Dog class also inherits the information of the Animal class. A dog is a living thing that moves!

3.2.1. The implementation of inheritance

To express this relationship in C++, after the name in the declaration we put a colon(:), the type of derivation (public or private) and the name of the class it is derived from.

```
class Dog : public Mammal
```

We will look at the derivation type later although for the moment we will consider it to be public. We will now focus our attention on the appearance of the new implementation:

```
#include <iostream>
enum BREEDS { GERMAN_SHEPHERD, POODLE,
             DOBERMAN, YORKSHIRE };
class Mammal
{
  public:
   Mammal();                  // constructor method

 ~Mammal();                   // destructor method
void assignAge(int nAge)
 { age = nAge } ;             // access methods
int getAge() const
 { return (age) };
protected:
 int age;
};

class Dog : public Mammal
{
  public:
   Dog();                      // constructor method
```

```
    ~Dog();                     // destructor method
    void assignBreed(BREEDS);   // access methods
    int getBreed() const;
    void bark() const
     { cout << "Woof "; };      // own methods
  private:
    BREEDS breed;
};

class Cat : public Mammal
{
  public:
    Cat();                      // constructor method
    ~Cat();                     // destructor method
    void miaow() const
     { cout << "Miaowww"; }     // own methods
};
```

In the implementation of the Mammal class, we first described the default constructor and destructor. Given that the member data `age` that is present in the Dog class is not an exclusive attribute of this class as all mammals have an age, we have transferred the member data `age` and its access methods (`getAge` and `assignAge`) to the new class.

Note

It should be stressed that the declaration of the type `protected` for the member data `age` allows it to be accessed from the derived classes. If we had declared it as `private`, the other classes would not have been able to see it or use it, not even the derived classes. If we had declared it as `public`, it would be visible from any object, however this is not advisable.

We have added the new breed data to the Dog class and we have defined its access methods (`getBreed` and `assignBreed`), as well as its predefined constructor and destructor. We have kept the method `bark` as a function of the Dog class: other mammals do not bark.

3.2.2. Constructors and destructors of derived classes

As the Dog class is derived from the Mammal class, `Dog` objects are essentially `Mammal`. It therefore will first call its base constructor, which will create a Mammal and we then fill in the rest of the information by calling the Dog constructor.

Mammal data
Dog data

When destroying an object in the Dog class, the process is reversed: First we call the Dog destructor, thus releasing the specific information, then we call the Mammal destructor.

We have already seen how to initialise the data of an object in the class we are defining, but it is also common that in the constructor of a class we will want to initialise the data belonging to its base class.

Example

Carrying on with the example of the Dog class, as well as initialising its breed we could also initialise its age (as it is a mammal it will have an age).

The constructor for the Mammal class also performs this task but we may only be interested in performing this operation for dogs and not for all animals. In this case we can perform the following initialisation in the constructor of the Dog class:

```
Dog :: Dog()
{
  assignBreed(POODLE);  // Access to breed
  assignAge(3);         // Access to age
};
```

As `assignAge` is a method belonging to the base class, it is automatically recognised.

In the example above we have defined two methods: `bark` and `miaow` for the classes Dog and Cat respectively. Most animals have the ability to make sounds to communicate, so we could therefore create a common method which we could call `makeSound`, and we could give this a general value for all animals except for dogs and cats, to which we give an individual value:

```
#include <iostream>

enum BREEDS { GERMAN_SHEPHERD, POODLE,
             DOBERMAN, YORKSHIRE };

class Mammal
{
  public:
  Mammal();                  // constructor method
                             // destructor method
  ~Mammal();
  void assignAge(int nAge)
   { age = nAge; } ;         // access methods
  int getAge() const
   { return (age); };
  void makeSound() const
   { cout << "Sound"; };
  protected:
   int age;
};
class Dog : public Mammal
{
  public:
   Dog();                       // constructor method
  ~Dog();                       // destructor method
  void assignBreed(BREEDS);     // access methods
  int getBreed() const;
  void makeSound() const
{ cout << "Woof "; };           // own methods
  private:
   BREEDS breed;
};

class Cat : public Mammal
{
  public:
   Cat();                       // constructor method
   ~Cat();                      // destructor method
   void makeSound() const
   { cout << "Miaowww"; }       // own methods
};

int main()
{
  Dog adog;
  Cat acat;
  Mammal amammal;

  amammal.makeSound;            // Result: "Sound"
  adog.makeSound;               // Result: Woof
  acat.makeSound;               // Result: Miaow
}
```

The method makeSound will have an end result depending on whether we call a Mammal, a Dog or a Cat. With derived classes (Dog and Cat) it is said that we have **redefined the member function** of the base class. As such, the derived class must define the base function with the same signature (return type, parameters and their types and the specifier const).

We need to distinguish between function redefinition and function overloading. The first of these deals with functions with the same name and the same signature in different classes (the base class and the derived class). In the second case they are functions with the same name but a different signature that are within the same class.

The result of the redefinition of functions is the **hiding** of the base function from the derived classes. In this respect we need to bear in mind that if we redefine a function in a derived class, all the overloads of this function in the base class will also be hidden. An attempt to use a hidden function will generate a compilation error. The solution will be to carry out the same function overloads in the derived class as in the base class.

However, if we want to we can access the hidden method by typing the name of the base class before the name of the function, followed by the scope operator (::).

```
adog.Mammal::makeSound();
```

3.3. Polymorphism

In the example, we have been using up to now we have only looked at how the class Dog (a derived class) inherits data and methods from the Mammal class (base class). In fact, this relationship is much stronger.

C++ allows the following types of expressions:

```
Mammal *ap_amammal = new Dog;
```

In these expressions we do not assign an object from the Mammal class to a pointer to a Mammal class but we assign it an object from a different class, the Dog class, although this complies with the situation that Dog is a derived class of Mammal.

This is, in fact, the nature of polymorphism: the same object can have different forms. We can assign a mammal object or an object from any of its derived classes to a pointer to a mammal object.

What is more, this assignation can be done during execution, as we shall see later on.

3.3.1. Virtual functions

Using the pointer introduced below we can call any method from the Mammal class. In this specific case, though, it would be really useful to be able to call the corresponding methods of the Dog class. We can do this using **virtual functions and methods**:

```
#include <iostream>

enum BREEDS { GERMAN_SHEPHERD, POODLE,
             DOBERMAN, YORKSHIRE };

class Mammal
{
  public:
   Mammal();              //constructor method
   virtual ~Mammal();         // destructor method
   virtual void makeSound() const
    { cout << "make a sound" << endl; };
  protected:
   int age;
};

class Dog : public Mammal
{
  public:
   Dog();                         // constructor method
   virtual ~Dog();                // destructor method
   int getBreed() const;
   virtual void makeSound() const
    { cout << "Woof" << endl; };  // own methods
  private:
   BREEDS breed;
};

class Cat : public Mammal
{
  public:
   Cat();                         // constructor method
   virtual ~Cat();                // destructor method
   virtual void makeSound() const
    { cout << "Miaow" << endl; }; // own methods
};

class Cow : public Mammal
{
  public:
   Cow();                          // constructor method
   virtual ~Cow();                 // destructor method
   virtual void makeSound() const
    { cout << "Moo" << endl; };    //own methods
};

int main()
{
  Mammal * ap_mammals[3];
  int i;

  ap_mammals [0] = new Cat;
  ap_mammals [1] = new Cow;
  ap_mammals [2] = new Dog;

  for (i=0; i<3 ; i++)
   ap_mammals[i] -> makeSound();
return 0;
}
```

On executing the program, first it declares a vector with 3 pointer type elements to Mammal and then it initialises several types of Mammals (Cat, Cow and Dog).

It then runs through this vector and calls the `makeSound` method for each of the elements. The output obtained is:

- `Miaow`
- `Moo`
- `Woof`

The program detects the type of object which has been created using `new` and calls the function `makeSound` for each one.

This would have worked the same if the user had been asked to indicate the order of the animals to the program. The internal operation of the program is based on detecting the type of object being pointed to during execution, this then replaces the virtual functions of the object of the base class with those corresponding to the derived object.

To do all this we defined the member function `makeSound` of the Mammal class as a virtual function.

3.3.2. The declaration of virtual functions

When we declare a function of a base class to be virtual we are implicitly declaring the functions of the derived class to be virtual, meaning they do not have to be explicitly declared as such. However, we recommend doing so to make the code clearer.

If the function is not declared as being virtual the program will understand that it must call the function of the base class, whatever type of pointer it is.

It is also important to note that these are always pointers to the base class (although it has been initialised with an object from the derived class), meaning they can only access functions of the base class. If one of these pointers attempts to access a specific function from the derived class, such as, for example, `getBreed()`, it will cause an unknown function error. These types of functions can only be directly accessed from pointers to objects of the derived class.

3.3.3. Virtual constructors and destructors

Constructors can not be virtual by definition. In our case, on initialising `new Dog` we are calling the constructor for the Dog class and that for the Mammal class, meaning a pointer is already created to the derived class.

When working with these pointers, one possible operation is to delete them. For their destruction we would therefore want to call the destructor from the derived class and then that from the base class. To do this we only have to declare the base class destructor to be virtual.

The practical rule to follow is to declare a destructor to be virtual when there are virtual functions within the class.

3.3.4. Abstract data types and pure virtual functions.

We have already said that classes correspond to the level of the ideas while objects correspond to specific elements.

We could therefore have a class in which it would not make sense to instance an object, while it would make sense to instance them in derived classes. Here we are talking about classes which we want to keep strictly in the realm of ideas, while their derived classes generate our objects.

An example could be the class WorkOfArt having the derived sub-classes: Painting, Sculpture, Literature, Architecture etc. We could consider the WorkOfArt class to be an abstract concept and we could refer to specific works based on a type of art (one of the derived classes). The criteria for declaring a class as an abstract data type will always be subjective and will depend on how we want to use the classes in the application.

```
class WorkOfArt
{
public:
  WorkOfArt();
  virtual ~WorkOfArt ();
  virtual void showWorkOfArt() = 0; //virtual pure
  void assignAuthor(String author);
  String getAuthor();
  String author;
};

class Painting : public WorkOfArt
{
  public:
  Painting();
  Painting ( const Painting & );
  virtual ~Painting ();
  virtual void showWorkOfArt();
  void assignTitle(String title);
  String getTitle();
private:
  String title;
};

Painting :: showWorkOfArt()
{ cout << "Photograph Painting \n" }
class Sculpture : public WorkOfArt
{
public:
  Sculpture();
  Sculpture ( const Sculpture & );
  virtual ~Sculpture ();
  virtual void showWorkOfArt();
  void assignTitle(String title);
  String getTitle();
private:
  String title;
};

Sculpture :: showWorkOfArt()
{ cout << "Photograph Sculpture \n" }
```

Within this abstract class we have defined a virtual function which will show a reproduction of the work of art. This reproduction will vary depending on the type of art. It could be in the form of a photograph, a video, a reading of a literary or theatrical text etc.

To declare the WorkOfArt class as an abstract data type, which is therefore not instanceable by any object, we only need to declare a **pure virtual function**.

To assign a pure virtual function we take a virtual function and assign it to 0:

```
virtual void showWorkOfArt() = 0;
```

Now, when we try to instance a WorkOfArt object (using `new WorkOfArt`), it would generate a compilation error.

When declaring a pure virtual function we also need to remember that this member function will also be inherited. We therefore need to redefine this function in the derived classes. If it is not redefined, the derived class will automatically become another abstract class.

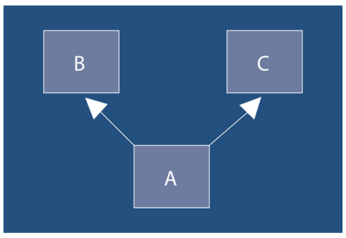
3.4. Avanced operations using inheritance

Despite the power provided by simple inheritance, sometimes it is not enough. We will now give you a brief introduction to the more advanced concepts relating to inheritance.

3.4.1. Multiple inheritance

Multiple inheritance allows a class to be derived from more than one base class.

Figure 12.



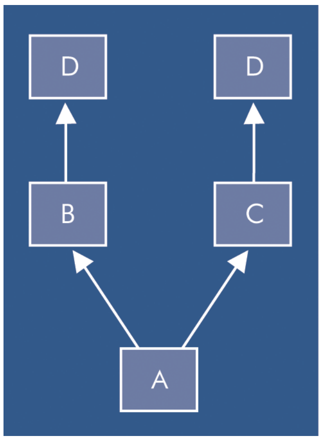
```
class A : public B, public C
```

In this example class A is derived from class B and class C. This situation raises several questions:

- What happens when the two derived classes have a function with the same name? It could cause a conflict for the compiler due to ambiguity, this could be resolved by adding a virtual function to class A which redefines this function, this would explicitly resolve the ambiguity.

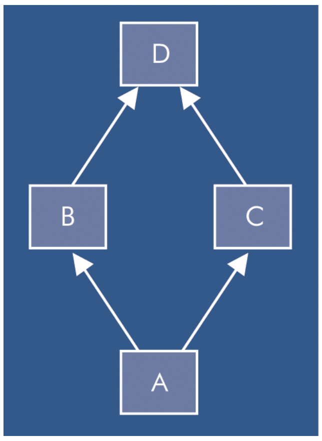
- What happens if the classes are derived from a common base class? If class A is derived from class D passing through class B and class C we will have two copies of class D (see the illustration), this can cause ambiguities. In this case the solution is provided by **virtual inheritance**.

Figure 13.



Using virtual inheritance we can tell the compiler that we only want one shared class D, meaning that the B and C classes are defined as virtual.

Figure 14.



```
class B: virtual D
class C: virtual D
class A : public B, public C
```

Generally a class only initialises its variables and its base class. When we declare a class to be virtual, the constructor that initialises the variables will be that of the most derived class.

3.4.2. Private inheritance

Sometimes we do not need, or even want, derived classes to have access to the data and functions of the base class. In this case we will use private inheritance.

With private inheritance, the variables and member functions of the base class are considered to be private, independently of the accessibility declared in the base class. This means that the functions inherited from the base class will not be accessible to any function which is not a member of the derived class.

3.5. Guidelines for the analysis and design of programs

The complexity of modern software projects requires that we use the "divide and conquer" strategy when analysing a problem, breaking down the original problem into smaller parts that are easier to deal with.

The traditional way of doing this was to break the problem down into simpler function or processes (top-down design) so that we got a hierarchical structure of processes and sub-processes.

In object oriented programming we break the problem down in a different way by focusing on the objects it includes and the relationships between them, not on the functions involved.

This process is known as:

- **Conceptualisation**. Projects normally arise from an idea which will guide its development. It is very useful to identify the general goal we wish to achieve and to keep this in focus during the various phases of the project.

- **Analysis**. This involves determining the needs (requirements) that the program should cover. During this phase all efforts should be focused on understanding the domain (the scope) of the problem in the real world (which elements are involved and which are related) and capturing the requirements.
  The first step in the analysis of the requirements is to identify the **use cases**, which are descriptions of the various processes of the domain in normal language. Each use case describes the interaction between an actor (a person or an element) and the system. The actor will send a message to the system and this will act in consequence (responding, cancelling, acting on another element etc.).
  From a complete set of use cases we can begin to develop the **domain model**, this is a document which contains all the information we know about the domain. Part of this modelling process is to describe all the objects which are involved (which may also come to be the design classes).
  The model is often expressed in **UML** (unified modelling language), however we will not go into that in this unit.

Example

If we were designing an application for a cash machine, one use case would be the withdrawal of cash from an account.

Example

The following would be objects in the cash machine project: the client, the cash machine, the bank, the receipt, the money, the credit card etc.

From the use cases we can describe various **scenarios**, these are specific scenarios in which the use case is developed. Working like this, we can complete the set of possible interactions for our model. Each scenario will also describe an environment with prior conditions and elements that activate it.
All these elements can be represented graphically using diagrams to show the interactions.
We also need to consider the restrictions of the scope in which they are working and other requirements of the client.

Note

UML is a convention which was established for the representation of the information in a model.

Example

In the cash machine project, one possible scenario would be that the client wishes to withdraw money from an account and there are insufficient funds.

- **Design**. Using the information from the analysis we focus on the problem to create the solution. We can consider the design to be the conversion of the requirements to an implementable software model. The result is a document that contains the design plan.
Firstly, we will need to identify the classes involved. The first (and simplest) step will be to write down the various scenarios and to create a class for each one. Later on we can consolidate them by grouping the synonyms.
Once the classes of the model have been defined, we can add classes that will be useful for the implementation of the project (views, reports, classes for conversions and data handling, the use of devices etc.).
Once we have established the initial set of classes (these can be modified later on) we can proceed with the modelling of the relationships and the way they interact. One of the most important points in the definition of a class is to determine its **responsibilities**: It is a basic principle that a class is responsible for something. If we can clearly identify this unique responsibility, the resulting code will be much easier to maintain. Those responsibilities which do not correspond to a class are delegated to related classes.
During this phase we also establish the **relationships** between the objects in the design that may or may not coincide with the objects in the analysis. They may be of different types. The type of relationship we have looked at most in this unit are generalisation relationships that have subsequently been implemented using public inheritance, but there are others and each one has its forms of implementation.
The design information is completed with the inclusion of the **dynamics** between the classes: the modelling of the interaction between the classes themselves using different types of diagrams.

The document that contains all the information on the design of a program is called the *design plan*.

- **Implementation**. In order to be able to apply the project we must convert the design plan into source code, in our case, to C++. The selected language will provide all the tools and working mechanisms to translate the definitions of classes, their requirements, their attributes and their relationships from the world of design to the actual environment. This phase is focused on efficiently coding each of the elements of the design.

- **Testing**. During this phase we test that the system does what it is supposed to do. If it does not, we will need to revise the specifications on the analysis, design and implementation level. The design of a good series of tests based on the use cases can avoid many problems with the final product. It is always better to have good series of tests which generate many failures during the analysis, design and implementation phases (allowing them to be corrected) than to find out about these errors during distribution.

- **Distribution**. An implementation of the program (prototype) is sent to the client for evaluation or for final installation.

3.5.1. Ways of developing a project

This is usually carried out using a **cascade process**: each of the phases is completed and once finished and reviewed it is passed to the next phase without being able to go back to the previous step.

This method appears to be ideal in theory but in practice it is terrible. For this reason object oriented design and analysis usually uses an **iterative process**. As the software progresses, the phases are repeated to achieve greater refinement, this means it can be adapted to the changes brought about by a greater understanding of the project by the designers, the developers and also the client.

This method provides another advantage in real life: it allows finished versions to be delivered even though they may need further refining. This allows us to introduce the idea of "adequate" software versions which can then be refined further depending on the needs of the client.

Summary

In this unit we have progressed from the C programming environment that uses the imperative model and is based on sequences of instructions, to an object oriented model which is based on the use of objects and their relationships.

We have had to come to understand the advantages of using a more abstract working model, but also one that is closer to the description of the entitities that must be managed in the real world and their relationships. These allow us to focus our attention on the concepts we wish to implement more than on the lines of code that will eventually be produced.

We have also looked at the tools provided by C++ for implementation: classes and objects. As well as their definition, we also reviewed the main attributes of the object oriented model used in C++. We have looked at inheritance between clases, homonymy and polymorphism.

Lastly we have seen that, due to changes in the philosophy of the new programming paradigm, we can not apply the same principles to the design of programs and we have therefore introduced new design rules that are more appropriate.

Self-evaluation

1. Designing an application which simulates the operation of a lift. The application should initially define three applications:

LIFT: [1] Enter [2] Exit [0] End

After each operation, it must show the occupancy of the lift.

[1] Enter signifies that a person goes into the lift.
[2] Exit signifies that a person leaves the lift.

2. Expand the above exercise to incorporate the following requirements:

- An operation that shows the status of the lift

LIFT: [1] Enter [2] Exit [3] Status [0] End

- Limit the capacity of the lift to 6 people.
- Limit the load of the lift to 500 kg.
- Request the code and weight of users to allow them to access the lift as per the established limits.

If access is denied to the user it should display a message saying why:

- < The lift is full >
- < The maximum weight limit has been exceeded >

If the user is allowed to enter the lift, the following message should be displayed: < #Code# enter the lift>.

When someone enters the lift, it should display a greeting message to occupants (<#code# says> Hello) and the others should reply individually with the message (<#code# replies> Hello).

When someone leaves the lift, their code must be requested and the load in the lift must be updated, the following message must be displayed: <#code# leave the lift>.

When someone leaves the lift, it should display the following message (<#code# says> Goodbye) and the others should reply individually with the message (<#code# replies> Goodbye).

To simplify things we will assume that no two passengers have the same code.

After each operation, it should be able to display the status of the lift (occupancy and load).

3. Expand the above exercise to incorporate three possible languages in which the users can speak.

On entering, the user should also be asked which language they want to use:

LANGUAGE: [1] Catalan [2] Spanish [3] English

- In Catalan the greeting is "Bon dia" and the farewell is "Adéu".
- In Spanish the greeting is "Buenos días" and the farewell is "Adiós".
- In English the greeting is "Hello" and the farewell is "Bye".

Answer key

1.

```
lift01.hpp
class Lift {
  private:
   int occupancy;
   int maximumoccupancy;
  public:
   // Constructors y destructors
   Lift();
   ~Lift();

   // Access functions
   void showOccupancy();
   int getOccupancy();
   void modifyOccupancy(int difOccupancy);

   // Functions of the method
   bool person_mayEnter();
   bool person_mayLeave();

   void person_enter();
   void person_leave();
};

lift01.cpp
#include <iostream>
#include "lift01.hpp"

// Constructors y destructors
Lift::Lift():
  occupancy(0), maximumoccupancy(6)
{ }

// Access functions
int Lift::getOccupancy()
{ return (occupancy); }

void Lift::modifyOccupancy(int difOccupancy)
{ occupancy += difOccupancy; }

void Lift::showOccupancy()
```

```
{ cout << "Current occupancy: " << occupancy << endl;}

bool Lift::person_mayEnter()
{ return (true); }

bool Lift::person_mayLeave()
{
  bool isOccupied;
  if (getOccupancy() > 0) isOccupied = true;
  else isOccupied = false;

  return (isOccupied);
}

void Lift::person_enter()
{ modifyOccupancy(1); }

void Lift::person_leave()
{
  int currentOccupancy = getOccupancy();
  if (currentOccupancy>0)
   modifyOccupancy(-1);
}
```

```
exerc01.cpp
#include <iostream>
#include "lift01.hpp"

int main( int argc, char *argv[] )
{
  char opc;
  bool leave = false;
  Lift aLift;
  do
  {
  cout << endl
  cout << "LIFT: [1] Enter [2] Exit    [0] End ";
    cin >> opc;
    switch (opc)
    {
```

```
    case '1':
     cout << "opc Enter" << endl;
     unLift.person_enter();
     break;
    case '2':
     cout << "opc Leave" << endl;
     if (unLift.person_mayLeave())
      unLift.person_leave();
     else cout << "Lift empty " << endl;
     break;
    case '0':
     leave = true;
     break;
    }
    unLift.showOccupation();
   } while (! leave);
   return 0;
 }
```

2.

```
lift02.hpp
#ifndef _LIFT02
#define _LIFT02
#include "person02.hpp"
class Lift {
  private:
   int occupancy;
   int load;
   int maximumOccupancy;
   int maximumLoad;
   Person *passengers[6];
  public:
   // Constructors y destructors
   Lift();
   ~Lift();

   // Access functions
   void showOccupancy();
   int getOccupancy();
   void modifyOccupancy(int difOccupancy);
   void showLoad();
   void modifyLoad(int difLoad);

   void showListPassengers();

   // Functions of the method
   bool person_mayEnter(Person *);
   bool person_select(Person *locatePerson,
                            Person **onePerson);

   void person_enter(Person *);
   void person_leave(Person *);

  void person_greetRestLift(Person *);
  void person_farewellRestLift(Person *);
};
#endif

lift 02.cpp
#include <iostream>
#include "lift02.hpp"
```

```
//
// Constructors y destructors
//

// In the constructor we initialise the maximum values
// for maximum occupancy and load of lift
// and the passengers vector to NULL pointers

Lift::Lift():
  occupancy(0), load(0),
  maximumOccupancy(6), maximumLoad(500)
{ for (int i=0;i<=5;++i) {passengers[i]=NULL;} }

Lift::~Lift():
{ // Free codes of passengers
  for (int i=0;i<=5;++i)
  { if (!(passengers[i]==NULL)) {delete(passengers[i]);} }
}

// Access functions
int Lift::getOccupancy()
{ return (occupancy); }

void Lift::modifyOccupancy(int difOccupancy)
{ occupancy += difOccupancy; }


void Lift::showOccupancy()
{ cout << "Current occupancy: " << occupancy ; }


int Lift::getLoad()
{ return (load); }


void Lift::modifyLoad(int difLoad)
{ load += difLoad; }


void Lift::showLoad()
{ cout << "Current load: " << load ; }


bool Lift::person_mayEnter(Person *onePerson)
{
  bool tmpMayEnter;
  // if occupancy does not exceed the occupancy limit and
  // if the load does not exceed the load limit
  // -> may enter

  if (occupancy + 1 > maximumOccupancy)
  {
   cout << " Warning: Lift is full. You may not enter. "
   cout << endl;
   return (false);
  }

  if (onePerson->getWeight() + load > maximumLoad)
  {
   cout << " Warning: The lift has exceeded its maximum load.
   cout << " You may not enter. " << endl;
   return (false);
  }
  return (true);
}
bool Lift::person_select(Person *locatePerson,
                                   Person **onePerson)
{
  int counter;
// Select a passenger in the lift.
bool personFound = false;
if (getOccupancy() > 0)
{
  counter=0;
  do
  {
   if (passengers[counter]!=NULL)
  {
   if ((passengers[counter]->getCode()==
     locatePerson->getCode() ))
   {
    *onePerson=passengers[counter];
     personFound=true;
     break;
   }
  }
  counter++;
 } while (counter<maximumOccupancy);
 if (counter>=maximumOccupancy) {*onePerson=NULL;}
```

```
  }
  return (personFound);
}
void Lift::person_Enter(Person *onePerson)
{
  int counter;
  modifyOccupancy(1);
  modifyLoad(onePerson->getWeight());
  cout << onePerson->getCode();
  cout << " enter the lift " << endl;
  counter=0;
  // we have already checked that there is space
  do
  {
   if (passengers[counter]==NULL )
   {
    passengers[counter]=onePerson;
    break;
   }
   counter++;
  } while (counter<maximumOccupancy);
}
void Lift::person_Leave(Person *onePerson)
{
int counter;
  counter=0;
  do
  {
   if ((passengers[counter]==onePerson ))
   {
    cout << onePerson->getCode();
    cout << " leave the lift " << endl;
    passengers[counter]=NULL;
    // Modify the occupancy and the load
    modifyOccupancy(-1);
    modifyLoad(-1 * (onePerson->getWeight()));
    break;
   }
   counter++;
  } while (counter<maximumOccupancy);
  if (counter == maximumOccupancy)
  { cout << "No one has this code. ";
    cout << "Nobody leaves the lift" << endl;}
```

```
}
void Lift::showListPassengers()
{
  int counter;
  Person *onePerson;
  if (getOccupancy() > 0)
  {
   cout << "List of passengers in the lift: " << endl;
   counter=0;
   do
   {
    if (!(passengers[counter]==NULL ))
    {
     onePerson=passengers[counter];
     cout << onePerson->getCode() << "; ";
    }
    counter++;
   } while (counter<maximumOccupancy);
   cout << endl;
  }
  else
  { cout << "The lift is empty" << endl; }
}
void Lift::person_greetRestLift( Person *onePerson)
{
  int counter;
  Person *otherPerson;
  if (getOccupancy() > 0)
  {
   counter=0;
   do
   {
    if (!(passengers[counter]==NULL ))
    {
     otherPerson=passengers[counter];
     if (!(onePerson->getCode()==
         otherPerson->getCode()))
     {
      cout << otherPerson->getCode();
      cout << " reply: " ;
      otherPerson->greet();
      cout << endl;
      }
     }
    counter++;
    } while (counter<maximumOccupancy);
  }
}
void Lift::person_farewellRestLift( Person *onePerson)
{
  int counter;
  Person *otherPerson;

  if (getOccupancy() > 0)
  {
   counter=0;
   do
   {
    if (!(passengers[counter]==NULL ))
    {
     otherPerson=passengers[counter];
     if (!(onePerson->getCode()==
         otherPerson->getCode()))
     {
      cout << otherPerson->getCode();
      cout << " reply: " ;
      otherPerson->farewell();
      cout << endl;
     }
    }
    counter++;
   } while (counter<maximumOccupancy);
  }
}
person02.hpp
#ifndef _PERSON02
#define _PERSON02

class Person
{
  private:
   int code;
```

```
    int weight;
   public:
    // Constructors
    Person();
    Person(int code, int weight);
    Person(const Person &);
    ~Person();

    // Access functions
    int getCode();
    void assignCode(int);
    int getWeight() const;

    void assignWeight(int nWeight);
    void assignPerson(int);
    void assignPerson(int,int);
    void requestData();
    void requestCode();

    void greet();
    void farewell();
};
#endif


person02.cpp
#include <iostream>
#include "person02.hpp"

Person::Person()
{ }
Person::Person(int nCode, int nWeight)
{
  code = nCode;
  weight = nWeight;
}

Person::~Person()
{ }

int Person::getWeight() const
{ return (weight); }

void Person::assignWeight(int nWeight)
{ weight = nWeight; }
int Person::getCode()
```

```
{ return (code); }

void Person::assignCode(int nCode)
{ code= nCode;}

void Person::assignPerson(int nCode)
{ this->code = nCode;}

void Person::assignPerson(int nCode, int nWeight)
{
  assignCode(nCode);
  assignWeight(nWeight);
}

void Person:: greet()
{ cout << "Hello \n" ; };

void Person:: farewell ()
{ cout << "Goodbye \n" ; };

void Person::requestCode()
{
  int nCode;
  cout << "Code: ";
  cin >> nCode;
  cout << endl;
  code = nCode;
}


exerc02.cpp
#include <iostream>
#include "lift02.hpp"
#include "person02.hpp"

void requestData(int *nCode, int *nWeight)
{
  cout << endl;
  cout << "Code: ";
  cin >> *nCode;
  cout << endl;
  cout << "Weight: ";
  cin >> *nWeight;
  cout << endl;
}

int main( int argc, char *argv[] )
{
  char opc;
  bool leave = false;
  Lift aLift;
  Person *onePerson;
  Person *locatePerson;

  do
  {
   cout << endl << "LIFT: ";
   cout << "[1] Enter [2] Exit [3] Status [0] End ";
   cin >> opc;
   switch (opc)
   {
    case '1': // option Enter
    {
     int nWeight;
     int nCode;

     requestData(&nCode, &nWeight);
     onePerson = new Person(nCode, nWeight);
     if (unLift.person_mayEnter(onePerson))
     {
      unLift.person_enter(onePerson);
      if (aLift.getOccupancy()>1)
      {
       cout << onePerson->getCode();
       cout << " says: " ;
       onePerson->greet();
       cout << endl; // Now the others reply
       unLift.person_greetRestLift(onePerson);
      }
     }
     break;
    }
    case '2': // option Leave
    {
     onePerson = NULL;
     locatePerson = new Person;
     locatePerson->getCode();
     if (unLift.person_select ( locatePerson,
                                          &onePerson))
     {
     unLift.person_leave(onePerson);
     if (aLift.getOccupancy()>0)
     {
      cout << onePerson->getCode()
      cout << " says: " ;
      onePerson->farewell();
      cout << endl; // Now the others reply
      unLift.person_farewellRestLift(onePerson);
      delete (onePerson);
     }
    }
    else
    {
     cout<<"There is no one with this code";
     cout << endl;
    }
    delete locatePerson;
    break;
   }
   case '3': //Status
   {
    unLift.showOccupation();
    cout << " - "; // To separate occupancy from load
    aLift.showLoad();
    cout << endl;
    aLift.showListPassengers();
    break;
   }
   case '0':
   {
    leave = true;
    break;
   }
  }
 } while (! leave);
 return 0;
}
```

3. `lift03.hpp` and `lift03.cpp` coincide with `lift02.hpp` and `lift02.cpp` of exercise 2.

```
person03.hpp
#ifndef _PERSON03
#define _PERSON03

class Person
{
  private:
   int code;
   int weight;
  public:
   // Constructors
   Person();
   Person(int code, int weight);
   Person(const Person &);
   ~Person();

   // Access functions
   int getCode();
   void assignCode(int);
   int getWeight() const;
   void assignWeight(int nWeight);
   void assignPerson(int,int);
   void requestCode();


   virtual void greet();
   virtual void farewell();
};

class Catalan: public Person
{
  public:
   Catalan()
   {
    assignCode (0);
    assignWeight (0);
   };

   Catalan(int nCode, int nWeight)
   {
    assignCode(nCode);
    assignWeight(nWeight);
    };

    virtual void greet()
    { cout << "Bon dia"; };

    virtual void farewell()
    { cout << "Adéu"; };
};

class Spanish: public Person
{
  public:
    Spanish()
    {
     assignCode (0);
     assignWeight (0);
    };

    Spanish(int nCode, int nWeight)
    {
     assignCode(nCode);
     assignWeight(nWeight);
    };

    virtual void greet()
    { cout << "Buenos días"; };

    virtual void farewell()
    { cout << "Adiós"; };
};

class English : public Person
{
  public:
    English()
    {
     assignCode (0);
     assignWeight (0);
    };

    English(int nCode, int nWeight)
```

```
   {
    assignCode(nCode);
    assignWeight(nWeight);
   };

   virtual void greet()
   { cout << "Hello"; };

   virtual void farewell()
   { cout << "Bye"; };
};
#endif
```

```
person03.cpp
#include <iostream>
#include "person03.hpp"

Person::Person()
{ }

Person::Person(int nCode, int nWeight)
 {
   code = nCode;
   weight = nWeight;
 }

 Person::~Person()
 { }

 int Person::getWeight() const
 { return (weight); }

 void Person::assignWeight(int nWeight)
 { weight = nWeight; }

 int Person::getCode()
 { return (code); }
```

```
void Person::assignCode(int nCode)
{ this->code = nCode;}

void Person::assignPerson(int nCode, int nWeight)
{
  assignCode(nCode);
  assignWeight(nWeight);
}

void Person:: greet()
{ cout << "Hello \n" ; };

void Person:: farewell ()
{ cout << "Goodbye \n" ; };

void Person::requestCode{
  int nCode;

  cout << "Code: ";
  cin >> nCode;
  cout << endl;

  assignCode(nCode);
}
```

```
         exerc03.cpp
#include <iostream>
#include "lift03.hpp"
#include "person03.hpp"
 void requestData(int *nCode, int *nWeight, int *nlanguage)
 {
   cout << endl;
   cout << "Code: ";
   cin >> *nCode;
   cout << endl;
   cout << "Weight: ";
   cin >> *nWeight;
   cout << endl;
```

```
  cout << "Language: [1] Catalan [2] Spanish [3] English ";
  cin >> *nlanguage;
  cout << endl;
}

int main( int argc, char *argv[] )
{
  char opc;
  bool leave = false;
  Lift aLift;
  Person *onePerson;
  Person *locatePerson;

do
{
  cout << endl << "LIFT: ";
  cout << "[1] Enter [2] Exit [3] Status [0] End ";
  cin >> opc;
  switch (opc)
  {
   case '1': // Option Enter
   {
    int nWeight;
    int nCode;
    int nlanguage;
    requestData(&nCode, &nWeight, &nlanguage);
    switch (nlanguage)
    {
     case 1:
     {
      onePerson = new Catalan(nCode, nWeight);
      break;
     }
     case 2:
     {
      unaPerson=new Spanish(nCode, nWeight);
      break;
      case 3:
      {
       onePerson = new English(nCode, nWeight);
  break;
 }
}
if (unLift.person_mayEnter(onePerson))
{
 unLift.person_enter(onePerson);
 if (aLift.getOccupancy()>1)
 {
  cout << onePerson->getCode();
  cout << " says: " ;
  onePerson->greet();
  cout << endl; // Now the others reply
  unLift.person_greetRestLift(onePerson);
 }
}
 break;
}
case '2': //Option Leave
{
 locatePerson = new Person;
 onePerson = NULL;

 locatePerson->getCode();
 if (unLift.person_select ( locatePerson,
   &onePerson))
 {
  unLift.person_leave(onePerson);
  if (aLift.getOccupancy()>0)
  {
   cout << onePerson->getCode();
   cout << " says: " ;
   onePerson->farewell();
   cout << endl; // Now the others reply
   unLift.person_farewellRestLift(onePerson);
   delete (onePerson);
  }
 }
 else
 {
  cout<<"There is no one with this code";
  cout << endl;
```

```
    }
    delete locatePerson;
    break;
  }
  case '3': //Status
  {
   unLift.showOccupation();
   cout << " - "; // To separate Occupancy from Load
   aLift.showLoad();
   cout << endl;
   aLift.showListPassengers();
   break;
  }
  case '0':
  {
   leave = true;
   break;
  }
 }
} while (! leave);
return 0;
```

Programming in Java

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Josep Anton Pérez López
Lluís Ribas i Xirgo

PID_00148428

Index

Introduction

In the previous units we have looked at the evolution undergone by programming languages throughout history and how the various programming paradigms have developed.

In the beginning the main cost of a computer system was the hardware: The internal components of computers were voluminous, slow and expensive. In comparison, the costs generated by the people involved in maintaining them and in data processing were almost negligible. Also the applications which could be executed had to be simple due to the physical restrictions. The emphasis in IT research was basically focused on the creation of smaller systems which were faster and cheaper.

This situation has changed dramatically over time. The revolution taking place in the hardware world has allowed the manufacture of computers which could not even be dreamed of 25 years ago, this revolution has also taken place in the world of software. In this respect the costs of materials have reduced dramatically while those for personnel have increased dramatically. The complexity of software has also increased due to, among other things, the increase in interactivity with the user.

Many of the lines of research today are aimed at improving performance during the software development stage in which human involvement is still fundamental. Much of this effort is focused on the generation of correct code and the re-use of prior work.

The object oriented programming paradigm has created a much closer relationship between the software development process and the actual world which applications represent. The incorporation of computers into many of the objects which surround us has expanded the number of platforms on which programs can be developed.

Java is a modern language which was conceived to provide a solution for this new environment. It is basically an object oriented language which has been designed to work on multiple platforms. The concept consists of creating a common intermediate platform for which applications are developed, this will then translate the generated result for each end machine.

This intermediate step allows us to:

- Write each application **just once**. Once it has been compiled for the common platform the application can be executed by any system which has the intermediate platform installed.

- Writing the common platform **just once**. By making a real machine able to execute the instructions of this common platform, meaning that it is able to translate them for the real machine, it will be able to execute all applications developed for this platform.

We therefore get the maximum amount of re-use. The cost of this is sacrificing some of the speed.

In terms of the generation of correct code, Java has several attributes which we will look at over the course of this unit. For the moment we should note that Java is based on C++ which will make it easier for us to learn for a large number of developers (knowledge re-use), but it has also been freed from many of the chains which connected C++ to C.

This "Clean up" has had positive consequences:

- The language is simpler as rarely-used complex concepts have been removed.

- The language is more direct. It has been estimated that Java can reduce the number of lines required by a quarter.

- The language is purer as it only works with the object oriented programming paradigm.

The fact that the language is very recent has allowed several attributes to be included in its core which simply did not exist when other languages were created, some of these are the following:

- Programming using threads (*threads*) allows us to take advantage of multi-processor architecture.

- Communications programming (TCP/IP etc.), which facilitates networking both locally and over the Internet.

- The programming of *applets*, mini-applications designed to be executed by web browsers.

- Support for the creation of graphic user interfaces and an event management system which facilitates the creation of applications following the event-driven programming paradigm.

In this unit we want to introduce the reader to the new programming environment and to present the main characteristics, based on your knowledge of C++ we will be aiming to achieve the following objectives:

1) An understanding of the Java development environment.
2) Be able to program in Java.
3) An understanding of the concepts of the use of threads and their applications in the Java environment.
4) An understanding of the basics of event-driven programming and the ability to develop simple examples.
5) The ability to create simple *applets* .

1. The origin of Java

In 1991, engineers from Sun Microsystems were attempting to develop programs for white goods and small electronic equipment in which processing power and memory were reduced. This required a programming language which was easy to develop applications with and which was reliable and could be adapted to different kinds of electronic devices.

Note

Given the variety of devices and processors existing in the market and their continual modification, they were looking for a working environment which did not depend on the machine executing it.

To do this they designed a system based on an intermediate platform which would run a new executable machine code and this platform would then be in charge of translating it for the underlying device. This generic machine code would be oriented towards the mode of operation of most of these devices and processors meaning that the final translation had to be very fast.

The whole process would consist of writing a program in a high level language and compiling it to generate generic code (*bytecode*) which can then be executed by the platform (the "virtual machine"). This achieves the goal of having to write code just once and being able to execute it anywhere the platform is available (*Write Once, Run EveryWhere*).

They first tried to do this with C++ but its complexity caused many difficulties, this led them to design a new language which was based on C++ to make it easier to learn. This new language also included the attributes of modern languages and reduced its complexity removing those functions which were not absolutely essential.

The project for this new language was initially known as *Oak*, but as the name was already registered it was finally given the name Java. As a consequence, the virtual machine able to execute this code on any platform was given the name Java Virtual Machine (JVM - *Java virtual machine*).

The first attempts at marketing it were unsuccessful but the development of the Internet promoted multi-platform technologies and the company became interested in Java. After a series of design modifications Java was first presented to the world as a computer language in 1995 and in 1996 Sun founded the company Java Soft which would develop products for this new environment and facilitate collaboration with third parties. In the same month, the first rudimentary version of the Jave Development Kit was released: JDK 1.0.

The first version of Java appeared at the beginning of 1997 and it was version 1.1, which considerably increased the performance of the language. Java 1.2 came out in 1998 and incorporated significant changes. For this reason, this and later versions were known as Java 2 platforms. In December 2003 the last version of the Java 2 platform made available for download from the Sun website was Java 1.4.2.

Note

You can find this version at the following website:
http://java.sun.com

The true revolution which drove the expansion of the language was brought about by the incorporation of a Java interpreter into Netscape Navigator in 1997.

2. General attributes of Java

Sun Microsystems describes Java as a language which is simple, object oriented, distributed, robust, secure, with neutral architecture, portable, interpreted, high-performance, multi-tasking and dynamic.

Let us take a closer look at these descriptions:

- **Simple**. In order to make learning it easier, they considered that the most commonly used languages by programmers were C and C++.
  Following on from C++, they designed a new language which was very close in order to make it easier to understand.
  With this in mind, Java has done away with a set of attributes in C++ which were rarely used and difficult to understand, for example, multi-inheritance, automatic coercion and operator overloading.

- **Object oriented**. Briefly put, object oriented design focuses on the design of data (objects), their functions and their relationships (methods). This essentially follows the same criteria as C++.

- **Distributed**. Java includes a large library of routines which allow us to work more easily with TCP/IP, HTTP and FTP protocols for example. We can create network connections using URL addresses with the same ease as we can working in the local area.

- **Robust.** One of the aims of Java is to make programs more reliable. To do so, its creators focused on three main aspects:
  - Strict control of compilation over time with the aim of detecting problems as quickly as possible. To do this, it uses a strong type control strategy similar to the one used by C++, although avoiding some of the bottlenecks often caused by C compatibility. It also allows for type control during linking.

  - It checks for any dynamic errors during execution.

  - It eliminates situations which are likely to generate errors. The most significant of these is the management of pointers. It treats them as true vectors and controls the possible index values. As it does not use pointer arithmetic (summing displacement to a memory location without limit control) it prevents memory over-writing and data corruption.

- **Security**. Java is oriented towards network distributed environments and, therefore, a great deal of emphasis has been placed on security against viruses and intrusions as well as on authentication.

- **Neutral architecture**. In order to be able to operate on a variety of processors and operating system architectures, the Java compiler provides a commonly executable code from any system which has Java run time. This means that authors of applications do not need to produce different versions for different systems (such as PC, Apple Macintosh etc.). A Java compiled code will work on all of them.
  To do this Java creates *bytecode* instructions that are designed to be easily-interpreted by an intermediate platform (the Java virtual machine) and translated to the native code of the machine on the fly.

- **Portable**. Neutral architecture provides many advantages in terms of portability but it is not the only aspect which has been improved. Portability is also improved by the libraries. For example, it has a Windows abstract class and implementations for Windows, Unix and Macintosh.

Example

In Java, no aspects depend on the implementation, such as the size of primitive types, for example. In Java, unlike with C or C++, the `int` type refers to a 32 bit integer with a complement on 2 and the `float` type refers to a 32 bit number as per the IEEE 754 standard.

- **Interpreted**. No *bytecode* in Java is translated to instructions for the native machine during execution (it is interpreted) and these are not stored in any location.

- **High performance**. We will sometimes want to improve the performance of the interpretation of the *bytecode*, although it is already fairly good. In this case we can translate them to the native code of the machine executing the application during run time. This requires the compilation of the JVM language to the language of the machine on which the program is being executed.
  Similarly, *bytecode* have been designed with the machine code in mind, making the machine code generation final process very simple. What is more, the generation of *bytecode* is very efficient and several optimisation procedures are applied to them.

- **Multi-tasking**. The Java language also includes tools for constructing applications with multiple execution threads, which simplifies use and makes programs more robust

- **Dynamic**. Java is designed to adapt to changing environments. For example, the way code has been implemented in C++ causes a side effect. If a program uses a class library and this is subsequently changed, we will need to compile the whole program again and redistribute it. Java avoids this problem by inter-connecting modules later on and this means new methods and instances can be added without having any effect on the clients.

The interfaces can be used to specify a set of methods that an object can perform, but the way in which objects can implement these methods is left open. A Java class can implement multiple interfaces, although it can only inherit from a single class. Interfaces provide flexibility and re-usability by connecting objects according to what we want them to do and not by what they do.

Classes in Java are represented by a class called Class during run time, this contains the run time class definitions. As such, it can make type comparisons during run time meaning we can trust the types in Java, whereas in C++ the compiler has to trust that the programmer has done this correctly.

3. The Java development environment

There are several options available on the market for developing Java programs. However, the Java Development Kit (JDK) is freely distributed by Sun and this is a set of programs and libraries that allow the development, compilation and executions of programs in Java. Besides, a *debugger* is also included for error detection.

It also includes tools which allows all the previous components (IDE - *integrated development environment*) to be incorporated and making life easier, although these may have compatibility issues between platforms or between resulting files which have not been optimised. For this reason and, in order to familiarise ourselves with all software creation processes, we have chosen to use the Sun tool set to develop all the applications included in this book.

Note

Among the IDE's currently available it is worth looking at the Eclipse project, which follows the open code philosophy and is a very complete development package (SDK - *standard development kit*) for most operating systems (Linux, Windows; Sun, Apple etc.).

This package can be downloaded at http://www.eclipse.org.

Another interesting IDE is JCreator, which, apart from developing a commercial version, it also comes in a light version which is very easy to handle.

This package can be downloaded at http://www.jcreator.com.

Another interesting aspect of Java is that several different types of applications can be created:

- **Independent applications**. A file directly executed on the virtual machine of the platform.

- *Applets*. These are mini applications that can not be executed directly on the virtual machine as they are designed to be loaded and executed by a web browser. For this reason they have very strict security limitations.

- *Servlets*. Applications with no user interface which are run on a server and which are designed to respond to remote navigators (HTML page requests, sending form data etc.) They usually output files such as HTML files for example.

All we need to generate any of the above applications is the following:

- A text editor to write the Java source code.

- The Java platform which allows you to compile, debug, run and document your programs.

3.1. The Java platform

A platform is a hardware or software environment that a program needs to be able to run. Although most platforms can be described as a combination of the operating system and hardware, the Java platform is different from others in that it is a software platform that operates on other hardware-based platforms (GNU/Linux, Solaris, Windows, Macintosh etc.).

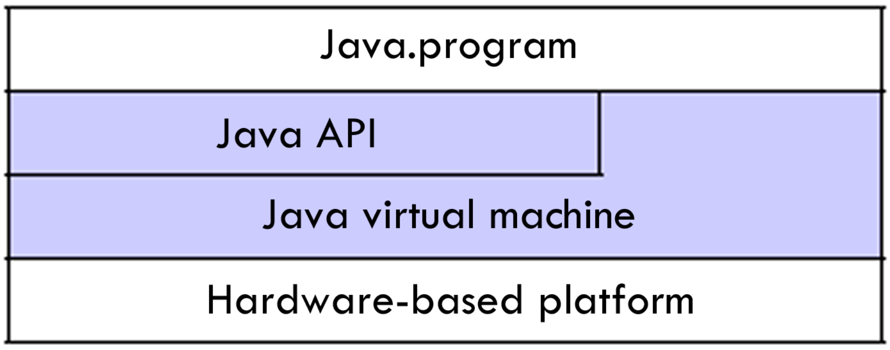
The Java platform has two components:

- Virtual Machine (VM). As we have already mentioned, one of the main characteristics of Java is that it is independent of the hardware platform: Once they have been compiled, programs can run on any platform.

The strategy used to achieve this is the generation of neutral code (*bytecode*) as a result of the compilation. Neutral code is very similar to machine code and can be executed from a "hypothetical machine" or "virtual machine". To be able to execute a program on any given platform, we will just need the "virtual machine" for that platform.

- *Application programming interface* (API). The Java API is a large collection of pre-written software which provides many capabilities such as graphics environments, communications, multi-processing etc. It is organised in libraries of related classes and interfaces. These libraries are known as *packages*.

The following diagram shows the structure of the Java platform and how the virtual machine isolates the source code (.java) from the hardware of the machine:



3.2. My first program in Java.

Once again, our first experience with the language will be to say hello to the world. We will divide this into three stages:

1) The creation of a source file. We will use our chosen text editor to write the code and save it under the name *HelloWorld.java*.

```
HelloWorld.java
/**
  * The HelloWorld class shows the message
  * "Hello World" on the standard output.
  */
public class HelloWorld {
    public static void main(String[] args) {
     // Shows "Hello World!"
     System.out.println("Hello World!");
    }
}
```

2) Compilation of the program generating a *bytecode*. For this we will use the **javac** compiler, which provides the development environment and translates the source code to instructions that the JVM can interpret.

After typing `"javac HelloWorld.java"` in the command interpreter, as long as there are no errors we will have our first Java program: a file called *HelloWorld.class*.

3) Executing the program in the Java virtual machine. Once the *bytecode* file has been generated, we can run it in the JVM by typing in the following instruction so that it can be interpreted by the computer and then the message "Hello World!" will appear on the screen.

```
java HelloWorld
```

3.3. Basic instructions and comments

In this respect, Java remains faithful to C and C++ and keeps the same syntax.

The only consideration to bear in mind is that, in Java, conditional expressions (for example the `if` condition) must return a value of the *boolean* type, whereas C++, in order to retain compatibility with C, allows numerical values to be returned and equates 0 to *false* and any other value to *true*.

As for comments, Java allows those formats from C++ (/* ... */ and // ...) and it also includes a new one: you can include text between the symbols /** (start of comment) and */ (end of comment).

In fact, this format is not so much used for commenting as for documentation. Java provides tools (for example, *javadoc*) to generate documentation from source codes that extract the comments made in the program in the following way.

Example

```
/**
* Comment text using the new Java format for
* inclusion in automatically created documentation.
*/
```

4. The differences between C++ and Java

As we have already mentioned, Java is based on C++ and provides an object oriented environment that will be very familiar to most programmers. However, Java has been designed to improve C++ in many respects, above all in those which allowed C++ to work in a "non-object oriented way" and which were included to make it more compatible with C.

4.1. Input/output

As Java was mainly designed to work with graphics, the classes which manage text input/output are very basic. They are governed by the System class which can be found in the java.lang library, there are three static objects in this class which are as follows:

- **`System.in`**. This receives data from the standard input (normally the keyboard) in an object of the InputStream class.

- **`System.out`**. This displays the data on the standard output (usually the screen), an object of the OutputStream class.

- **`System.err`**. Displays error messages on the screen.

The basic methods these objects have are the following:

- **`System.in.read()`**. Reads a character and returns it as an integer.

- **`System.out.print(var)`**. Prints a variable of any primitive type.

- **`System.out.println(var)`**. The same as the last one but with the addition of a line feed at the end.

Therefore, in order to write a message we only really need to use the instructions `System.out.print()` and `System.out.println()`:

```
int anInteger = 35;
double aDouble = 3.1415;

System.out.println("Displaying a text");
System.out.print("Displaying an integer ");
System.out.println (anInteger);
System.out.print("Displaying a double ");
System.out.println (aDouble);
```

While outputting data is fairly easy, data input is far less accessible as the basic read element is a character. We will now look at an example that demonstrates the process for reading a character string:

```
String myVar;
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// Input is ended when pressing Enter
myVar = br.readLine();
```

Reading complete lines can be done using the `BufferedReader` object, whose `readLine()` method calls a character reader (a `Reader` object) until it finds the end of line symbol ("\n" or "\r"). But in this case the input stream is an `InputStream` object and not a `Reader`. We will therefore need a class that acts as a reader for an `InputStream`. This will be the class `InputStreamReader`.

However, the above example is valid for *Strings*. When we want to read an integer or other data types, they must be converted after reading. However, this conversion can generate a fatal error in the system if the entered text does not coincide with the expected type. In this case, Java always obliges us to use error checks. Error management (which generates exception calls) is done in the same way as in C++ using the sentence `try {... } catch {...} finally {...}`.

We will now look at how we can design a class to return an integer read from the keyboard:

```
Read.java
import java.io.*;
public class Read
{
  public static String getString()
  {
   String str = "";
   try
   {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    str = br.readLine();
   }
   catch(IOException e)
   {
    System.err.println("Error: " + e.getMessage());
   }
   return str; // returns the typed data
  }
```

```
    public static int getInt()
     {
      try
      {
       return Integer.parseInt(getString());
      }
      catch(NumberFormatException e)
      {
       return Integer.MIN_VALUE; // smallest value
      }
    }
  // getInt
    // we can define a function for each type...
    public static double getDouble() {} // getDouble

  }
    // Read
```

The `try { ... }` block includes the piece of code that may be susceptible to an error. If one occurs, an exception is launched and caught by the block `catch { ... }`.

When converting `String` types to numbers, the exception which may be produced is of the type `NumberFormatException`. There could be more `catch` blocks to deal with different types of exceptions. In the example, if an error occurs, the numerical value returned will be the minimum possible value an integer can take.

The `finally { ... }` block is a piece of code which is executed whether an error occurs or not (for example, close files), although its use is optional.

We can develop functions for each of the primitive types in Java in a similar way. Lastly, it would read an integer number in the following way:

```
  int i;
  ...
  i = Read.getInt( );
```

4.2. The preprocessor

Java does not have a preprocessor, so several orders (generally originating in C) are not included. The most well-known of these are the following:

- `defines`. These orders were used to define constants and in C++ they had already become fairly redundant as you could declare `const`variables, and they are now implemented using `final`.

- `include`. This order, which is used to include the content of a file, was very useful in C++, mainly for the re-use of header files. In Java there are no header files and the libraries (or packages) are included using the instruction `import`.

4.3. The declaration of variables and constants

The declaration of variables stays the same but the method of defining constants has changed: in Java the variable is preceded by the reserved word `final`; we do not need to assign it a value at the time of declaration. However, once we have assigned it a value it cannot be changed.

```
final int i;
int j = 2;
...
i = j + 2; // once it has been assigned a value it can not be changed
```

4.4. Data types

Java classifies data types into two categories: Primitive and reference. The first of these contains a value while the second contains the memory address where the information is stored.

The primitive types of data (`byte, short, int, long, float, double, char` and `boolean`) are basically the same as in C++, although there have been a couple of changes which we will look at now:

- Numerical types have the same size no matter what platform they are executed on.

- Numerical types may not take the specifier `unsigned`.

- The `char` type uses the Unicode character set, which is 16 bit. Characters 0 to 127 correspond to those of the ASCII code.

- If variables are not initialised explicitly, Java automatically initialises the data to zero (or its equivalent) thus removing any junk values they may contain.

The reference types in Java are vectors, classes and interfaces. Variables of this type store their memory address and are therefore similar to pointers in other languages. However, as explicit operations are not allowed with memory addresses, to access them we only need to use the name of the variable.

Java has also done away with the `struct` and `union` types, which can now be implemented using `class` and which were kept in C++ for compatibility with C. It also gets rid of the `enum` type, although this can be emulated using numeric constants with the `final`.

It has also done away with `typedefs` for the definition of types, which in C++ had already become fairly redundant as the classes Structs, Union and Enum had become types.

Lastly, only automatic type coercions (*typecasting*) are permitted for secure conversions, meaning conversions in which there is no risk of data loss. For example, it does allow automatic conversion of the `int` type to the `float` type, but not in the reverse direction where the decimals would be lost. If data may get lost, we need to tell it explicitly that we want to perform a type conversion.

Another notable aspect of Java is the way it implements vectors. It treats them as real objects and generates an exception (error) when its limits are exceeded. They also have a member called `length` to indicate their length, which provides added safety by preventing undesired memory access.

Java has two types for working with character strings: `String` and `StringBuffer`. Strings defined between double inverted commas are automatically converted to `String` objects and they cannot be modified. The `StringBuffer` type is similar, but it allows the modification of its value and provides methods for manipulation.

4.5. The management of dynamic variables

As we mentioned in our explanation of C++, direct memory management is a very powerful tool, but it can also be dangerous: any errors in management can cause very serious problems for the application and even, maybe, for the system.

Pointers were present in C and C++ due to the use of strings and vectors. Java provides objects both for strings and vectors, so pointers are no longer necessary in these cases. The other big requirement, that of passing parameters by variable, is covered by the use of references.

As the issue of security is given a high priority in Java, the developers decided not to use pointers, at least not in the way they are used in C and C++.

C++ has two ways of working with pointers:

- Either with their address, even allowing mathematical operations to be performed on them (pointer).

- Or with their content (* pointer).

Java does away with all operations on memory addresses. When we talk about *references* , this has a slightly different meaning than in C++. A dynamic variable is in fact a reference to the object (pointer):

- To see the content of the variable we only need to use the form (pointer).

- To create a new element we can still use the operator `new`.

- If we assign a reference type variable (an object for example) to another variable of the same type (an object of the same class) the content is not duplicated, but the first variable points to the same position of the second variable. The end result being that both have the same content.

Java does not allow us to operate directly on memory addresses, making it simpler to access the content: this is done using the name of the variable (instead of using the dereferenced form *variable_name).

Another of the main risks involved with direct memory management is associated with correctly freeing the memory occupied by dynamic variables once they are no longer being used. Java resolves this problem by providing a tool which automatically frees this space when it detects that a variable will not be used again. This tool is known as the *garbage collector* (and it is part of Java while its programs are running. We do not therefore need to use the `delete` instruction, we only need to assign the pointer to `null` and the garbage collector detects that the memory area is no longer in use and frees it.

If we want to, instead of waiting for the garbage to be collected automatically, we can invoke the process using the function `gc()`. However, this call is only considered to be a suggestion for the JVM.

4.6. Functions and the passing of parameters

As we already know Java only uses object oriented programming. Therefore, it does not allow independent functions (they must always be included in classes) or global functions. What is more, methods must be implemented within the class definition. This also removes the need for header files. The compiler itself detects if a class has already been loaded to prevent duplication. Despite its similarity with the `inline` functions, it is only formal because it behaves differently internally: Java does not implement `inline`.

On the other hand, Java still supports function overloading, although it does not allow the programmer to overload operators despite the fact that the compiler uses this attribute internally.

In Java all parameters are passed by value.

In the case of primitive data types, the methods always receive a copy of the original value that cannot be modified.

With reference data types, the value of the reference is also copied. However, due to the nature of references, the changes made to the variable receiving the parameter also affect the original variable.

To modify variables passed to the function by parameter we need to include them as member variables of the class and pass the reference to an object in this class as an argument.

5. Classes in Java

As we have already mentioned, one of the motivations for the creation of Java was to produce a "pure" object oriented language which always complies with this programming paradigm. This was not true with C++ to allow it be compatible with C. Classes are therefore the fundamental components of Java: everything must be included in a class. Classes are defined in a similar way to C++, although there are some differences:

```
Point2D.java
class Point2D
{
  int x, y;

// initialising the coordinate origin
   Point2D()
   {
    x = 0;
    y = 0;
   }

// initialising a specific x,y coordinate
   Point2D(int coordx, int coordy)
   {
    x = coordx;
    y = coordy;
   }

   // calculating the distance to another point
   float distance(Point2D npoint)
   {
    int dx = x - npoint.x;
    int dy = y - npoint.y;
    return ( Math.sqrt(dx * dx + dy * dy));
   }
}
```

- The first difference is the inclusion of the definition of the method inside the class and not separately, as in C++. Following this criteria we no longer need the scope operator (::).

- The second difference is that we do not need the semi-colon (;) at the end.

- Classes are stored in a file with the same name and with the extension .java (Point.java).

One attribute Java has in common with C and C++ is that it is also sensitive to capital letters, meaning that the class Point2D is not the same as `point2d` or `pOiNt2d`.

Java allows more than one class to be stored in a file but only one of these may be public. This one must have the same name as the file. We will therefore usually use a different file for each class, with rare exceptions.

Similarly to C++, the attributes (or member variables) and methods (or member functions) are declared in the class definition, as we can see from the above example.

5.1. The declaration of objects

Once a class has been defined, to declare an object in that class we only need to put the name of the class (like with a type) before the name of the object.

```
Point2D    pointOne;
```

The result is that `pointOne` is a reference to an object in the class `Point2D`. This reference initially has the value `null` and no memory space will have been reserved for it. To be able to use this variable to store information we need to create an instance using the operator `new`. When this is used, it will call the constructor of the `Point2D` object that has been defined.

```
pointOne = new Point2D(2,2); // initialised to (2,2)
```

One important difference of Java with respect to C++ is the use of references to manipulate objects. As we have mentioned before, the assignment of two variables declared as objects only implies the assignment of its reference:

```
Point2D    pointTwo;
pointTwo = pointOne;
```

By adding the above instruction we have not reserved any memory for the reference to the object `pointTwo`. When making the assignment, `pointTwo` will reference the same object pointed to by `pointOne`, and not to a copy. Therefore, any changes made to the attributes of `pointOne` will be reflected in `pointTwo`.

5.2. Accessing objects

Once an object has been created, any of its attributes and methods can be accessed using the point (.) operator as in C++.

```
int i;
float dist;

i = pointOne.x;
dist = pointOne.distance(5,1);
```

In C++ we could access the object by dereferencing a pointer to that object (`*pointer`), in which case, accessing the attributes or methods could be done using the point operator (`*pointer.attribute`) or by using the abbreviated access operator → (`pointer→attribute`). In Java, as the dereferenced form *pointer does not exist, neither does the → operator.

Lastly, as with C++, Java allows objects within methods of the class to be accessed using the object `this`.

5.3. The destruction of objects

When we have created an object it must be destroyed when it is no longer in use. The way memory management functions in Java avoids many of the conflicts that appear in other languages and it allows us to delegate this responsibility to an automatic process: the garbage collector that detects when an area of memory is no longer referenced and will free it at a time when the system is not under pressure.

When working with a class we will sometimes be using additional resources such as files. Once the activity of a class has finished, we can often close the activity of these additional resources. In these cases we need to perform a manual procedure similar to that of destructors in C++. To do this, we can implement a method called `finalise()` which is also called by the garbage collector itself, if it is present. Inside this method we can write the code which explicitly frees the additional resources we used. The method `finalise` is always of the type `static void`.

```
class MyClass
{
  MyClass() //constructor
  {
   ...      //Initialisation instructions
  }

  static void finalise() //destructor
  {
   ... //instructions to free up resources
  }
}
```

5.4. Copy constructors

C++ has copy constructors to make sure that a complete copy is made of the data when an assignment is made or when we assign a parameter or the return value of a function.

Java uses a different philosophy, as we have already seen. Assignments between objects do not involve copying their content, the second reference will reference the first object. It therefore always accesses the same content and no additional memory reservation operation is required. Java does not need copy constructors as a consequence of this.

5.5. Simple inheritance and multiple inheritance

In Java, to indicate that one class is derived from another (meaning it inherits some or all of its attributes and methods) we use the term `extends`. We will look at the example using dogs and mammals again.

```
 class Mammal
 {
   int age;

   Mammal()
   { age = 0; }

   void assignAge(int nAge)
   { age = nAge; }

   int getAge()
   { return (age); }

   void makeSound()
   { System.out.println("Sound "); }
 }

 class Dog extends Mammal
 {
   void makeSound()
   { System.out.println("Woof "); }
 }
```

In the above example we are stating that the Dog class has been derived from the Mammal class. We could also read it in reverse and say that the Mammal class is the superclass of the Dog class.

In C++ we could use multiple inheritance meaning that methods and attributes could be received from several classes. This is not possible in Java, although interfaces provide a similar functionality.

6. Inheritance and polymorphism

Inheritance and polymorphism are essential properties of the object oriented design paradigm. We have already looked at these concepts in the unit on C++ and they have been maintained in Java. However, their implementation is slightly different in Java and we will look at this now.

6.1. The references this and super

Sometimes we will want to access the attributes or methods of the object serving as the base for the object they are in. As we have already seen, both C++ and Java provide access using the reference `this`.

What is new in Java is that we can also access the attributes and methods of the object of the superclass using the reference `super`.

6.2. The Object class

Another difference between C++ and Java is that all of the objects belong to the same hierarchical tree, whose root is the Object class which all the others inherit from: if the definition of a class does not have the term `Extends`, it will be assumed that it inherits directly from Object.

We can say that the Object class is the superclass from which all other classes in Java are directly or indirectly derived.

The Object class provides a series of common methods, among which are the following:

- `public boolean equals` (`Object` obj). This is used to compare the content of two objects, it returns *true* if the received object is the same as the object calling it. If we only want to compare two references to an object we can use the comparison operators == and `!=`.

- `protected Object Clone`. Returns a copy of the object.

6.3. Polymorphism

C++ implemented the capability for a variable to be able to take several forms using object pointers. As we have already mentioned, Java does not use pointers and covers this function through the use of references, but the functionality is similar.

```
Mammal mammalOne = new Dog;
Mammal mammalTwo = new Mammal;
```

We must remember that in Java the declaration of an object is always a reference to it.

6.4. Abstract classes and methods

In C++ we saw that classes sometimes correspond to theoretical elements for which there is no point instancing objects and for which we always have to create objects of their derived classes.

This was implemented in C++ using pure virtual functions represented in a slightly peculiar way: they were declared assigning the virtual function to 0.

The implementation in Java is far simpler: we put the reserved word `abstract` before the name of the function. On declaring a function to be `abstract`, we are indicating that the class is abstract as well. However, we recommend that this is also specified in the declaration by putting the word `abstract` in front of the reserved word `class`.

The act of defining a function as being abstract means that all derived classes which can receive this method will redefine it. If they do not, they will inherit the function as being abstract and will also be abstract as a consequence, which will prevent us from instancing objects for these classes.

```
abstract class WorkOfArt
{
  String author;

  WorkOfArt(){} //constructor

  abstract void showWorkOfArt(); //abstract
  void assignAuthor(String nAuthor)
   { author = nAuthor; }
  String getAuthor();
  { return (author); }
  };
```

In the example above we have declared the function `showWorkOfArt()`to be abstract, meaning it will need to be redefined in derived classes. It is not therefore defined. We should also note that as it is an abstract class we will not be able to perform `new WorkOfArt`.

6.5. Final classes and methods

We have already looked at the concept of final variables. We have seen how final variables can not be modified once they have been initialised. This concept can also be applied to classes and methods:

- Final classes do not have, nor can they have, derived classes.
- Final methods can not be redefined in derived classes.

The use of the reserved word `final` becomes an extra safety measure to prevent the incorrect or malicious use of the properties of inheritance that could supplant established functions.

6.6. Interfaces

An interface is a collection of method definitions (without their implementations) whose function is to define a behaviour protocol that can be implemented by any class independently of its position in the hierarchy of classes.

When we indicate that an implemented class is an interface, it is obliged to redefine all of the defined methods. In this respect, interfaces are similar to abstract classes. However, while a class can only inherit from a superclass (only simple inheritance is allowed), it can implement several interfaces. This only means that it must comply with each of the protocols defined in a class.

We will now look at an example of declaring an interface:

```
public interface NameInterface extends SuperInterface1,
SuperInterface2
   { body interface }
```

If an interface has not been specified as public, it will only be accessible to the classes defined in the same package.

The body of the interface contains the declarations for all the methods included in it. Each declaration ends in a semi-colon (;) as they have no implementations and they are implicitly considered to be `public` and `abstract`.

The body can also contain constants, in which case they are considered to be `public`, `static` and `final`.

To indicate that a class implements an `interface`, we only need to add the keyword `implements` in its declaration. Java allows multiple inheritance between interfaces:

```
class MyClass extends SuperClass implements Interface1,
interface2
{ ... }
```

When a class declares an interface it is as if it were signing a contract through which it commits to implementing the methods of the interface and its superinterfaces. The only way for it not to do this is to declare the class as `abstract`, meaning it can not instance objects and this obligation is transferred to its derived classes.

At first sight there seem to be many similarities between abstract classes and interfaces, but there are also significant differences:

- An interface can not implement methods while abstract classes can.
- A class can have several interfaces but only one superclass.
- Interfaces are not part of the class hierarchy meaning that non-related classes can be implemented in the same interface.

Another relevant aspect of interfaces is that when we define them we are declaring a new type of reference data. A variable with this data type can be instanced by any class which implements this interface. This provides another way of applying polymorphism.

6.7. Packages

Java uses packages to organise classes. A package is a collection of related classes and interfaces that provide access protection and manage the name space. Classes and interfaces must always belong to a package.

Note

The classes and interfaces that are a part of the Java platform belong to several packages depending on their function: `java.lang` includes the fundamental classes, `java.io` contains the input/output classes etc.

The organisation of classes into packages largely prevents conflicts in name choices.

In order to define a class or an interface in a package, we just need to include the following expression in the first line of the archive:

```
package myPackage;
```

If no package is defined, it will be included in the default package (`default package`), this solution is fine for small applications or when we are starting to work with Java.

Accessing the name of the class can be done using the long name:

```
myPackage.MyClass
```

Another possibility is to import public classes from the package using the keyword `import`. We can then use the name of the class or interface in the program without prefixing it:

```
import myPackage.MyClass; //only imports the class
import myPackage.*        // only imports the package
```

We must bear in mind that importing a package does not imply the importation of the various sub-packages it may contain.

Example

Importing `java.awt` will not include the sub-package `java.awt.event`.

It is a convention that Java by default always imports from the package `java.lang`.

To organise the classes and the possible packages, a sub-directory is created for each package where the various classes in the package are included. Each package can also contain sub-packages that will be found in a sub-directory. By organising the files and directories in this way, both the compiler and the interpreter have an automatic mechanism for locating the classes that other applications need.

Example

The class `graphics.figures.rectangle` would be found in the package `graphics.figures` and the file would be located in graphics\figures\rectangle. java

6.8. The API (*applications programming interface*) for Java

The multitude of function libraries provided by the language is one of the fundamental aspects of Java; these libraries are standard, well documented, and they also work on all platforms.

This set of libraries is organised into packages and is included in the Java API. The main classes are as follows:

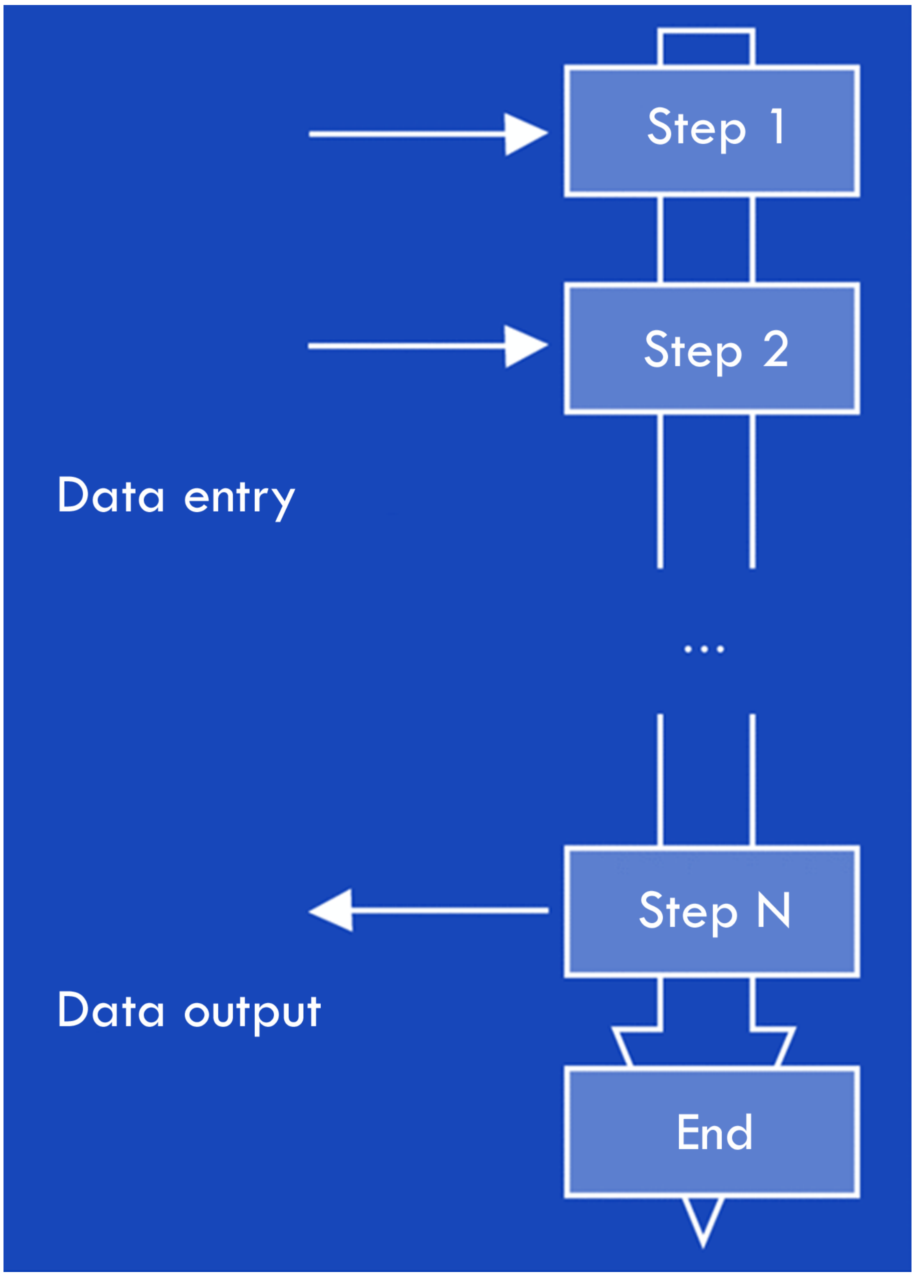
Table 9.

Package	Classes incorporated
java.lang	Fundamental classes for the language such as the String class and others.
java.io	Input and output classes using data streams, and system files.
java.util	Utility classes such as data and class collections, the event model, time utilities, random number generation and others.
java.math	A class containing all mathematical functions.
java.applet	A class of utilities to create *applets* and classes used by *applets* to communicate with their context.
java.awt	Classes that allow the creation of graphic interfaces for users, to draw images and create graphics.
javax.swing	Classes containing graphics components that work on all Java platforms.
java.security	Classes responsible for security in Java (encryption etc.).
java.net	Classes with functions for network applications.
java.sql	A class that incorporates JDBC for connecting Java to databases.

7. The event driven programming paradigm

All the programming paradigms we have looked at up till now are similar in that they use a sequential instruction flow and they use data for the development of the application. In order to function they usually require a beginning, a sequence of actions and an end.

Figure 15.



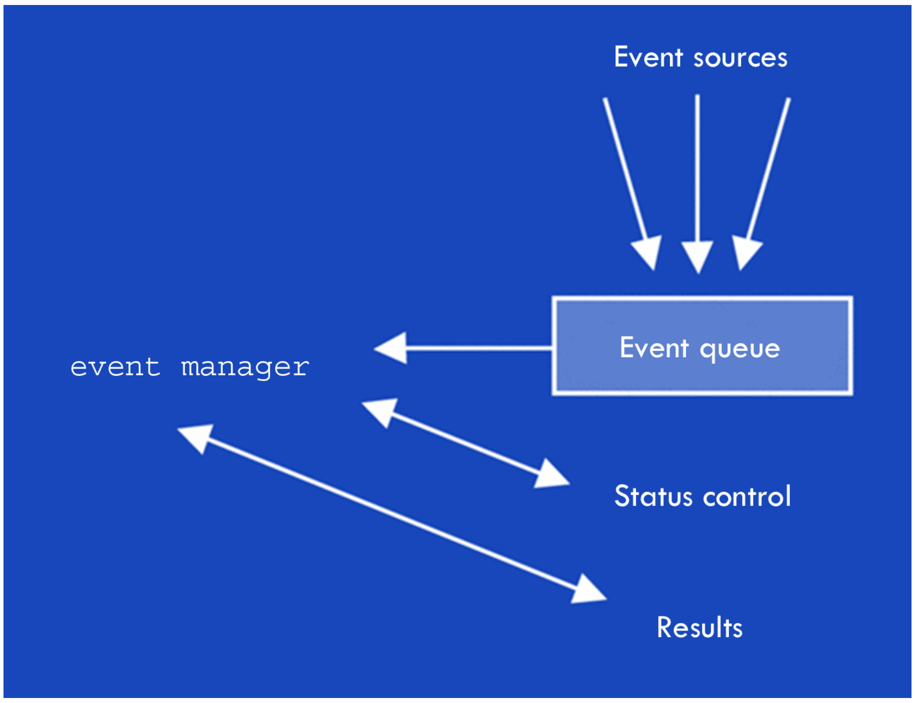
Program flow in imperative programming

Within this sequential operation the process also receives external events which may be expected (data input by the user using the keyboard, the mouse and other methods, the reading of system information etc.) or unexpected (system errors etc.). Each of these external events is called an ***event***.

In the previous paradigms, events did not alter the projected flow of instructions: events are used to resolve the instructions and if this is not possible, the program ends.

In the event-driven programming paradigm there is no single sequence of actions, it prepares reactions to events that may occur on the execution of the program. Therefore, in this model it is the data entered which regulates the sequence of the application. We can also see that applications will need to be designed differently with respect to previous paradigms: they will be able to remain active for an indefinite time while they receive and manage events.

Figure 16.



Flow chart for an event driven program

7.1. Events in Java

To manage events Java uses the **delegation event model**. In this model, a component receives an event and transfers it to the event manager assigned to manage it (*event listener*). We therefore have a separation of the code between the generation of the event and its treatment, thus making programming easier.

There are four type of elements involved:

- The event (what is received). In most cases the operating system generates the event and manages all the communications operations with the user and the environment. It will be stored in an object derived from the Event class and the type of object will depend on the type of event that has occurred. The main ones are related to the graphics environment: `ActionEvent, KeyEvent, MouseEvent, AdjustmentEvent, WindowEvent, TextEvent, ItemEvent, FocusEvent, ComponentEvent, ContainerEvent`.

Each of these classes has its own attributes and access methods.

- The source of the event (where it is received). This is the element that has generated the event and that therefore collects the information either to treat it or, in our case, to transfer it to the event manager. In graphics environments this will usually be the element the user has interacted with (a button, a text box etc.).

- The event manager (who manages it). This is a specialised class that indicates the desired response for each event. Each manager can respond to different types of events by assigning to it the appropriate profiles.

- The manager profile (what operations should be implemented by the manager). To make this task easier, there are interfaces that indicate the methods to be implemented for each type of event. Usually, the name of this interface takes the form <nameEvent>Listener.

Example

`KeyListener` is the interface for keyboard events and includes the three following methods: `keyPressed`, `keyReleased` and `keyTyped`. In some cases the fact that all three must be implemented can create an unnecessary load. To deal with this, Java provides the <nameEvent>Adapter which implements the various void methods allowing us to redefine only those methods we are interested in.

The main profiles (or interfaces) defined by Java are the following: `ActionListener, KeyListener, MouseListener, WindowListener, TextListener, ItemListener, FocusListener, AdjustmentListener, ComponentListener` and `ContainerListener`. They are all derived from the interface `EventListener`.

Lastly, we need to establish the relationship between the source of the event and its manager. To do this, we have to add a method of the following type in the source class: `add<nameEvent>Listener`.

Example

If we want to add a *Listener* to a button object in the Button class for mouse events we have to type: `button.addMouseListener(eventManager)`.

Actually, it could be considered that the events are not really sent to the event manager but that it is the event manager that is assigned to the event.

Note

We will be able to understand the functionality of events more easily if we look at a practical example such as the creation of an *applet* using the Swing graphics library, which we will look at more closely in this unit.

8. Execution threads

Current operating systems allow for multi-tasking, or at least the appearance of it when the computer only has a single processor as it will only be able to perform one operation at a time. However, we can organise the functionality of this processor so that it shares its time between several activities or it uses free time in one operation to perform another.

Each of these activities is called a *process*. A process is a program that is executed independently and with its own memory space. Multi-tasking operating systems thus allow several processes to run at the same time.

Each of these processes can have one or more execution threads, each of which corresponds to a sequential flow of instructions. In this case, all of the execution threads share the same memory space and use the same context and the same resources assigned to the process.

Java incorporates the ability for a process to have multiple simultaneous execution threads. A full explanation of their implementation in Java goes beyond the scope of this course and we will therefore only look at the basics for the creation of threads and their life cycle.

8.1. The creation of execution threads

In Java there are two ways of creating execution threads:

- The creation of a new class which inherits from `java.lang.Thread` and overloading the `run()` method for this class.

- Creating a new class using the `java.lang.Runnable` interface where the `run()`method will be implemented, and then creating a *Thread* object to which an object of the new class will be passed as an argument.

Wherever possible, we should use the first of these as it is far simpler. However, if the class already inherits from some other superclass, we will not be able to derive others from the *Thread* class (Java does not allow multiple inheritance), meaning we will have to use the second method.

We will look at examples of each way of creating execution threads:

The creation of execution threads deriving from the class *Thread*

```
TestThread.java
```

```
class TestThread
{
  public static void main(String args[] )
  {
  AThread a = new AThread();
  BThread b = new BThread();
  a.start();
  b.start();
  }
}

class AThread extends Thread
{
  public void run()
  {
  int i;
  for (i=1;i<=10; i++)
   System.out.print(" A"+i);
  }
}

class BThread extends Thread
{
  public void run()
  {
   int i;
   for (i=1;i<=10; i++)
    System.out.print(" B"+i);
  }
}
```

In the above example, two new classes derived from the class *Thread*: the classes `AThread` and `BThread`. Each of these shows a counter on the screen that is needed for the initiation of the process.

In the class `TestThreads`, where we have the method `main()`, we proceed with the instancing of an object for each of the *Thread* classes and execute them. The final result will be of the following type (though not necessarily in this order):

A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 B7 A8 B8 A9 B9 A10 B10

Lastly, we just need to remember that `TestThreads` will execute 3 threads: the main one and the two created ones.

The creation of execution threads using the interface `Runnable`

```
Test2Thread.java

class Test2Thread
{
  public static void main(String args[] )
  {
   AThread a = new AThread();
   BThread b = new BThread();
   a.start();
   b.start();
  }
}

class AThread implements Runnable
{
  Thread t;
  public void start()
  {
   t = new Thread(this);
   t.start();
  }

  public void run()
  {
   int i;
   for (i=1;i<=50; i++)
    System.out.print(" A"+i);
  }
}

class BThread implements Runnable
{
  Thread t;

  public void start()
  {
   t = new Thread(this);
   t.start();
  }

  public void run()
  {
   int i;
   for (i=1;i<=50; i++)
    System.out.print(" B"+i);
  }
}
```

In this example we can see that the principal class main() has not changed but it has implemented the classes *AThread* and *BThread*. In each of these, as well as implementing the `Runnable`interface, we need to define an object of the class `Thread` and redefine the method `start()` so that it calls the `start()` of the object of the class `Thread` passing to it the current object `this`.

We will finish with two points: it is possible to pass a name to each execution thread to identify it given that the class *Thread* has the constructor overloaded to allow this option:

```
public Thread (String name);
public Thread (Runnable destination, String name);
```

We can always recover the name using the method:

```
public final String getName();
```

8.2. The life cycle of execution threads

The life cycle of execution threads can be represented by the states they go through:

- New: the *thread* has been created but is not initialised, meaning it has not yet executed the method `start()`.

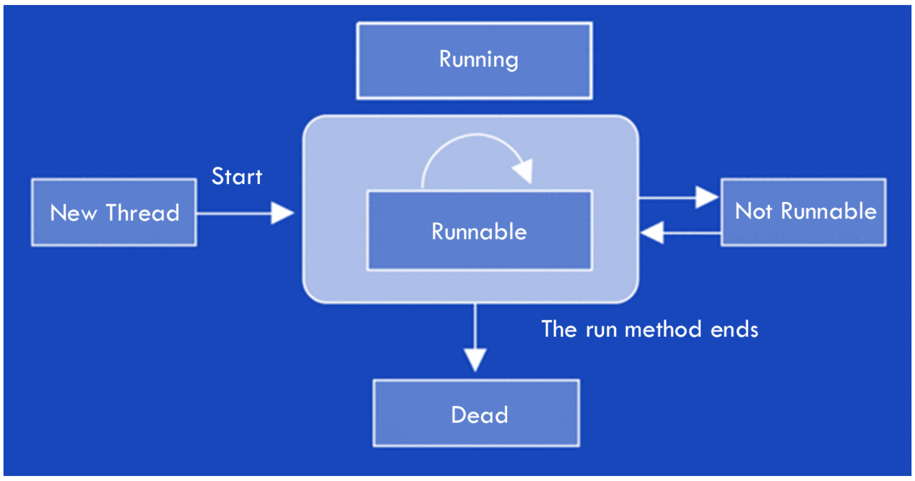
- Runnable: the *thread* is being executed or is ready to do so.

- Blocked (or *not runnable*): the *thread* is blocked by an internal message `sleep()`, `suspend()` or `wait()` , or by some other internal activity such as waiting for data to be entered. If it is in this state it will not be included in the list of tasks to be executed by the processor.

To return to the Runnable state it must receive the internal message `resume()` or `notify()` or it must end the situation that caused the blockage.

- Dead*:* the usual method for ending a *thread* is for it to have finished executing the instructions of the method `run()`. We could also use the method `stop()`, but this option is considered to be "dangerous" and it is not recommended.

Figure 17.


Running
Start
New Thread
Runnable
Not Runnable
The run method ends
Dead

9. *Applets*

An *applet* is a mini-application in Java designed to be executed on an Internet browser. To include an *applet* in an HTML page we only need to include the information using labels <APPLET> ... </APPLET>.

Most Internet browsers work in a graphics environment. Therefore *applet* must be adapted to them using graphic libraries. In this section we will use the `java.awt` library, which has been included in Java since the original versions. We will go further into the libraries available in Java later on in this unit.

The main attributes of *applets* are as follows:

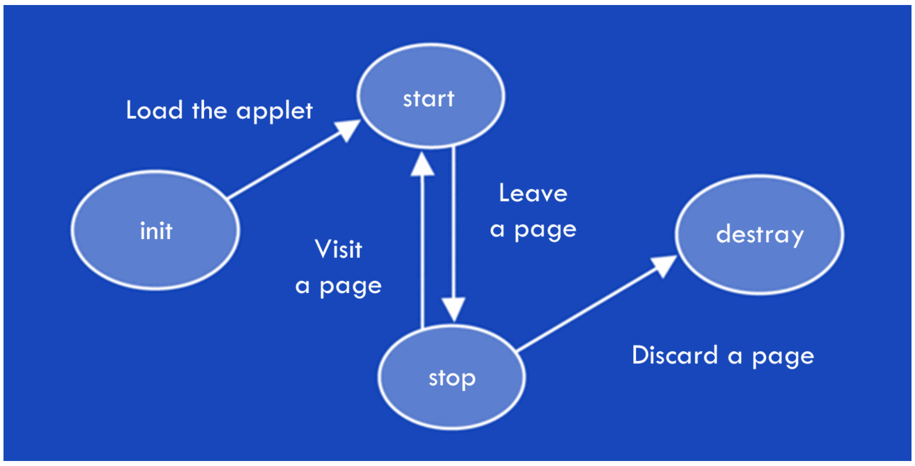
- .class files are downloaded from an HTTP server to the browser via the Internet, the JVM then executes them.
- As they use the Internet, they have very strict security restrictions, for example they can only read and write files from the server (and not from the local computer), the information they can access on the local computer is limited etc.
- No *applets* have their own window and they are executed in a browser window.
- From a programming point of view, the following aspects are important:
- They do not need the method `main`. They are executed using other mechanisms.
- They always derive from the java.applet.Applet class and therefore need to redefine some of their methods `such as init(), start(), stop()` and `destroy()`.
- They often also redefine other methods such as `paint()`, `update()` and `repaint()` that are inherited from higher classes for graphic tasks.
- They have a series of methods for getting information on the *applet* and on other *applets* being executed on the same page, such as `getAppletInfo()`, `getAppletContext()`, `getParameter()` etc.

9.1. The life cycle of *applets*

Due to its nature, the life cycle of an *applet* is more complex than that of a normal application. Each of the phases of the life cycle is marked by a call to a method of the *applet*:

- `void init()`. It is called when the *applet* is loaded and contains the initialisations needed.
- `void start()`. This is called when the page has loaded and has been stopped (due to minimisation or switching web pages etc.) and has then been re-activated.
- `void stop()`. This is called automatically when the *applet*. In this method the threads that are being executed are usually stopped to conserve resources.
- `void destroy()`. This method is called to free up the resources (except the memory) of the *applet*.

Figure 18.



As *applets* are graphic applications that appear in a browser window, it can also be useful to redefine the following method:

- void paint(Graphics g). This function should include all graphics operations as this method is also called when the *applet* is drawn for the first time and when it is re-drawn.

9.2. How to include *applets* in an HTML page

As we have already mentioned, to call an applet from an html page using `<APPLET> ... <\APPLET>` labels, these should include at least the following information:

- `CODE` = name of the *applet* (for example, myApplet.class)
- `WIDTH` = width of the window
- `HEIGHT` = height of the window

And, optionally, the following attributes:

- `NAME` = "aname" which allows it to communicate with other *applets*
- `ARCHIVE` = "anarchive" where the classes are stored in a .zip or a .jar
- `PARAM NAME` = "param1" VALUE = "value1" to be able to pass parameters to the *applet*.

9.3. My first *applet* in Java

The best way to understand the operation of *applets* is with a practical example. To create our first *applet* we will follow these steps:

1) Create a source file. We will use our chosen text editor to write the code and save it under the name `HelloWorldApplet.java`.

```
HelloWorldApplet.java

import java.applet.*;
import java.awt.*;
/**
 * The HelloWorld class shows the message
 * "Hello World" on the standard output.
 */
public class HelloWorldApplet extends Applet{
  public void paint(Graphics g)
  {
   // Shows "Hello World!"
   g.drawString("HelloWorld!", 75, 30 );
  }
}
```

2) Create an HTML file. We will use the text editor to write the text.

```
HelloWorldApplet.html


<HTML>
<HEAD>
<TITLE>My first applet</TITLE>
</HEAD>
<BODY>
I would like to send a message to you all:
<APPLET CODE="HelloWorldApplet.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

3) Compile the program to generate the *bytecode*.

```
javac HelloWorldApplet.java"
```

4) View the HelloWorldApplet.html page from a browser.

10. Programming graphic interfaces in Java

The appearance of graphic interfaces was a big step forward in the development of systems and applications. Until these appeared, programs were based on text pages (or consoles) and, in general, the flow of information in these programs was sequential and guided by the various options entered as the application requested them.

Graphic interfaces provide much faster communications for users and make it easier for them to interact with the system from many points on the screen. At any time we can choose from many very diverse operations (for example entering data, selecting menu options, modifying active forms, changing applications etc.) and, therefore, from many instruction flows, each one of which will be a response to an individual event.

Programs that use these interfaces are a clear example of the event-driven programming paradigm.

Graphic interfaces have evolved over time and new components have emerged (buttons, drop-down lists, option buttons etc.) to speed up communication between the user and the computer. Interacting with the components generates a series of status changes, each one of which is an event that may cause a certain action to be taken. These are all therefore possible events.

Graphic interface applications are programmed using a set of graphics components (from forms to controls such as buttons and labels) that are defined as individual objects with their own methods and variables.

While variables correspond to new properties needed for the description of the object (lengths, colours, blocks etc.), method allow us to encode a response to all of the different events which may happen to that component.

10.1. User interfaces in Java

Ever since version 1.0 Java has implemented a graphics routine package called AWT (*abstract windows toolkit*) which is included in the package `java.awt` , this contains all of the components needed to construct a graphics user interface (GUI-*graphic user interface*) and to manage events. This means that interfaces generated using this library will work in all Java environments, including the various browsers.

Many aspects of this package were revised and improved in version 1.1, but there was still one problem: AWT includes components that depend on the platform, which goes against one of the fundamental pillars of the Java philosophy.

In version 1.2 (or Java 2) a new graphic interface was implemented that solved this problem: the Swing package. As well as other advantages over AWT, this package includes customisable aspects (different *looks and feels*, such as Metal, which Java uses for its skin, and proprietary Motifs for Unix and Windows) and a wide variety of components that can be easily identified as their names begin with J.

Swing keeps the AWT event management, although this has been enriched with the javax.swing.event package.

The main problem with this is that some current browsers have not included it yet, meaning that in *applets* their use is limited.

Although this course does not cover the development of applications using graphic interfaces, we can get an idea of the basic concepts and the use of events by looking at a small example using the Swing library.

10.2. Example of an *applet* using Swing

In the following example we will define an *applet* that follows the Swing interface. The first difference between this and the *applet* we looked at before is the inclusion of the package `javax.swing.*`.

We define the class `HelloSwing` that inherits from the class `Japplet` (which corresponds to *applets* in Swing). This class includes the definition of the `init` method which defines a new button (`new Jbutton`) and adds it to the screen panel (`.add`).

Buttons receive events of the class `ActionEvent` and its events are managed by the class that implements the interface `ActionListener`.

For this function we have declared the class `EventManager` that, internally, redefines the method `actionPerformed` (the only method defined in the interface `ActionListener`) so that it opens a new window using the method showMessageDialog.

Lastly, we just need to tell the class `HelloSwing` that the class `EventManager` will manage the messages from the button. To do this we use the method `.addActionListener(EventManager)`

```
HelloSwing.java
```

```
import javax.swing.*;
import java.awt.event.*;
public class HelloSwing extends Japplet
{
  public void init()
  { //constructor
   JButton button = new JButton("Press here!");
   EventManager myManager = new EventManager();
    button.addActionListener(myManager); //Button manager
    getContentPane().add(button);
  } // init
} // HelloSwing

class EventManager implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
   String title = "Congratulations";
   String mensaje = "Hello world, from Swing";
   JOptionPane.showMessageDialog(null, message,
       title, JOptionPane.INFORMATION_MESSAGE);

  } // actionPerformed
} // class EventManager
```

11. Introduction to visual information

Although we have been using text mode development environments for simplicity reasons, in reality integrated development environments (IDE) are used which incorporate the tools needed by the programmer in the process of generating runnable source code (editor, compiler, *debugger* etc.).

However, there is no difference between these in terms of programming language: it is considered to be "textual" given that the primitive instructions are expressed using text.

Modern development environments allow us to work in a more visual way in that we can create windows (forms, reports etc.) by dragging controls to their position and then entering values for their attributes (colours, measurements etc.) and the code for each of the events they generate. However, the nature of the language does not change and it is still considered to be "textual".

Programming using "visual" languages can be considered to be a new programming paradigm. A *visual language* is one that manipulates information visually and supports visual interaction or allows programming using visual expressions. Their primitive components are therefore graphics, animations, drawings and icons.

In other words, programs are relationships between different instructions that are represented graphically. When these relationships are sequential and the instructions are expressed using words, the programming will be easily recognisable. However, we have already seen that concurrent programming and event driven programming do not use sequential instructions and they usually have a high level of abstraction.

Visual programming is therefore a good technique for describing programs whose execution flow follows the above-mentioned paradigms.

Despite the confusion brought about by the names of the various programming environments such as the Microsoft Visual family (Visual C++, Visual Basic etc.), these languages are still classified as "textual" languages, although their graphic programming environments mean that they come close to being visual programming.

Summary

In this unit we have looked at a new object oriented programming language that is independent from the platform where it is being executed. It provides a virtual machine which runs on each platform. Therefore, a program developer will only need to write the source code once and compile it to generate a common "runnable" code, meaning that the application can run in different environments such as the Unix system, PC systems and Apple Macintosh. This philosophy is known as "*write once, run everywhere*".

Java is an evolution of C++ which has been adapted to the conditions described above. It allows programmers to use their knowledge of the C and C++ languages so they can learn it more quickly.

As the Java environment is very small, it can be incorporated for use in web browsers. We took advantage of the fact that the use of these browsers usually implies the existence of a graphics environment to briefly look at the use of graphics libraries and the event driven programming model.

Java also includes advanced operations within its language as standard which, in other languages, would be performed by the operating system or additional libraries. One of these attributes is the programming of several execution threads (*threads*) within a single process. This unit has introduced us to the subject.

Self-evaluation

1. Expand the Read.java class to implement the reading of variables of the type `double`.

2. Enter the date (requesting a string for the town and three numbers for the date) and return it in text form.

Example

Input: Barcelona 15 02 2003

Output: Barcelona, 15th February 2003

3. Implement an application that is able to tell if a figure made up of four points is a square, a rectangle, a rhomboid or another type of polygon.

These would be defined in the following way:

- Square: sides 1, 2, 3 and 4 are equal; 2 diagonals are equal
- Rectangle: Sides 1 and 3, 2 and 4 are equal; 2 diagonals are equal
- Rhomboid: sides 1, 2, 3 and 4 are equal; the 2 diagonals are different
- Polygon: any other situation

To do this we will need to define the class Point2D which defines the x, y coordinates and the "distance to another point" method.

Example

(0,0) (1,0) (1,1) (0,1) Square

(0.1) (1.0) (2.1) (1.2) Square

(0,0) (2,0) (2,1) (0,1) Rectangle

(0,2) (1,0) (2,2) (1,4) Rhomboid

4) Convert the code of the lift exercise (exercise 3 of unit 4) to Java.

Answer key

1.

```
Read.java

import java.io.*;

public class Read
{
  public static String getString()
  {
   String str = "";
   try
   {
    InputStreamReader isr = new
    InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    str = br.readLine();
   }
   catch(IOException e)
 { System.err.println("Error: " + e.getMessage());
}
    return str; // returns the typed data
   }

   public static int getInt()
   {
    try
    { return Integer.parseInt(getString()); }
    catch(NumberFormatException e)
    { return 0; // Integer.MIN_VALUE }
   } // getInt

   public static double getDouble()
   {
    return Double.parseDouble(getString());
   }
   catch(NumberFormatException e)
   {
       return 0; // Double.MIN_VALUE
      }
     } // getDouble
  } // Read
```

2.

```
constructDate.java

import java.io.*;
public class constructDate
{
  static String nameMonth(int nmonth)
  {
   String strmonth;
   switch (nmonth)
   {
    case 1: { strmonth = "January"; break; }
    case 2: { strmonth = "February"; break; }
    case 3: { strmonth = "March"; break; }
    case 4: { strmonth = "April"; break; }
    case 5: { strmonth = "May"; break; }
    case 6: { strmonth = "June"; break; }
    case 7: { strmonth = "July"; break; }
    case 8: { strmonth = "August"; break; }
    case 9: { strmonth = "September"; break; }
    case 10: { strmonth = "October"; break; }
    case 11: { strmonth = "November"; break; }
    case 12: { strmonth = "December"; break; }
    default: { strmonth = " -- "; break; }
   } // switch nmonth
   return (strmonth);
  } // nameMonth

  public static void main(String args[] )
  {
   String town;
   int day, month, year;
   String mydate, strmonth;
   System.out.print(" Town: ");
   town = Read.getString();
   System.out.print(" Day: ");
   day = Read.getInt();
   System.out.print(" Month: ");
   month = Read.getInt();
   System.out.print(" Year: ");
```

```
     year = Read.getInt();
     mydate = town + ", " + day;
    mydate = mydate +" of "+ nameMonth(month) +"
                        of "+ year;
    System.out.print(" The date entered is: ");
    System.out.println (mydate);
    } // main
  } // class
```

3.

```
Point2D.java

class Point2D
{
  public int x, y;
  // initialising the coordinate origin
  Point2D()
  { x = 0; y = 0; }

 // initialising a specific x,y coordinate
  Point2D(int coordx, int coordy)
  { x = coordx; y = coordy; }



  // calculating the distance to another point
  double distance(Point2D myPoint)
  {
   int dx = x - myPoint.x;
   int dy = y - myPoint.y;
   return ( Math.sqrt(dx * dx + dy * dy));
  }
}

AppRecogniseFigure.java
class AppRecogniseFigure
{
static public void main(String args[])
  {
  int i;
  int coordx, coordy;
  // Enter 4 points and
  // indicate which is closest to the origin.

  Point2D listPoints[];
  listPoints = new Point2D[4];

  // Enter data
  for (i=0; i<4; i++)
  {
   System.out.println("Enter the point (" + i + ")" );
```

```
    System.out.print("Coordinate x " );
    coordx = Read.getInt();
    System.out.print("Coordinate y " );
    coordy = Read.getInt();
    listPoints[i] = new Point2D(coordx, coordy);
   } //for

   // indicate if the 4 points form a
   // square: dist1 = dist2 = dist3 = dist4
   // diag1 = diag2
   // rhomboid: dist1 = dist2 = dist3 = dist4
   // diag1 <> diag2
   // rectangle: dist1 = dist3, dist2 = dist4
   // diag1 = diag2
   // polygon: other cases

   double dist[] = new double[4];
   double diag[] = new double[3];

   // distance calculations
   for (i=0; i<3 ; i++)
   {
    dist[i] = listPoints[i].distance(listPoints[i+1]);
    System.out.print("Distance "+i + " " + dist[i] );
   } //for
   dist[3] = listPoints[3].distance(listPoints[0]);
   System.out.println("Distance "+i + " " + dist[3] );

   // calculating diagonals
   for (i=0; i<2; i++)
   {
    diag[i] = listPoints[i].distance(listPoints[i+2]);
   } //for
 if ( (dist[0] == dist[2]) && (dist[1] == dist[3]) )
   {

   // it is a square, rectangle or rhomboid
   // it is a square or rhomboid
   if (diag[0] == diag[1])
```

4.

```
AppLift.java

import java.io.*;

public class AppLift{

  static int n_code, n_weight, n_language;


  public static void requestData()
  {
   System.out.print("Code: ");
   n_code = Read.getInt();

  System.out.print("Weight: ");
  n_weight = Read.getInt();

  System.out.print(
      "Language: [1] Catalan [2] Spanish [3] English ");
  n_language = Read.getInt();
 } // requestData

 public static void showStateLift(Lift nA)
 {
  nA.showOccupancy();
  System.out.print(" - ");
  nA.showLoad();
  System.out.println(" ");
  nA.showListPassengers();
 } // showStateLift


 public static void main( String[] args)
 {
  int opc;
  boolean leave = false;
  Lift aLift;
  Person onePerson;
  Person locatePerson;


  opc=0;


 aLift = new Lift();// initialising lift
 aPerson = null;// initialising aPerson


 do {
   System.out.print(
"LIFT: [1] Enter [2] Exit [3] Status [0] End ");
   opc = Read.getInt();

   switch (opc)
   {
   case 1: { // Option Enter
    requestData();
    switch (n_language)
   {
    case 1: { //"Catalan"
     onePerson = new Catalan (n_code, n_weight);
     break;
    }
    case 2: { //"Spanish"
     onePerson = new Spanish (n_code, n_weight);
     break;
    }
    case 3: { //"English"
     aPerson = new English(ncode, nweight);
     break;
    }
     default: { //"English"
     aPerson = new English(ncode, nweight);
     break;
    }
   } //switch n_language
   if (aLift.person_MayEnter(aPerson))
   {
     aLift.person_Enter(aPerson);
     if (aLift.getOccupancy()>1)
     {
      System.out.print(onePerson.getCode());
      System.out.print(" says: ");
      aPerson.greet();
      System.out.println(" "); // The others reply
      aLift.restoLift_Greet(aPerson);
     }
    } //may enter
```

```
 break;
}
case 2: { //Option Leave

 locatePerson = new Person(); //For example
 aPerson = null;

 locatePerson.requestCode();
 if (aLift.person_Select(locatePerson))
 {
  aPerson = aLift.getRefPerson();
  aLift.person_Leave( aPerson );
  if (aLift.getOccupancy()>0)
  {
   System.out.print(onePerson.getCode());
   System.out.print(" says: ");
   aPerson.farewell();
   System.out.println(" "); // The others reply
   aLift.restLift_Farewell(onePerson);
   aPerson=null;
  }
 } else {
   System.out.println(
    "There is no one with this code");
 } // select
 locatePerson=null;
 break;
}
case 3: { //Status
 showState(aLift);
 break;
}

case 0: { //End
 System.out.println("End");
 leave = true;
 break;
    } //switch opc
 } while (! leave);
 } // main
} //AppLift

Lift.java

import java.io.*;

class Lift {
  private int occupancy;
  private int load;
  private int maximumOccupancy;
  private int maximumLoad;
  private Person passengers[];
  private Person refPersonSelected;
//
// Constructors y destructors
//
Lift()
{
  occupancy = 0;
  load=0;
  maximumOccupancy=6;
  maximumLoad=500;
  passengers = new Person[6];
  refPersonSelected = null;
} //Lift()

// Access functions
int GetOccupancy()
{ return (occupancy); }

void ModifyOccupancy(int dif_occupancy)
{ occupancy += dif_occupancy; }

void ShowOccupancy()
{
  System.out.print("Current occupancy: ");
  System.out.print(occupancy );
}
```

```
int getLoad()
{ return (load); }

void modifyLoad(int dif_load)
{ load += dif_load; }


void showLoad()
{ System.out.print("Current load: ");
  System.out.print(load) ;
}


Persona getRefPerson()
{return (refPersonSelected);}


boolean person_MayEnter(Person aPerson)
{
  // if occupancy does not exceed the occupancy limit and
  // if the load does not exceed the load limit
  // -> may enter


  boolean tmpMayEnter;
  if (occupancy + 1 > maximumOccupancy)
  {
   System.out.println(
  "Warning: The lift is full. You may not enter");
   return (false);
  }


  if (aPerson.getWeight() + load > maximumLoad )
  {
   System.out.println(
"Warning: The lift has exceeded its maximum load. You may not
   return (false);
  }
  return (true);
}
boolean person_select(Person locatePerson)
{
  int counter;
  // Select a person from among the passengers of the lift.
  boolean personFound = false;
  if (getOccupancy() > 0)
  {
   counter=0;
   do {
    if (passengers[counter] != null)
    {
     if(passengers[counter].sameCode(locatePerson)
     {
       refPersonSelected=passengers[counter];
       personFound=true;
       break;
      }
     }
    counter++;
    } while (counter<maximumOccupancy);
    if (counter>=maximumOccupancy)
     {refPersonSelected=null;}
   }
   return (personFound);
 }

 void person_Enter(Person onePerson)
 {
   int counter;
   modifyOccupancy(1);
   modifyLoad(onePerson.getWeight());
   System.out.print(onePerson.getCode());
   System.out.println(" enter the lift ");

   counter=0;
   // we have already checked that there is space
   do {
    if (passengers[counter]==null )
    {
     passengers[counter]=onePerson;
     break;
    }
    counter++;
   } while (counter<maximumOccupancy);
 }
```

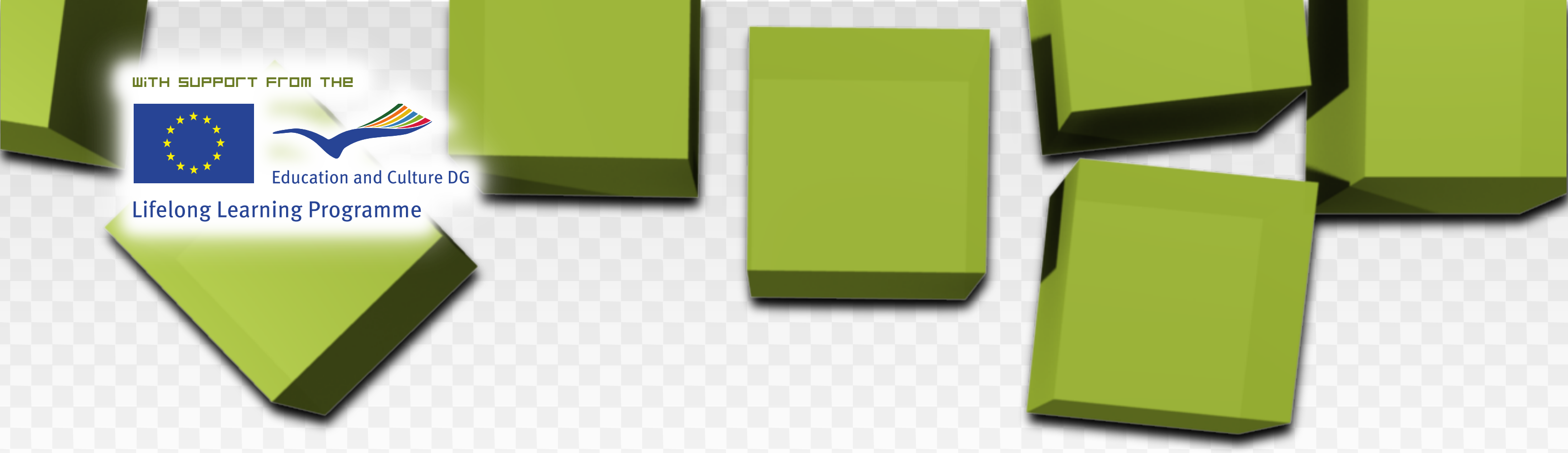
```
void person_Leave(Person onePerson)
{
  int counter;

  counter=0;
  do {
   if ((passengers[counter]==onePerson ))
   {
    System.out.print(onePerson.getCode());
    System.out.println(" leave the lift ");
    passengers[counter]=null;

    // Modify the occupancy and the load
    modifyOccupancy(-1);
    modifyLoad(-1 * (aPerson.getWeight()));
    break;
   }
   counter++;
  } while (counter<maximumOccupancy);
  if (counter==maximumOccupancy)
   {System.out.println(
   "There is no one with this code. Nobody leaves ");}
}

void showListPassengers()
{
  int counter;
  Person onePerson;

  if (getOccupancy() > 0)
  {
   System.out.println("List of passengers in the lift:");
   counter=0;
   do {
    if (!(passengers[counter]==null ))
    {
    onePerson=passengers[counter];
    System.out.print(onePerson.getCode());
    System.out.print("; ");
   }
```

```
      counter++;
     } while (counter<maximumOccupancy);
     System.out.println(" ");
    } else {
     System.out.println("The lift is empty");
    }
   }

   void restLift_Greet(Person onePerson)
   {
     int counter;
     Person otherPerson;

     if (getOccupancy() > 0)
     {
      counter=0;
      do {
       if (!(passengers[counter]==null ))
       {
        otherPerson=passengers[counter];
        if (!onePerson.sameCode(otherPerson) )
      {
       System.out.print(otherPerson.getCode());
       System.out.print(" replies: ");
       otherPerson.greet();
       System.out.println(" ");
      }
     }
     counter++;
    } while (counter<maximumOccupancy);
   }
 }

 void restLift_Farewell(Person aPerson)
 {
   int counter;
   Person otherPerson;
```

```
  if (getOccupancy() > 0)
  {
   counter=0;
   do {
    if (!(passengers[counter]==null ))
    {
     otherPerson=passengers[counter];
     if (!(aPerson.sameCode(otherPerson))
     {
      System.out.print(otherPerson.getCode());
      System.out.print(" replies: ");
      otherPerson.farewell();
      System.out.print(" ");
     }
    }
    counter++;
   } while (counter<maximumOccupancy);
  }
}

} // class Lift

Person.java

import java.io.*;

class Person {
  private int code;
  private int weight;

  Person()
  { }

  Person(int n_code, int n_weight)
  {
   code = n_code;
   weight = n_weight;
  }

  public int getWeight()
   { return (weight); }
  public void assignWeight(int n_weight)
   { weight = n_weight; }

  public int getCode()
  { return (code); }
  public void assignCode(int n_code)
{ this.code = n_code; }

  public void assignPerson(int n_code, int n_weight)
  {
   assignCode( n_code );
   assignWeight( n_weight );
  }
  void greet() {};
  void farewell() {};

  public void requestCode()
  {
   int n_code=0;
   System.out.print("Code: ");
   n_code = Read.getInt();
   assignCode( n_code );
  }

  public boolean sameCode(Person otherPerson)
  {
   return
    (this.getCode()==otherPerson.getCode());
  }
} //class Person
```

```
Catalan.java

class Catalan extends Person
{
   Catalan()
   { Person(0, 0); };
```

```
  Catalan(int n_code, int n_weight)
  { Person (n_code, n_weight); };

  void greet()
  { System.out.println("Bon dia"); };

  void farewell()
  { System.out.println("Adéu"); };
}

Spanish.java

class Spanish extends Person
{
  Spanish()
  { Person(0, 0); };

  Spanish(int n_code, int n_weight)
  { Person (n_code, n_weight); };

  void greet()
  { System.out.println("Buenos días"); };

  void farewell()
  { System.out.println("Adiós"); };
}

English.java
class English extends Person
{

  English()
  { Person(0, 0); };

  English (int n_code, int n_weight)
  { Person (n_code, n_weight); };
  void greet()
  { System.out.println("Hello"); };
  void farewell()
  { System.out.println("Bye"); };
}
```



With support from the
Education and Culture DG
Lifelong Learning Programme

This coursebook is designed for IT specialists and developers that are starting their way in the free software development universe. Free software is developed with specific collaboration technics and tools that engage and enable world-wide communities. Professionals need to handle different programming technics, languages and develop specific workgroup skills.

The programming languages used in this module are C, C++ or Java.

freeknowledgeinstitute
UOC
Universitat Oberta de Catalunya
www.uoc.edu
www.ou.nl
UNIVERSITETET I AGDER