2 General Features of C++

In this chapter, we explain how to write, compile, and execute (run) a basic C++ program.

The program is written in a file using a text editor such as *vi*, *gedit*, or *emacs*. A file containing C++ code is conventionally designated by one of the suffixes:

`.c` `.cc` `.cpp` `.cxx`

Thus, a C++ source file can be named

`kalambaka.c` `edessa.cc` `kourkoubinia.cpp` `mageiras.cxx`

The C++ source files are compiled and linked through a C++ compiler to produce the corresponding binary executable, as discussed in Chapter 1.

Free C++ compilers are available for the Linux platform thanks to the *gnu* free-software foundation. Cygwin and Borland offer complimentary compilers for other operating systems. Some compilers are bundled in an integrated development environment (IDE) offering dazzling graphical user interfaces (GUI).

2.1 The main function

Each C++ application (complete code) has a main function that is first loaded into memory and then transferred to the CPU for execution. When execution has been concluded, the main function returns the integer `0`. This practice is motivated partly by issues of backward compatibility.

The main function has the general syntax:

```
int main()
{
...
return 0;
}
```

where:

- `int` indicates that an integer will be returned on completion. The penultimate line sets this integer to `0`, signaling the success of the execution.
- The parentheses after `main` enclose the arguments of the main function; in this case, there are no arguments.
- The curly brackets mark the beginning and the end of the enclosed main program consisting of various instructions.
- The dots stand for additional lines of code.
- The semicolon is a delimiter, marking the end of the preceding command `return 0`, which concludes the execution.

A simplified version of the main program that returns nothing on execution is:

```
main()
{
...
}
```

However, the previous structure with the return statement included is highly recommended as a standard practice.

In Chapter 5, we will see that the main function can not only return, but also receive information from the operating system. In that case, the parentheses in

```
int main()
```

will enclose command line arguments.

Problem

2.1.1. We saw that C++ uses parentheses and curly brackets. What other bracket delimiters do you anticipate based on the symbols printed on your keyboard?

2.2 Grammar and syntax

Next, we review the most important rules regarding the grammar and syntax of C++. If an error occurs during compilation or execution, this list should serve as a first checkpoint.

C++ is (lower and upper) case sensitive

For example, the variable `echidna` is different from the variable `echiDna`, and the C++ command `return` is not equivalent to the non-existent command `Return`.

Beginning of a statement

A C++ statement or command may begin at any place in a line and continue onto the next line. In fact, a statement may take several lines of code. We say that C++ is written in *free form.*

End of a statement

The end of a statement is indicated by a semicolon ";" (statement delimiter.) Thus, we write:

```
a=5;
```

If we do not include the semicolon, the compiler will assume that the statement in the next line is a continuation of the statement in the present line.

Multiple commands in a line

Two or more statements can be placed in the same line provided they are separated with semicolons. Thus, we may write:

```
a=5; b=10;
```

White space

An empty (blank) space separates two words. The compiler ignores more than one empty space between two words. A number cannot be broken up into pieces separated by white space; thus, we may not write 92 093 instead of 92093.

Statement and command blocks

Blocks of statements or commands defining procedures are enclosed by curly brackets (block delimiters)

```
{
...
}
```

Note that it is not necessary to put a semicolon after the closing bracket. This practice is consistent with the structure of the main program discussed in Section 2.1.

In-line comments

In-line comments may be inserted following the double slash "//". For example, we may write:

```
a = 10.0; // ignore me
```

The text: `// ignore me` is ignored by the compiler.

To deactivate (comment out) a line, we write:

```
// a = 34.5;
```

A distinction should be made between the slash (/) and the backslash (\). These are two different symbols separated by two rows on the keyboard.

Commentary

All text enclosed between a slash-asterisk pair (/*) and the converse asterisk-slash pair (*/) is commentary and ignored by the compiler. Thus, we may write:

```
/* ---- main program ----- */
```

To provide documentation at the beginning of a code, we may write:

```
/* PROGRAM: late
AUTHOR: Justin Case
PURPOSE: produce an excuse for being late */
```

Problems

2.2.1. How many commands are executed in the following line?

```
a=3.0; // b=4.0;
```

2.2.2. How does the compiler interpret the following line?

```
/* my other /* car is */ a vartburg */
```

2.3 Data types

In mathematical modeling and computer programming, we introduce variables representing abstract notions and physical objects. Examples are the temperature, the velocity, the balance of a bank account, and the truthfulness of a theorem.

In C++, the name of a variable must start with a letter and contain only letters, numbers, and the underscore (_). Names reserved for C++ grammar and syntax given in Appendix E cannot be employed. Acceptable variables obey the rules discussed in this section.

Numerical variable declaration:

Every numerical variable must be declared either as an integer (whole number) or as a real (floating point) number registered in single or double precision. In the remainder of this text, we adopt a mathematical viewpoint and we refer to a non-integer as a real number.

Suppose that `a` is an integer, `b` is a real number registered in single precision, and `c` is a real number registered in double precision. The statements declaring these variables are:

```
int a;
float b;
double c;
```

Suppose that `i` is an integer and `j` is another integer. We can declare either:

```
int i;
int j;
```

or

```
int i, j;
```

Note the obligatory use of a comma.

Why does a variable have to be declared? Appropriate space must be reserved in memory by the compiler.

Numerical variable initialization and evaluation

A numerical variable is not necessarily initialized to zero by default when declared, and may be given a value already recorded previously in the assigned memory address.

Once declared, a numerical variable can be initialized or evaluated. For example, we may write:

```
int a;
a=875;
```

Declaration and initialization can be combined into a single statement:

```
int a = 875;
```

An equivalent but less common statement is:

```
int a (875);
```

In these statements, the numerical value 875 is a *literal.*

To introduce a real number registered in single precision, we may state:

```
float b = -9.30;
```

or

```
float c = 10.45e-3;
```

meaning that $c = 10.3 \times 10^{-3}$. The numerical values on the right-hand sides of these statements are literals.

A literal cannot be broken up into pieces separated by white space. For example, the following declaration is incorrect:

```
double pi=3.141592 653589 793238;
```

The correct declaration is:

```
double pi=3.141592653589793238;
```

Integer evaluation

An integer can be evaluated in the decimal, octal, or hexadecimal system. The statement:

```
int a=72;
```

implies

$$a = 7 \times 10^2 + 2 \times 10^0.$$

The statement:

```
int a = 023;
```

with a leading zero (0), implies

$$a = 2 \times 8^1 + 3 \times 8^0.$$

The statement:

```
int a = 0xA4;
```

with a leading zero (0) followed by `x` implies

$$a = 10 \times 16^1 + 4 \times 16^0.$$

Boolean variables

A Boolean variables can be either `false` or `true`. When a Boolean variable is printed, it appears as 1 or 0, respectively, for true and false.

The following statements declare and initialize the Boolean variable `hot`:

```
bool hot;
hot = true;
```

An equivalent statement is:

```
bool hot = true;
```

Boolean variables are useful for assessing states and making logical decisions based on deduced outcomes.

Characters

A single character is encoded according to the ASCII protocol described in Appendix D. The following statements declare and initialize a character:

```
char a;
a = 66;
```

In compact form:

```
char a = 66;
```

When the character `a` is printed, it appears as the letter B. Alternatively, we may define:

```
char a;
a = 'B';
```

or even combine the two statements into one line:

```
char a = 'B';
```

Note the mandatory use of *single* quotes. This example confirms that the ASCII code of the letter B is 66.

To find the ASCII code of a character, we may typecast it as an integer. For example, we may write:

```
char a = 'B';
int c = a;
```

If we print the integer c, it will have the value 66.

Strings

A string is an array of characters. The following statements define and initialize a string:

```
string name;
name = "Kolokotronis";
```

Note the mandatory use of *double* quotes. The two statements can be consolidated into one:

```
string name = "Kolokotronis";
```

Alternatively, we may state:

```
string name ("Kolokotronis");
```

Other data types

C++ supports the data types shown in Table 2.3.1. The number of bytes reserved in memory and the range of the data types depend on the specific system architecture. The values shown in Table 2.3.1 are those found on most 32-bit systems. For other systems, the general convention is that `int` has the natural size suggested by the system architecture (one word), and each of the four integer types:

Type	*Description*	*Byte size*
short int short signed short int unsigned short int	Short integer Ranges from -32768 to 32767 Ranges from 0 to 65535	2
int signed int unsigned int	Integer Ranges from -2147483648 to 2147483647 Ranges from 0 to 4294967295	4
long int long signed long int unsigned long int	Long integer Ranges from-2147483648 to 2147483647 Ranges from 0 to 4294967295	4
float	Floating point number Real number inside $3.4\,\mathrm{e}^{\pm 38}$	4
double	Double precision Floating point number Real number inside $1.7\,\mathrm{e}^{\pm 308}$	8
long double	Long double precision Floating point number Real number inside $1.7\,\mathrm{e}^{\pm 308}$	12
bool	Boolean value "true" or "false"	1
char signed char unsigned char	Encoded character Integer ranging from -128 to 127 Integer ranging from 0 to 255	1
wchar_t	Wide character Used for non-English letters	4
string $stringname$ char $stringname[]$	String of characters Array of characters	4

Table 2.3.1 Data types supported by C++ and their common memory allocation. *unsigned* only allows positive integers. *signed* is the default type of integers and characters.

```
char  short  int  long
```

must be at least as large as the one preceding it. The same applies to the floating point types:

```
float     double    long double
```

Each must provide at least as much precision as the one preceding it.

The size of the different data types listed in Table 2.3.1 can be confirmed by using the `sizeof` operator discussed in Section 3.1.

Constants

To fix the value of a variable and thus render the variable a constant, we include the keyword `const`. For example, we may declare

```
const float temperature;
```

Constants are variables that, once evaluated, remain fixed and thus cease to be variables.

Aliases

We can introduce an alias of a declared variable so that we can refer to it by a different name. For example, we may declare:

```
float a;
float& a_alias = a;
```

Since `a_alias` and `a` are truly the same variable, any operation on one amounts to the same operation on the other. In C++, an alias is better known as a reference.

Defined data types

C++ allows us to duplicate a data type into something that is either more familiar or more convenient. For example, if *year* is a non-negative integer, we may declare:

```
unsigned int year;
```

Since the year is positive, we have exercised the `unsigned` option.

We can duplicate the cumbersome "`unsigned int`" into "`hronos`" meaning year in Greek, by stating:

```
typedef unsigned int hronos;
```

The data types `unsigned int` and `hronos` are now synonyms. We may then declare:

```
hronos year;
```

The Unix-savvy reader will notice that the "`typedef`" command works like the copy command, "`cp`", the move command, "`mv`", and the symbolic link command,"`ln -l`":

```
cp file1 file2
```

copies file1 into file2,

```
mv file1 file2
```

renames file1 into file2, and

```
ln -s file1 file2
```

symbolically links file1 into its alter ego file2.

Problems

2.3.1. Declare and initialize at the value of 77 the integer `a` using (a) the octal, and (b) the hexadecimal representation.

2.3.2. What are the values of the integers `c` and `d` evaluated by the following statements?

```
char a = '='; int c = a;
char b = '1'; int d = b;
```

2.4 Vectors, arrays, and composite data types

The basic data types introduced in Section 2.3 can be extended into composite groups that facilitate notation and book-keeping in a broad range of scientific and other applications.

Vectors

In C++, a one-dimensional array (vector) v_i is declared as `v[n]`, where n is an integer, and $i = 0, \ldots, n-1$. Thus, the lower limit of an array index is always 0.

For example, a vector **v** with thirty slots occupied by real numbers registered in double precision, beginning at `v[0]` and ending at `v[29]`, is declared as:

```
double v[30];
```

Note that the elements of the vector are denoted using square brackets, `v[i]`, not parentheses, `v(i)`. Parentheses in C++ enclose function arguments.

Similarly, we can declare:

```
char a[19];
```

and

```
string a[27];
```

Successive elements of a vector are stored in consecutive memory blocks whose length depends on the data type.

In C++ jargon, the term "vector" sometimes implies a one-dimensional array with variable length.

Matrices

A two-dimensional array (matrix) A_{ij} is declared as `A[m][n]`, where n and m are two integers, $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$. The lower limit of both indices is 0.

For example, the two indices of the 15×30 matrix $A[15][30]$ begin at $i, j = 0$ and end, respectively, at $i = 14$ and $j = 29$. If the elements of this matrix are integers, we declare:

```
int A[15][30];
```

Note that the elements of the matrix are denoted as `v[i][j]`, not `v(i,j)`. The individual indices of a matrix are individually enclosed by square brackets.

Similarly, we can declare the array of characters:

```
char A[13][23];
```

and the array of strings:

```
string A[9][38];
```

Successive *rows* of a matrix are stored in consecutive memory blocks.

Data structures

Consider a group of M objects,

$$o1, o2, \ldots, oM,$$

a group of N properties,

$$p1, p2, \ldots, pN,$$

and denote the jth property of the ith object by:

$$oi.pj$$

The individual properties of the objects can be accommodated in a data structure defined, for example, as:

```
struct somename
{
  int p1;
  float p2;
  double p3;
  double p4;
}
o1, o2, o3;
```

Alternatively, we may define a data structure in terms of the properties alone by declaring:

```
struct somename
{
  int p1;
  float p2;
  double p3;
  double p4;
};
```

and then introduce members by declaring:

```
somename o1;
somename o2, o3;
```

Objects and properties are threaded with a dot (.) into variables that

convey expected meanings:

```
int o1.p1;
float o1.p2;
double o2.p3;
char o1.p4;
```

The mathematically inclined reader will recognize that this threading is the tensor product of two vectors, $\mathbf{o} \otimes \mathbf{p}$. In computer memory, the variables

```
o1.p1   o1.p2   o1.p3 ...
```

are stored in consecutive memory blocks.

As an example, we define the used car lot structure:

```
struct car
{
  string make;
  int year;
  int miles;
  bool lemon;
}
vartburg1, skoda1, skoda2;
```

and then set:

```
skoda1.make = "skoda";
vartburg1.miles= 98932;
skoda1.lemon = true;
skoda2.lemon = false;
```

Data structures and their members are preludes to classes and objects discussed in Chapter 6.

Enumerated groups

One way to represent a property, such as flavor, is to encode it using integers. For example, we may assign:

$$\text{bitter} \to 4, \quad \text{sweet} \to 5, \quad \text{salty} \to 6, \quad \text{hot} \to 7, \quad \text{sour} \to 8.$$

We then know that if `peasoup_flavor=6`, the soup is salty.

C++ allows us to mask this encoding by defining enumerations. In our example, we declare:

```
enum flavor {bitter=4, sweet, salty, hot, sour};
```

where bitter is encoded as 4, sweet is encoded as 5, salty is encoded as 6, hot is encoded as 7, and sour is encoded as 8. The starting integer, 4, is arbitrary and can be omitted, in which case the default value of 0 is used. We may then state:

```
flavor peasoup_flavor;
peasoup_flavor = salty;
```

The broad range of standard features offered by C++, combined with its ability to generate unlimited user-defined structures, explain its popularity and suitability for building large code.

Problems

2.4.1. Define a structure of your choice.

2.4.2. Define an enumerated group of your choice.

2.5 System header files

When a FORTRAN 77 code is compiled to produce an executable (binary) file, the linker automatically attaches the necessary library files that allow, for example, data to be read from the keyboard and data to be written to the monitor. Other library files ensure the availability of intrinsic mathematical and further functions.

In contrast, in C++, supporting functions, mathematical functions, and other ancillary services required during execution must be explicitly requested. This is done by placing at the beginning of each file containing the C++ code an `include` statement or a collection of `include` statements handled by the preprocessor. The C++ preprocessor runs as part of the compilation process, adding to the compiled program necessary code and removing unnecessary code.

An `include` statement asks the preprocessor to attach at the location of the statement a copy of a *header file* containing the definition of a desired class of system or user-defined functions. Both are regarded as *external implementations.*

The system header files reside in a subdirectory of a directory where the C++ compiler was installed, whereas the user-defined header files reside in user-specified directories. For example, in Linux, system header files reside in *include* directories, such as the */usr/include* or the */use/local/include* directory.

Once the header files have been copied, the compiler searches for and attaches the implementations of the required external functions located in system or user-defined library files and directories.

For example, putting at the beginning of the code the statement:

```
#include <iostream>
```

instructs the C++ preprocessor to attach a header file containing the definition, but not the implementation, of functions in the input/output stream library. In the Fedora Core 5 Linux distribution, the `iostream` header file is located in the */usr/include/c++/4.1.1* directory.

Thus, the main function of a code that uses this library has the general structure:

```
#include <iostream>
...
int main()
{
...
return 0;
}
```

where the three dots denote additional lines of code.

Similarly, putting at the beginning of a source code the statement:

```
#include <cmath>
```

ensures the availability of the C++ mathematical library. In this case, *cmath* is a header file containing the definition, but not the implementation, of mathematical functions.

Thus, the main function of a code that uses both the input/output and the mathematical libraries has the general syntax:

```
#include <iostream>
#include <cmath>
...
int main()
{
...
return 0;
}
```

where the three dots denote additional lines of code.

A statement following the # character in a C++ code is a *compiler* or *preprocessor directive.* Other directives are available.

Problems

2.5.1. Locate the directory hosting the `iostream` header file in your computer.

2.5.2. Prepare a list of mathematical functions declared in the `cmath` header file.

2.6 Standard namespace

Immediately after the include statements, we state:

```
using namespace std;
```

which declares that the names of the functions defined in the standard `std` system library will be adopted in the code. This means that the names will be stated plainly and without reference to the `std` library.

In large codes written by many authors, and in codes linked with libraries obtained from different sources or vendors, names may have multiple meanings defined in different namespaces.

If we do not make the "using namespace std" declaration, then instead of stating:

```
string a;
```

we would have to state the more cumbersome:

```
std::string a;
```

What names are defined in the `std` library? We can find this out by trial and error, commenting out the "using namespace std" line and studying the errors issued on compilation.

Thus, the main function of a code that uses the standard input/output library and the mathematical library has the general form:

```
#include <iostream>
#include <cmath>
using namespace std;
...
int main()
{
...
return 0;
}
```

where the three dots denote additional lines of code. This fundamental pattern will be used as a template in all subsequent codes.

Problem

2.6.1. Is the integer declaration `int` in the standard namespace? Deduce this by trial and error.

2.7 Compiling in Unix

Suppose that a self-contained C++ program has been written in a single file named *addition.cc*. To compile the program on a Unix system, we navigate to the directory where this file resides, and issue the command:

```
c++ addition.cc
```

This statement invokes the C++ compiler with a single argument equal to the file name. The compiler will run and produce an executable binary file named *a.out*, which may then be loaded into memory (executed) by issuing the command:

```
a.out
```

It is assumed that the search path for executables includes the current working directory where the *a.out* file resides, designated by a dot (.). To be safe, we issue the command:

```
./a.out
```

which specifies that the executable is in the current directory.

Alternatively, we may compile the file by issuing the command:

```
c++ -o add addition.cc
```

This will produce an executable file named *add*, which may then be loaded (executed) by issuing the command:

```
add
```

or the safer command:

```
./add
```

Other compilation options are available, as explained in the compiler manual invoked by typing:

```
man gcc
```

for the GNU project C and C++ compilers.

Makefiles

C++ programs are routinely compiled by way of Unix makefiles, even if a code consists of a single file. If a complete C++ code is contained in the file *addition.cc*, we create a file named *makefile* or *Makefile* in the host directory of *addition.cc*, containing the following lines:

```
LIBS =
papaya:  addition.o
      c++ -o add addition.o $(LIBS)
addition.o:  addition.cc
      c++ -c addition.cc
```

The empty spaces in the third and fifth lines must be generated by pressing the TAB *key inside the text editor.*

- The first line of the makefile defines the variable `LIBS` as the union of external binary libraries and header files to be linked with the source code. In this case, no libraries or header files are needed, and the variable `LIBS` is left empty.

- The second line defines a project named *papaya* that depends on the object file *addition.o*. Subsequent indented lines specify project tasks.

- The third line names the process for creating the executable *add*; in this case, the process is compilation and linking. Note that the name of the executable is not necessarily the same as the name of the the source file. The flag `-o` requests the production of an executable.

- The fourth line defines the project *addition.o* that depends on the source file *addition.cc*. The flag `-c` signifies compilation.

- The fifth line states the process for creating the object file *addition.o*; in this case, the process is compilation.

The object file *addition.o* and the executable *add* are generated by issuing the command:

```
make papaya
```

make is a Unix application that reads information from the file *makefile* or *Makefile* in the working directory. In our example, the application performs all operations necessary to complete the project papaya. A condensed version of the papaya project is:

```
papaya:  addition.cc
      c++ -o add addition.cc
```

Other projects can be defined in the same makefile. If we type:

```
make
```

the first project will be tackled. Further information on the `make` utility can be obtained by referring to the manual pages printed on the screen by issuing the command:

```
man make
```

In Chapter 4, we will discuss situations in which the C++ code is split into two or more files. Each file is compiled individually, and the object files are linked to generate the executable. In these cases, compiling through a makefile is practically our only option. If the code is written in an integrated development environment (IDE), the compilation process is handled as a project through a graphical user interface (GUI).

Typesetting this book

This book was written in the typesetting language *latex*. To compile the source file named *book.tex* and create a portable-document-format (pdf) file, we have used the makefile:

```
manuscript:
      latex book.tex
      makeindex book
      dvips -o book.ps book.dvi
      ps2pdf book.ps
```

The first line names the task. The second and third lines compile the source code, prepare the subject index, and generate a compiled device-independent (dvi) file named *book.dvi*. The fourth line generates a postscript (ps) file named *book.ps* from the dvi file. The fourth line generates a pdf file named *book.pdf* as a translation of the ps file. To initiate the task, we issue the command:

```
make manuscript
```

Postscript is a computer language like C++. A postscript printer understands this language and translates it into pen calls that draw images and high-quality fonts.

Software distributions

Suppose that a project named *rizogalo* and another project named *trahanas* are defined in the makefile. To execute both, we define in the same makefile the task:

```
all:
      rizogalo
      trahanas
```

and issue the command:

```
make all
```

The tasks `install` and `clean`, defining software installation and distillation procedures, are common in makefiles accompanying software distributions. To install software, we type

```
make install
```

To remove unneeded object files, we type

```
make clean
```

Problems

2.7.1. Define in a makefile a project called *clean* that removes all `.o` object files from the current directory using the Unix *rm* command (see Appendix A.)

2.7.2. Define in a makefile a project that generates the executable of a C++ program and then runs the executable.

2.8 Simple codes

We are in a position to write, compile, and execute a simple C++ code.

The following program declares, evaluates, and prints an integer:

```
#include <iostream>
using namespace std;
```

```
int main()
{
int year;
year = 1821;
cout << year << "\n";
return 0;
}
```

The output of the code is:

```
1821
```

The `cout` statement prints on the screen the value of the variable *year* using the `cout` function of the internal *iostream* library, and then moves the cursor to a new line instructed by the `\n` string. The syntax of these output commands will be discussed in detail in Chapter 3.

The fifth and sixth lines could have been consolidated into:

```
int year = 2006;
```

This compact writing is common among experienced programmers, though it is often stretched to the point of obfuscation. Albert Einstein once said: "Things should be made as simple as possible, but not any simpler."

The following C++ code contained in the file *addition.cc* evaluates the real variables a and b in double precision, adds them into the new variable c, and then prints the value of c on the screen along with a comforting message:

```
#include <iostream>
using namespace std;

int main()
{
double a=4;
double b=2;
double c;
c=a+b;
cout << c << "\n";
string message;
message = "peace on earth";
cout << message << "\n";
return 0;
}
```

The output of the code is:

```
6
peace on earth
```

The first `cout` statement prints the variable c on the screen using the *cout* function of the internal iostream library, and then moves the cursor to a new line instructed by the `endl` directive, as will be discussed in Chapter 3. The second `cout` statement performs a similar task.

Problems

2.8.1. Write a program that prints on the screen the name of a person that you most admire.

2.8.2. Investigate whether the following statement is permissible:

```
cout << int a=3;
```

2.8.3. Run the following program. Report and discuss the output.

```
#include <iostream>
using namespace std;

int main()
{
bool honest = true;
cout << honest << "\n";
cout << !honest << "\n";
return 0;
}
```

2.8.4. What is the output of the following program?

```
#include <iostream>
using namespace std;

int main()
{
string first_name;
string last_name;
first_name = "Mother";
last_name = "Theresa";
string name = first_name + " " + last_name;
cout << name << endl;
}
```

http://www.springer.com/978-0-387-68992-0

Introduction to C++ Programming and Graphics

Pozrikidis, C.

2007, XII, 372 p., Hardcover
ISBN: 978-0-387-68992-0