Programming Languages

Lecture notes

Edited by

Filipe Rodrigues
Raquel Bernardino

ISEG - Lisbon School of Economics and Management

Contents

Introduction

Programming knowledge is essential nowadays because our civilization heavily depends on software. Programming is present almost everywhere — from simple household appliances like washing machines to large objects such as ships, airplanes, and satellites. To program, a programming language is needed, that is, a coded language that can be understood by computers. There are several programming languages, such as C++, C, C#, Java, Python, etc. In this course, we will use the C++ language because, in addition to being one of the most widely used programming languages in the world and being available on almost every type of computer, it induces good programming practices, which are essential for programmers in early learning stages. With the knowledge gained in this course, you will later be able to easily learn other programming languages on your own.

To write programs in C++, we will use Qt Creator. In addition to allowing us to write and run our programs, Qt Creator also offers a number of other features, including graphical applications, which we will not explore in this course but that you can learn on your own later if you wish.

This course booklet contains some additional information about the C++ programming language that will not be taught or assessed in the Programming Languages course. The sections with this information are marked as **Supplementary Information**.

The Basics

Programming is the act of telling the computer what it must do to achieve a specific goal or to solve a particular problem. For this, it is necessary to write the instructions that the computer must execute in detail, using a language it can understand. This set of instructions is what we call a *program*. Being machines, computers do not have the ability to think, and therefore, all instructions must be written explicitly and in detail. To validate all the instructions written, the computer uses a *compiler*. The compiler's role is to check whether the computer can understand and execute all the instructions we wrote. This means that the compiler only detects writing errors (called *syntax* errors) in our code. It does not detect runtime errors—that is, it does not check whether the instructions we wrote actually do what we want the computer to do.

For the compiler to identify the end of each instruction, instructions usually end with a semicolon ";", although there are some exceptions. In QT Creator, compilation errors appear in red and must be corrected before the program can be executed. In addition, the compiler also issues *warnings* (shown in yellow) which should also be corrected, although not correcting them does not prevent the program from running. As we will see, some *warnings* can be ignored because they do not affect how the program runs; however, others may severely distort the program's behavior.

The compiler analyzes all the instructions we write, except those that are *comments* made by the programmer, and complains whenever something is wrong. *Comments* are programmer notes and appear after the symbol "//" or between the symbols "/* ... */". They are completely ignored by the compiler, so we can write anything we want in them. In QT Creator, comments appear in green.

The programs we will create use many functionalities that were previously defined by others, such as the sine function or the square root function. These commands are often implemented using many lines of code and are stored in specific C++ *packages*. These predefined commands (sine, square root, power, etc.) can be used directly in the programs we create without needing to know how they were originally implemented. However, for this to work, we must first inform the compiler where these commands are defined. To do so, we use the *#include* directive followed by the name of the package that contains the commands we want to use. For example, all mathematical functions are available in the *cmath* package, so if we want to use them in our program, we must begin by writing *#include <cmath>*. Gradually, we will learn which of the many available packages are useful to us. For now, just remember that all mathematical functions are in the *cmath* package and that the *iostream* package must be included whenever we want to read/write information to/from the screen, as it contains all the basic input and output commands we will use.

The header of a C++ program is called *preamble*, and this is where we include all external elements the program needs, such as packages.

```
//Preamble

int main(){
   //Comment: Write your code here

   /* This is also
       a
   comment */

   return 0;
}
```

When a C++ program is executed, the starting point is always the *main* function, so this function must always be present in the program. When this function is found, the program sequentially executes all lines of code from top to bottom (unless there are instructions that change the program's flow, as we will see in Chapter 2). The *main* function is delimited by curly brackets "{ }", within which we must write our code. The last instruction of the *main* function is "*return 0;*". This instruction was used in many operating systems to check whether the program ended successfully. Therefore, the code shown above is the skeleton of any C++ program.

The *main* function shown above contains only comments inside it, which will be completely ignored by the compiler. Thus, the program shown is essentially an empty program since no instructions are executed.

Chapter 1

Variables and Operators

Variables are the basic elements of any programming language and are essentially used to store information in the computer's memory.

Section 1.1 presents the different types of variables. Sections 1.2 and 1.3 explain how to write variables to the screen and how to read them from it, respectively. Section 1.4 introduces constant variables and, finally, Section 1.5 details the various types of existing operators.

1.1 Types of Variables

An object is a region of the computer's memory that can hold a value. A *variable* is a named object. Each variable is characterized by a name and a type. The name of a variable must follow some rules:

1. It must start with a letter (it may also start with an *underscore*, but this should be avoided).
2. It cannot match reserved words of the language, such as: *main*, *if*, *else*, *while*, *int*, *double*, *try*, etc. Reserved words are usually displayed in a different color in the Qt Creator.
3. It cannot contain spaces or characters other than numbers, letters, or the *underscore*.

The names we use should not be too long and should be as descriptive as possible to make the program easier to read. For example, if we want to create a variable to store the age of a person, the name *age* is probably the most descriptive one for that variable. It is important to note, however, that the same program cannot have two variables with the same name. If we need two variables to store two ages, we can, for example, use the names *age_1* and *age_2* or simply *age1* and *age2*. It is also important to note that C++ is case-sensitive, meaning that, for example, the names *age* and *Age* do not refer to the same variable. In the programs we write, we should not use variables with similar names to avoid confusion.

All variables used in a program must first be *declared* so that the computer can allocate memory space for the data type to be stored. To do this, we must write the variable's type followed by its name, that is,

```
int age;  //Declaration of a variable named age of type int
```

When a variable is declared, the value existing in the associated memory location - which is considered a "garbage value" - is immediately assigned to it. To avoid using garbage (unknown) values in the

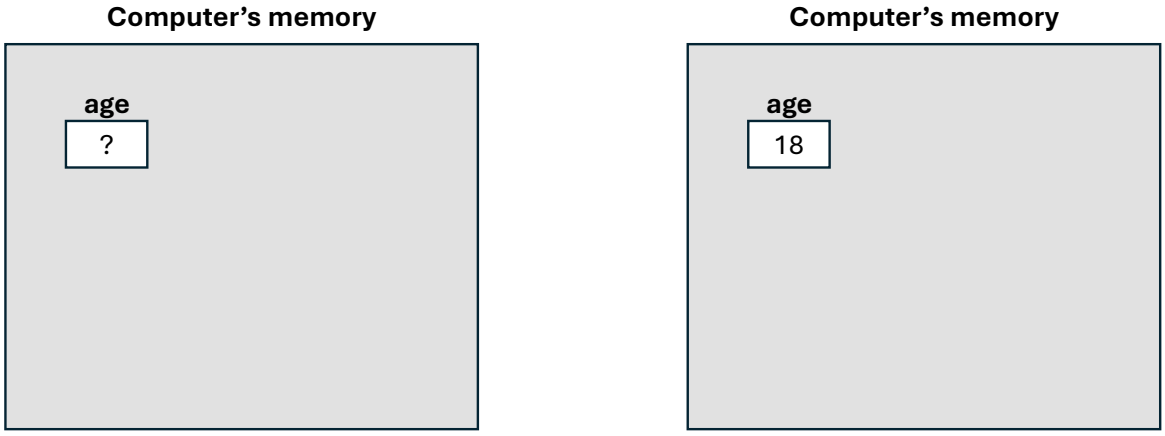
program, we should assign a value to the variables when they are created. This is called *initialization* and is illustrated below:

```
int age;
age = 18;  //Initialization of the variable age with the value 18
```

The declaration and initialization of a variable can be done in the same statement as follows:

```
int age = 18; //Initialization of the variable age with the value 18
```

Roughly speaking, creating variables is like creating boxes in the computer's memory where values of the defined type will be stored. Figure 1.1 contains a schematic representation of what happens in the computer's memory when a variable is declared (Figure 1.1a) and initialized (Figure 1.1b).



(a) Declaration. (b) Initialization.

Figure 1.1: Declaration *versus* initialization of a variable.

The type of a variable indicates the nature of the values it can assume. Table 1.1 contains examples of different data types.

Table 1.1: Examples of variable types.

	Types
Integer numbers	*short* *int* *long* *long long int*
Unsigned integer numbers	*size_t*
Decimal numbers	*float* *double*
Character	*char*
Text	*string*
Boolean value	*bool*

The first four data types are used to store integer values (positive or negative) and differ in the range of values they can hold, that is, in the amount of memory space they occupy. The *short* type can store

smaller integer values (fewer digits), while the *long long int* type can store larger integer values (more digits). In our programs, the *int* type is generally the most used because it can represent numbers sufficiently large. The *size_t* type represents non-negative integer numbers. For decimal numbers, we can use the *float* or *double* types. However, since the precision of the *double* type is greater than that of the *float* type — that is, the *double* type allows for storing more decimal places — we will only use it. A variable of type *char* can store any character, that is, a letter (without accent or cedilla), a digit (from 0 to 9), or a symbol (/, +, ;, $, etc.). A *string* is a data type that stores a sequence of characters, that is, text. Finally, the *bool* type can only take the logical values *true* or *false*, where *true* corresponds to the value 1 and *false* corresponds to the value 0. Boolean values are not stored as *true* and *false*, but rather as integers with the aforementioned correspondence.

All the variable types presented are *primitive* data types, except for the *string* type. The code below shows examples of variables of the different types mentioned above.

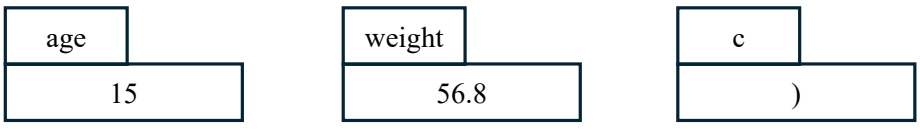
```
#include <iostream>
using namespace std;

int main(){

  int age = 18;              // Integer variable named age with value 18
  double weight = 56.8;      // Decimal variable named weight
  string name = "Pedro";     // Variable that stores a set of characters
  char c = ')';              // Variable that stores a character named c
  bool logic = true;         // Boolean variable named logic
  double height;             // Uninitialized variable named height

  return 0;
}
```

Six variables are declared in the code above, and the first five are also initialized. Note that characters (type *char*) are defined using single quotes, while text (type *string*) is defined using double quotes. Recall that it is not mandatory to initialize variables when they are declared, but it is good practice to do so because, if variables are declared without being initialized, they may assume arbitrary values, called *garbage values*. In the first line of the program, we have the instruction *#include <iostream>*, which is necessary when using variables of type *string*. In the second line, there is the instruction *using namespace std;*, which we will discuss later. As we have already seen, the instruction *int age = 18* creates a variable named "age" to store the integer number 18. Very informally, this process can be seen as creating a *box* and placing the number 18 inside it (see Figure 1.1b). The logic is the same for the remaining data types, except for *strings*, since they are not primitive data types.



The variables usually take different values during the execution of a program, but at each moment of the execution, they only store one value. It is important to remember that a variable is like a box that can only contain a single value inside it.

```
int main(){
   int x = 10;     //Variable initialized with the value 10
   x = 20;         //The value of the variable is changed to 20
   //...
   x = 30;         //The value of the variable is changed to 30
   return 0;
}
```

In the above code, the variable x is initialized with the value 10. Its value is then changed to 20 and later to 30, which will be its final value.

1.2 Writing Variables – Output

In the programs we build, it is often necessary to display information to the user, which is usually done on the screen. This is what we call *printing*. To print something, we use the *cout* command, which stands for console output, together with the operator <<.

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

int main(){

   int age = 15;
   string name = "Pedro";
   cout << "Output: \n";
   cout << name;
   cout << endl;
   cout << age;
   cout << "\n";

   return 0;
}
```

```
Output:
Pedro
15
```

To use the *cout* command, it is necessary to include the *iostream* package and write the second line of the program, i.e., `using namespace std;`. The instruction "$\backslash n$" is a control command used to make a line break on the information displayed on the screen, that is, to make a paragraph. The instruction *endl* means *end of line* and can be used alternatively to "$\backslash n$". When executed, this program first declares and initializes the variables *age* and *name*. After that, it encounters the first *cout* and writes the message "Output:" on the screen. On that same line, it finds the controller $\backslash n$, which introduces a line break in the output. In the next line of code, the program accesses the variable (or box) named *name* and writes what is stored inside it, which in this case is "Pedro". Upon reaching the next line of code, it simply introduces a line break because it found the *endl* controller. After that, the program accesses the variable named *age* and writes what is stored there. The last *cout* only introduces one more line break. In practice, the five previous *cout* commands can (and should) be written with a single instruction, which simplifies the writing, as follows:

```
cout << "Output: \n" << name << endl << age << "\n";
```

which leads to the same output.

1.3 Assigning Values to Variables – Input

To assign values to a variable, we can simply use the assignment operator `=`. For example, $x = 5$. The function of the `=` operator is to assign the value on the right-hand side to the variable on the left-hand side. For instance, the instruction $x = 5$ corresponds to $x \leftarrow 5$, that is, it assigns the value 5 to the variable x. The operator `=` is used to assign values to variables when we know the value to be assigned at the moment the code is written. However, in many situations, the value of the variables is not known beforehand; it is only defined by the user later on. For these cases, we should use the *cin* command, which stands for *console input*, together with the operator `>>`, to read information provided by the user.

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

int main(){

   int age;
   cout << "Enter age: ";
   cin >> age;

   cout << "The age is: " << age << "\n";

   return 0;
}
```

To use the *cin* command, it is also necessary to include the *iostream* package and write the second line of the program. This program begins by declaring a variable of type *int* with the name *age*. Then, it writes the text *"Enter age: "* on the screen. Next, the program moves to the next line and will wait until the user enters an integer. When the user provides the number, it is stored in the variable *age*. Finally, the program writes to the screen *"The age is: "*, retrieves the value stored in the variable *age*, prints it on the screen, and ends with a line break.

Although it is good practice to initialize variables, it is not necessary to do so in cases where their value is requested from the user (almost) immediately after declaration, as in the previous example. The first three code lines inside the *main* function of the previous example are always used when we want to request the value of a variable from the user. Those three lines exemplify the process called *reading a variable*.

The operators `>>` and `<<` used, respectively, in the instructions *cin* and *cout* indicate movement. More precisely, when we write *cin* `>>` x, we are *sending* what was typed on the screen (left side) to the variable x on the right side. On the other hand, when we write *cout* `<<` x, we are *sending* what is on the right side (the value of the variable x) to be displayed on the screen (left side).

1.4 Constant Variables

As we have seen earlier, the value of a variable can be successively changed during the execution of a program. However, in some situations, it may be useful to use variables with values that we do not want to change. These types of variables are called *constants* and are defined using the reserved word *const*. A good example is the case of π, which can be defined as *const double pi = 3.141592*. This not only allows us to always use the variable *pi* throughout the program instead of writing the value 3.141592, but it also prevents the value of the variable *pi* from being changed. That is, it will not be possible to make a new assignment to it, such as *pi = 3.14*. This example is illustrated in the next code excerpt, which does not compile because it attempts to change the value of a variable defined as a constant.

```
int main(){
    const double pi = 3.141592;
    pi = 3.14;    //ERROR!
    return 0;
}
```

1.5 Operators

C++ contains operators of different types, namely simple arithmetic operators, compound arithmetic operators, relational operators, and logical operators.

1.5.1 Arithmetic operators

The simple arithmetic operators are those we already know from mathematics and are shown in the table below, where the values presented in the "Result" column correspond to the case in which *a=13* and *b=5*.

Table 1.2: Simple arithmetic operators.

Operator	Name	Example	Result
+	Sum	*a+b*	18
-	Subtraction	*a-b*	8
*	Multiplication	*a*b*	65
/	Integer or decimal division	*a/b*	2 or 2.6
%	Remainder of integer division (Modulo)	*a%b*	3

The result of the "/" operator is the integer division whenever both operands are of an integer data type. If one of the operands is of a decimal type, the result of the "/" operator is decimal division. To obtain decimal division between two integer-type variables, it is first necessary to convert one of them to a decimal data type before performing the division. This process is called *cast*, and one way to do it is to write the desired data type in parentheses before the variable to be converted. That is, if *a* and *b* are variables of type *int*, decimal division can be obtained with the instruction *(double) a/b*. This means that the program first converts the variable *a* to type *double* and then performs the division between a *double* and an *int*, producing a decimal result.

The arithmetic operators presented in the previous table are used between operands of numeric types. However, the "+" operator can also be used with *strings*, functioning as a concatenation (joining) operator.

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

int main(){
    string a = "I am ";
    int c = 18;
    string b = " years old.";

    string sentence = a + to_string(c) + b + " I am young!";
    cout << sentence;

    return 0;
}
```

In the above example, the variable *sentence* of type *string*, which results from the concatenation of several *strings* and one integer, is printed on the screen. Variables of type *string* can be concatenated directly; however, in the case of numeric data types, it is necessary to use the *to_string* function. The purpose of the *to_string* function is to convert a numeric value into a *string*, which can then be directly concatenated with other *strings*. In the above example, the output printed on the screen, which is the value of the variable `sentence`, is "*I am 18 years old. I am young!*".

Compound arithmetic operators allow us to simplify the writing of instructions. For example, writing $a = a + 3$ is the same as writing $a \mathrel{+}= 3$. To use these operators, it is necessary to have a good understanding of the meaning and functioning of the `=` operator explained earlier. Recall that this operator assigns the value on its right-hand side to the variable on its left-hand side. Therefore, when we write $a = a + 3$, we are not saying that the right side is equal to the left side; such an interpretation would make no sense mathematically. The meaning of the expression $a = a + 3$ is $a \leftarrow (a + 3)$. Suppose the value of a is 6. When the program reaches the instruction $a = a + 3$, it first evaluates the right-hand side expression $a + 3$, which equals 9. Then, it assigns the value 9 to the variable a. From that point on, the value of a is 9. The table below presents the arithmetic assignment operators and the resulting value of the variable a at the end of the operations, considering the initial values $a = 6$ and $b = 2$.

Table 1.3: Compound arithmetic operators (considering a=6 and b=2).

Operator	**Name**	**Example**	**Meaning**	**Value of *a***
`+=`	Addition/assignment	*a+=b*	*a=a+b*	8
`-=`	Subtraction/assignment	*a-=b*	*a=a-b*	4
`*=`	Multiplication/assignment	*a*=b*	*a=a*b*	12
`/=`	Division/assignment	*a/=b*	*a=a/b*	3
`%=`	Modulus/assignment	*a%=b*	*a=a%b*	0
`++`	Increment	*a++*	*a=a+1*	7
`--`	Decrement	*a--*	*a=a-1*	5

These operators are applied to numeric data types; however, the `+=` operator can also be applied to *strings*. In this case, it functions as a concatenation plus assignment operator, concatenating the value on the right-hand side to the value on the left-hand side. In the example below, the final value of the variable b is not changed; it remains "BB". The value of the variable a (printed on the screen) will be "$AABB$". That is, since a += b is equivalent to $a = a + b$, the value of b is concatenated to the initial value of a, and the result is stored in the variable a.

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

int main(){
    string a = "AA";
    string b = "BB";
    a += b
    cout << a;
    return 0;
}
```

The increment operator a++ is equivalent to writing a+=1, which is also equivalent to $a = a + 1$. Increment and decrement operators have the particularity of being used either as a prefix (++a) or as a suffix (a++). When used in isolation, their meaning is exactly the same. However, in operations where the result of the increment or decrement is evaluated within another expression, the results can differ. In the case of the prefix increment operator (++a), the value of the variable is incremented first and then returned. That is, the variable is incremented before the expression is evaluated, which means that the evaluation of the expression will be done by considering the incremented value. In the case of the suffix increment operator (a++), the value of the variable is returned first and then incremented. That is, the variable is incremented only after the expression is evaluated. Consider the example below:

Table 1.4: Difference between prefix and suffix increment operators (a=3 and b=3).

Example	**Final value of *a***	**Final value of *b***
a=++b	4	4
a=b++	3	4

When used in isolation, we should prefer the prefix increment operator (++a) because, as we will see later, it is more efficient than the postfix increment operator (a++).

1.5.2 Relational Operators

Relational operators are used to compare two expressions. The result of this comparison is a value of type *bool*, which can be *true* (if the comparison is true) or *false* (otherwise).

These operators are fundamental for the next section. For now, it is important to highlight the operator `==`, which is completely different from the assignment operator `=` used earlier. The operator `==` checks whether the value on its right-hand side is equal to the value on its left-hand side, returning *true* or *false*. For example, the result of the comparison *5==8/2* is *false* because 5 is not equal to *8/2 (=4)*. The operator `=`, on the other hand, is used to assign the value on the right-hand side to the variable on the left-hand side and is not used for comparisons.

Table 1.5: Relational operators.

Operator	Meaning
<	lower than
>	greater than
<=	lower than or equal to
>=	greater than or equal to
==	equal to
!=	different from

1.5.3 Logical Operators

Logical operators are used to negate and combine expressions. Therefore, the result of operations with logical operators is also *true* or *false*.

Table 1.6: Logical operators.

Operator	Meaning
&& or *and*	conjunction (and)
|| or *or*	disjunction (or)
!	logical not

The ! operator placed to the left of an expression inverts its logical value. That is, if the expression is true, it becomes false, and vice versa. For example, consider three integer variables a=*5*, b=*3*, and c=*2*. Then, we have:

$$\begin{aligned} b > a &\longrightarrow false \\ !(b > a) &\longrightarrow true \\ !(b > a) \;\&\&\; c == a - b \;||\; c > b &\longrightarrow true \\ (\; !(b > a) \;\; or \;\; c == a - b \;) \;\; and \;\; c > b &\longrightarrow false \\ b > a \;||\; c == a - b \;\&\&\; b > c &\longrightarrow true \end{aligned}$$

Note that the last expression is equivalent to $b > a \;||\; (\; c == a - b \;\&\&\; b > c \;)$ because the conjunction (read "and") has higher precedence than the disjunction (read "or").

1.5.4 Ternary Operator (Supplementary Information)

The ternary or conditional operator evaluates an expression and returns different values depending on the result of that evaluation. This operator will not be taught in class and is included here only as supplementary information. The syntax of this operator is as follows:

(<*condition*> ? <*result1*> : <*result2*>)

If the <*condition*> is true, the operator will return <*result1*>. Otherwise, it returns <*result2*>. For example, when we write

```
int x;
x = (7==5 ? 4 : 3);
```

variable x will have value 3 because the condition 7==5 is false.

Chapter 2

Control Structures

Control structures are essential in any programming language and are divided into conditional and loop structures. Conditional control structures are primarily used to execute specific instructions depending on whether certain conditions are met. In other words, they allow the program to follow different paths. In contrast, loop control structures are associated with the repetition of instructions/processes.

Conditional control structures are presented in Section 2.1. Section 2.2 details the importance of using brackets and indentation. Finally, loop control structures are presented in Section 2.3.

2.1 Conditional Control Structures

Three different conditional structures will be presented, namely the *if* structure, the *if else* structure, and nested conditional structures.

2.1.1 The *if* Structure

The simplest conditional control structure in programming is the *if* structure. Its general syntax is:

```
if ( Condition ) {
   Instruction block
}
```

An *if* is characterized by a condition and a block of instructions. Instruction blocks are defined using brackets and contain several instructions. The condition evaluates to *true* or *false*, and it can therefore be a boolean variable or a logical expression that, in most cases, involves the relational and logical operators presented in the previous chapter. The instruction block of the *if* structure will only be executed if the condition is true.

The following example contains a snippet of code where the *if* structure is used.

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

int main(){
    int a = 1;
    int b;
    cout << "Enter the value of b: ";
    cin >> b;

    if ( b > 10 && b % 3 == 0 ) {
        ++a;
    }

    return 0;
}
```

In this example, the program begins by declaring two variables, `a` and `b`, and initializing the first with value 1. Then, the user is prompted to insert the value of `b`. When reaching the *if* statement, the program evaluates the logical value of the condition `b>10 && b%3==0`. If the result of this evaluation is *true*, the program will enter the *if* block (which in this case contains only one instruction) and will increment the value of the variable `a` by one.

Suppose the user inserts 20 for `b`. Although 20 is greater than 10, the remainder of 20 divided by 3 is not zero, so the logical result of the *if* condition is *false*. In this case, the program will not execute the instructions associated with the *if* and will immediately proceed to the final *return*, and thus the final value of the variable `a` is 1. Now, suppose the user inserts 15. In this case, since $15 > 10$ and the remainder of 15 divided by 3 is zero, the *if* condition is true, and so the program will execute the instructions in the *if* block. As such, the value of the variable a at the end of the program will be 2.

In this example, two aspects are particularly important. First, the use of the equality operator `==`, which is justified by the fact that a comparison (not an assignment) is being made. Second, the meaning of the second part of the logical expression, i.e., `b%3==0`. Checking whether the remainder of dividing b by 3 is zero is equivalent to checking whether b is a multiple of 3, which is the standard way of defining conditions like "is a multiple of".

2.1.2 The *if else* Structure

The *if else* structure has two instruction blocks, allowing us to define actions in case the *if* condition evaluates to false. Thus, Instruction Block 1 will be executed if the *if* condition is true, while Instruction Block 2 will be executed otherwise, that is, if the condition is false. The general syntax of an *if else* is:

```
if ( Condition ) {
   Instruction Block 1
} else{
   Instruction Block 2
}
```

Note that *else* does not require a condition because it implicitly represents the negation of the condition written in the *if*. Consider the following example:

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.
4.     if ( b % 3 == 0 or b % 2 == 0 ) {
5.         a = 10;
6.         c += a;
7.     }else{
8.          a = 20;
9.          c -= a;
10.    }
11.    return 0;
12.}
```

The condition associated with the *if* can be read as "*b is a multiple of 3 or 2*". Since `b`=19, the condition is false, and so the program jumps immediately from line 4 to line 7 — the *else* line — and then executes all instructions within that block. As a result, the program sets the value of `a` to 20 and then updates the value of `c` to 3-20, which equals -17. At the end of the program, we have `a`=20, `b`=19, and `c`=−17. In this example, the condition implicitly associated with the *else* is "*b is not a multiple of 3 nor 2*", i.e., `b % 3 != 0 and b % 2 != 0`.

As a final remark, it is important to note that whenever we use an *else*, it must be associated with an *if*. However, the reverse is not required, i.e., we can have an *if* without any associated *else*, as we saw in the previous section.

2.1.3 Nested Conditional Structures

The *if else* structure allows the program to follow two distinct paths depending on the evaluation of a condition. However, there are situations where more than two paths are possible. In such cases, we can use *nested conditional* structures.

The general syntax of the compact form of nested conditional structures is shown on the right. On the left, we have the expanded version of the same structure, using multiple *if else* structures:

```
if ( Condition 1 ) {
   Instruction Block 1
} else  {
    if ( Condition 2 ){
          Instruction Block 2
    } else {
          if ( Condition 3 ){
              Instruction Block 3
          } else {
              Instruction Block 4
          }
    }
}
```

```
if ( Condition 1 ) {
   Instruction Block 1
} else if ( Condition 2 ){
   Instruction Block 2
} else if ( Condition 3 ){
   Instruction Block 3
} else {
   Instruction Block 4
}
```

Consider the compact form of nested conditional structures. If Condition 1 is true, Instruction Block 1 is executed. If Condition 2 is true and Condition 1 is false, then Instruction Block 2 is executed.

Instruction Block 3 is only executed if Condition 3 is true and both Conditions 1 and 2 are false. Instruction Block 4 is executed only if all Conditions 1, 2, and 3 are false. It is important to emphasize that in this structure, one and only one instruction block is executed, even if more than one condition is true. If two or more conditions are true, the only instruction block that is executed is the first one encountered. For example, if both Condition 2 and Condition 3 are *true*, the instruction block that is executed is Instruction Block 2.

To clarify the differences between the conditional control structures presented so far, consider the following example. Suppose that the unit price of a given product depends on the quantity purchased, and we want to write a program that, given the number of units to purchase, calculates the final price to be paid. If the unit price of the product is determined according to the table below:

Quantity	<50	$[50, 99[$	$[100, 150[$	≥ 150
Price	5	4	3.5	3.3

then the following code, although it does not use nested conditional structures, achieves the desired result.

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

int main(){
    int qt, price;
    cout << "Quantity: ";
    cin >> qt;

    if ( qt < 50 ) {
        price = 5 * qt;
    }
    if ( qt >= 50 and qt < 99 ) {
        price = 4 * qt;
    }
    if ( qt >= 100 and qt < 150 ) {
        price = 3.5 * qt;
    }
    if ( qt >= 150 ) {
        price = 3.3 * qt;
    }

    cout << "Final price: " << price;
    return 0;
}
```

The code above is composed of four independent *if* structures. The same code can be written using nested conditional structures, as shown below.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int qt, price;
6.     cout << "Quantity: ";
7.     cin >> qt;
8.
9.     if ( qt < 50 ) {
10.        price = 5 * qt;
11.    }else{
12.         if ( qt < 99 ) {
13.            price = 4 * qt;
14.         }else{
15.              if ( qt < 150 ) {
16.                 price = 3.5 * qt;
17.              }else{
18.                 price = 3.3 * qt;
19.              }
20.         }
21.    }
22.
23.    cout<<"Final price: "<<price;
24.    return 0;
25. }
```

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int qt, price;
6.     cout << "Quantity: ";
7.     cin >> qt;
8.
9.     if ( qt < 50 ) {
10.        price = 5 * qt;
11.    }else if( qt < 99 ) {
12.        price = 4 * qt;
13.    }else if ( qt < 150 )
14.        price = 3.5 * qt;
15.    }else{
16.        price = 3.3 * qt;
17.    }
18.
19.    cout<<"Final price: "<<price;
20.    return 0;
21. }
```

In the first implementation, there is a main control structure that starts at line 9 and ends at line 21. The *else* of this structure (line 11) contains in its instruction block a new *if* that starts at line 12 and ends at line 20. This new *if*, in turn, also includes another *if* within the instruction block of its *else*, which begins at line 15 and ends at line 19. However, it is important to highlight that each *if* can have at most one associated *else*. The conditions of the *if*s shown here may seem incomplete compared to those in the previous code, but they are in fact correct. We will analyze different cases to better understand how nested conditional structures work. Consider the code on the left.

1. Suppose the quantity *qt* entered by the user at line 7 is 30. When the program reaches line 9, it checks whether $qt<50$, which is indeed true. Therefore, the program will enter the first *if*, execute the instruction at line 10, and then immediately jump to line 21 (end of the first *if*). In the end, we will have $price=150$. Note that since the condition of the first *if* is true, the program does not enter the associated *else* (lines 11–20).

2. Suppose the quantity *qt* entered by the user is 60. When the program reaches the first *if* (line 9), it verifies that the condition $qt<50$ is false, so it proceeds immediately to the associated *else* block (line 11). Within that block, the program first evaluates the condition of the *if* at line 12, i.e., it checks whether $qt<99$. Since the condition is true, the program enters that *if* and executes the instruction at line 13. It then proceeds to line 20 and then to line 21. Note that since the program entered the *if* at line 12, it does not enter the *else* at line 14.

3. Suppose the quantity *qt* entered by the user is 200. As in the previous case, the program will enter the *else* at line 11 and execute the instructions in that block. The condition in the *if* at line 12 is false, so the program enters the *else* at line 14. When it reaches line 15, the program evaluates the condition *qt<150*, which is false, so it enters the *else* at line 17 and then executes the instruction at line 18. It proceeds to line 19, then to line 20, and finally to line 21, without performing any further actions.

Based on this example, we can understand, for example, why we can simply write `qt<150` at line 15 instead of `qt<150 and qt>=100`. This works because if the program reaches line 15, it must have already entered the *else* at line 11 (i.e., *qt≥50*) and the *else* at line 14 (i.e., *qt≥100*).

The second code snippet presented above uses the compact form of nested conditional structures. Recall that when using a structure of this type, it is important to keep in mind that the program will enter only one instruction block: either the block associated with the *if*, one of the blocks associated with the *else if*, or the block associated with the *else*. Once the program enters one of these blocks—whichever it may be—it executes the instructions in that block and then immediately jumps to the end of the control structure (line 17 in this example).

When using nested control structures, it is essential to always keep in mind how the *if else* structure works—specifically, that the program either executes the instructions in the block associated with the *if* or those in the block associated with the *else*, but never both.

2.2 Use of Brackets and Indentation

The correct use of brackets and code indentation are two very important aspects of programming. On one hand, brackets are used to delimit blocks of instructions, as we saw in the previous section, and incorrect placement can lead to syntax errors or runtime errors. On the other hand, code indentation is completely ignored by the compiler and therefore does not affect the program's behavior. Indenting the code serves only to greatly improve its readability, making it clear which instructions belong to which blocks. Below, the same code is presented in an indented form (left side) and in a non-indented form (right side). This example clearly illustrates the importance of indentation, as in the first case, we can easily see where each instruction block begins and ends.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main(){
5.      int qt, price;
6.      cout << "Quantity: ";
7.      cin >> qt;
8.
9.      if ( qt < 50 ) {
10.          price = 5 * qt;
11.      }else{
12.          if ( qt < 99 ) {
13.              price = 4 * qt;
14.          }else{
15.              if ( qt < 150 ) {
16.                  price = 3.5 * qt;
17.              }else{
18.                  price = 3.3 * qt;
19.              }
20.          }
21.      }
22.
23.      cout<<"Final price: "<<price;
24.      return 0;
25.  }
```

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main(){
5.  int qt, price;
6.  cout << "Quantity: ";
7.  cin >> qt;
8.
9.  if ( qt < 50 ) {
10. price = 5 * qt;
11. }else{
12. if ( qt < 99 ) {
13. price = 4 * qt;
14. }else{
15. if ( qt < 150 ) {
16. price = 3.5 * qt;
17. }else{
18. price = 3.3 * qt;
19. }
20. }
21. }
22.
23. cout<<"Final price: "<<price;
24. return 0;
25. }
```

In Qt Creator, we can automatically indent the code we write by pressing *Control* + *A* (to select everything) followed by *Control* + *I* (to indent). This automatic indentation helps us understand how the compiler interprets the code we have written and allows us to verify whether that interpretation aligns with our intentions.

The use of brackets is essential for delimiting instruction blocks. However, when an instruction block contains only a single instruction, the brackets may be omitted. In the example below, there is only one instruction associated with the *if* on line 4, so the bracket on line 4 and the first bracket on line 6 can be omitted. The brackets associated with the *if* on line 7 cannot be removed because the block contains more than one instruction (two, in this case). However, the second bracket on line 10 and the bracket on line 12 can be removed since the block they delimit contains only one instruction.

The brackets we mentioned can be removed from the code in order to make it more compact. However, the second bracket on line 6 and the bracket on line 13 can also be removed. This is because the first *else* block also contains, in fact, only a single instruction inside it—an *if else* instruction which, although spread over several lines, is considered a single instruction. This kind of situation can cause some confusion in the early stages of programming, so it is recommended to keep the brackets in such cases. Once again, it is important to remember that in Qt Creator, we can automatically indent our code, making it easier to spot bracket placement/omission errors as well as clearly identifying which instructions belong to which blocks.

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.
4.     if ( b % 3 == 0  ) {
5.         c += a;
6.     }else{
7.         if(c>2){
8.             a = 20;
9.             c -= a;
10.         }else{
11.             a = 20;
12.         }
13.     }
14.     return 0;
18. }
```

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.
4.     if ( b % 3 == 0  )
5.         c += a;
6.     else
7.         if(c>2){
8.             a = 20;
9.             c -= a;
10.         }else
11.             a = 20;
12.
13.
14.     return 0;
18. }
```

The next example reinforces the importance of using brackets and indentation and illustrates a property of the *if else* structure that has not yet been shown.

```
1. int main(){
2. int a = 1, b = 19, c = 3;
3. if (b % 3 == 0)
4. if (c>2)
5. a = 30;
6. else
7. a = 20;
8. return 0;
9. }
10.
11.
```

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.     if (b % 3 == 0) {
4.         if (c>2){
5.             a = 30;
6.         }
7.     }else{
8.         a = 20;
9.     }
10.    return 0;
11. }
```

The first code excerpt contains no brackets and is not indented. Additionally, it includes two *if*s and only one *else*, which makes it difficult to determine to which *if* the *else* is associated with, creating ambiguity. In such ambiguous cases, the *else* is paired with the last *if* within the same instruction block, that is, the *if* on line 4. The second code excerpt includes brackets and is indented, making it clear that in this case the *else* is associated with the first *if*, which begins on line 3, and that the second *if* (line 4) is contained within its instruction block.

2.3 Loop Control Structures

Loop structures allow the execution of a set of instructions repeatedly while a given condition is satisfied. The C++ programming language offers three loop control structures: *while*, *do-while*, and *for*. As mentioned before, a C++ program always begins by executing the *main* function and then proceeds

sequentially line by line from top to bottom, unless there are instructions that alter this flow. Loop instructions are the first type of instructions we study that change the normal execution sequence of a program.

2.3.1 The *while* Structure

The *while* structure is the simplest of the three cyclic structures available in C++ and has a structure similar to that of the *if* statement. The general syntax of a *while* loop is:

```
while (Condition){
     Instruction Block
}
```

When a *while* structure is executed, the program first checks whether the `Condition` is true. If it is, the Instruction Block is executed. Unlike the *if* structure, after executing the Instruction Block, the program returns to the top of the *while* structure and evaluates the `Condition` again. If the `Condition` is still evaluated as *true*, the Instruction Block is executed again. This process repeats until the `Condition` is evaluated as *false*. When that happens, the program exits the loop and continues its execution with the line of code immediately following the *while* structure. Thus, a *while* loop can be read as: "While the condition is true, execute the instruction block".

Consider the following example where a *while* structure is used to print the numbers from 1 to 10 on the computer screen:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int i = 1;
6.     while( i <= 10 ){
7.         cout << i << " ";
8.         ++i;
9.     }
10.
11.    return 0;
12. }
```

We will now analyze in detail what the program is doing. First, the variable `i` is declared and initialized to 1, since that is the first number we want to print. Then, because the condition `i = 1 <= 10` is true, the instruction block of the *while* loop (lines 7–8) is executed. The first instruction in the block prints the value 1 (the current value of the variable `i`) and a space to the screen. The second instruction increments the value of the variable `i` to 2. Now, the program returns to line 6, and the condition is evaluated again. Since `2 <= 10` is true, the instruction block is executed again, and the value 2 followed by a space is printed to the screen. The loop continues to execute until the variable `i` has the value 11, at which point the condition `11 <= 10` is false, and the instruction block of the *while* is not executed. The program then proceeds to line 10.

In the previous example, to implement a *while* loop, we needed:

1. to define and initialize a loop control variable (variable *i*);

2. to have a condition or stopping criterion (`i <= 10`);

3. to update the loop control variable (`++i)`.

These three components are always, in some way, present in a *while* loop, and failing to include them can result in coding errors.

Failing to initialize the loop control variable with an appropriate value may result in the instruction block associated with the *while* loop never being executed. For example, if in the previous example we had `int i = 15;` on line 5, the *while* condition would be evaluated as false, and the program would jump directly to line 10. The loop control variable should be declared and initialized outside the *while* structure. Variables declared inside instruction blocks, particularly inside the *while* block, do not exist outside those blocks. This is called the *scope* of the variable, that is, the region of the program where the variable is recognized.

When the loop control variable is not updated, we may encounter an *infinite loop* in which the *while* condition always remains true. This means the program will execute the instruction block indefinitely, never exiting the loop, and possibly ending with a system *crash*. If, in the previous example, we had omitted the instruction on line 8 (`++i;`), the variable `i` would always remain at the value 1, making the condition `i <= 10` perpetually true. We would thus have an infinite loop.

2.3.2 The *do-while* Structure

The general syntax of *do-while* is:

```
do{
   Instruction Block
}while (Condition);
```

A *do-while* loop can be interpreted as: "Do what is in the instruction block while the given condition is true". When the program encounters a *do-while* loop, it begins by immediately executing the instruction block inside it without checking any condition (unlike the *while* structure). After executing the Instruction Block, the program evaluates the logical value of the `Condition`. If the condition evaluates to true, the program re-executes all instructions within the loop's instruction block and evaluates the `Condition` again. This process repeats as long as the *while* condition remains true, allowing for multiple iterations. As soon as the condition evaluates to false, the program immediately exits the loop and proceeds to the line of code immediately below it.

The main difference between the *do-while* structure and the *while* structure is that, in the first iteration of the *do-while* loop, the instruction block is always executed since the condition is only evaluated afterward. Therefore, the instruction block of the *do-while* structure is always executed at least once, whereas the instruction block of the *while* structure may never be executed.

Consider the following two examples.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int n = 3;
6.
7.     int i = 1;
8.     do{
9.         cout << i << " ";
10.        ++i;
11.    }while( i <= n );
12.
13.    return 0;
14. }
```

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int n;
6.     int count = 0;
7.
8.     do{
9.         cout << "Value: ";
10.        cin >> n;
11.        if ( n > 0)
12.            ++count;
13.    }while( n > 0 );
14.
15.    cout << "Total: " << count;
16.    return 0;
17. }
```

When executed, the first program (left) begins by declaring and initializing the integer variable `n` with the value 3. Upon reaching line 7, the program declares and initializes a new variable `i` with the value 1. This variable is used in the condition of the *do-while* loop, and it is based on its value that the loop will either continue executing or be interrupted. Therefore, the variable `i` in this program is the loop control variable.

After line 7, the program proceeds to line 8 and then to line 9, where it prints the value of the variable `i` (which is 1) and a space to the screen. Next, it moves to line 10 where it increments the value of `i` by one, i.e., `i = 2`. The program then goes to line 11 and checks whether the condition associated with the *while*, that is, `i <= n`, is satisfied. Since it is, the program returns to line 8 to execute all instructions within the *do-while* block again. When it reaches line 9, the program prints the new value of the variable `i` (which is now 2) and a space, then increments the value of the variable `i` again in line 10. The condition associated with the *while* is evaluated once more, and the result is still `true` because $i = 3 \leq 3 = n$. As a result, the program once again executes the instruction block of the *do-while*, meaning it goes back to line 8 and immediately to line 9 to print the value 3 on the screen. Then, in line 10, it increments the value of the variable i by one, making it 4.

Next, the *while* condition is evaluated, and since it is now false (because `i = 4`), the program exits the loop and proceeds to line 12 and then to line 13, where the program ends. Since all of the previously mentioned instructions are executed very quickly, the user will only see the final result displayed on the screen, which in this case is: "1 2 3 ". Through this analysis, we understand that the goal of the first program is, in fact, to print the first n natural numbers.

To reinforce the importance of the `++i` instruction on line 9, let us try running the program without it. The variable `i` is initialized with the value 1 on line 7. If line 10 does not exist in the program, the value of the variable `i` will never change. This means that the condition `i <= n` will always be true, and therefore the program will *indefinitely* execute the instruction block of the *do-while*, continuously printing the value of i (which is 1) to the screen, thus entering an infinite loop.

Now, consider the second program. The objective of this program is to repeatedly ask the user to enter positive integers until a non-positive integer is entered. At the end of the program, a message is displayed with the total number of positive numbers entered by the user. In each iteration of the loop,

the program asks the user to input a value (lines 9 and 10). If the entered value is positive (line 11), the program increments the variable `count` by one (line 12). This variable is used to count the number of positive values entered.

After executing the instruction block, the program evaluates the *while* condition, that is, it checks whether the last number entered by the user is positive. If so, the program executes all instructions in the instruction block again. Otherwise, the program exits the loop, proceeds to line 15, and displays the final message.

There are a few important points to highlight in this program. First, the required initialization of the variable `count` on line 6. Since on line 12 the variable `count` is being incremented - meaning it assumes its previous value plus one — it is essential that this variable has a well-defined initial value. In this case, that value is zero because initially (when the program reaches line 6), no positive numbers have been entered yet.

The second point is the use of the `if` on line 11. Is this *if* really necessary? Since the goal of the program is to count how many positive numbers the user has entered, using this *if* is fundamental. If it didn't exist, then when the user entered a negative or zero value to stop the loop, that value would also be counted by the `count` variable, because the increment would happen before the *while* condition is evaluated.

A final observation is that the variable `count` must be declared outside the loop, meaning its scope is the *main* function. If the variable were declared inside the loop, not only would the program fail to produce the intended result, but it would also be impossible to access it on line 15 to print its value.

2.3.3 The *for* Structure

The last control structure we will present is the *for* structure. Any *for* loop can be rewritten as a *while* (or *do-while*) loop, and vice versa. The general syntax of a *for* loop is:

```
for (Initialization; Condition; Increment){
     Instruction Block
}
```

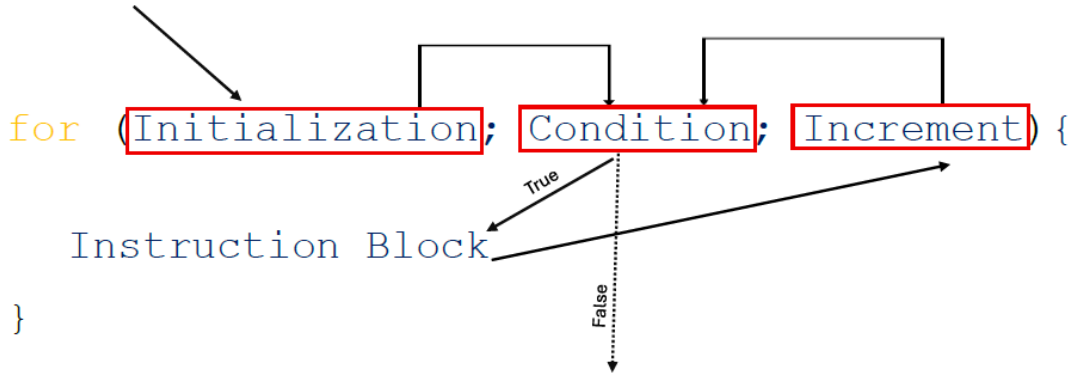
The *for* loop combines the declaration and initialization of the control variable, the loop's execution condition, and the increment of the control variable in a single place.

Consider the following example, which shows a program that prints the numbers from 1 to 10 on the screen using both a *while* loop and a *for* loop:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int i = 1;
6.      while( i <= 10 ){
7.         cout << i << " ";
8.         ++i;
9.     }
10.
11.    return 0;
12. }
```

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.      for(int i = 1; i <= 10; ++i){
6.         cout << i << " ";
7.     }
8.
9.    return 0;
10. }
```

As can be seen in this example, lines 5, 6, and 8 of the first program are, in a way, combined into a single line (line 5) in the second program. Consider now the flow diagram followed by a *for* loop.



When encountering a *for* loop, the program begins by initializing the loop control variable and then evaluates the logical expression that defines the condition. As long as that condition is satisfied, the program, in this order, executes the instruction block of the *for* loop, increments the loop control variable, and re-evaluates the condition. This process is repeated until the condition is evaluated as false.

Three different loop structures have been presented, all of which can be converted into one another. However, in certain situations, the use of one type of loop is more appropriate than the others. The *while* and *do-while* loops are generally used when the number of iterations is not known in advance. For example, in a program that repeatedly asks the user for positive values, it is not known in advance how many values will be entered, so in this case, it makes more sense to use a *while* or *do-while* loop. In situations where the number of iterations is known in advance, it is preferable to use the *for* loop, since all the loop-related information (initialization of control variable, condition, and increment) is presented in its first line. For instance, if the goal is to sum the first 30 natural numbers, it is already known that 30 iterations will be required, making the *for* loop the recommended choice.

2.3.4 Nested Loops

Often, using a single loop is not sufficient to implement certain algorithms, and it becomes necessary to use *nested loops*. That is, loops that contain other loops within their instruction block. Consider the following example:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.    int n = 5;
6.
7.    for(int i = 1; i < n; ++i)
8.       for(int j = i + 1; j < n; ++j)
9.          cout<<"("<<i<<","<<j<<") ";
10.
11.   return 0;
12. }
```

```
i = 1
    j = 2
        Writes on the screen "(1,2)  "
    j = 3
        Writes on the screen "(1,3)  "
    j = 4
        Writes on the screen "(1,4)  "
    j = 5 (End of 2nd loop)
i = 2
    j = 3
        Writes on the screen "(2,3)  "
    j = 4
        Writes on the screen "(2,4)  "
    j = 5 (End of 2nd loop)
i = 3
    j = 4
        Writes on the screen "(3,4)  "
    j = 5 (End of 2nd loop)
i = 4
    j = 5 (End of 2nd loop)
i = 5 (End of 1st loop)
```

This program includes an outer *for* loop (which starts on line 7 and ends on line 11) that contains within its instruction block an inner *for* loop (which starts on line 8 and ends on line 10). The control variable of the outer loop is the variable `i`, and the control variable of the inner loop is the variable `j`. The program's execution process is then as presented on the right-hand side.

Upon reaching line 7, the program declares and initializes the variable `i` with the value 1. Then, it checks whether the condition `i<n` is satisfied, which in this case is since $1 < 5$. Therefore, the program enters the instruction block of the outer *for* loop, which is the inner *for* loop, and executes it. In the inner loop, the program declares and initializes the variable `j` with the value `i+1`, which in this case is 2. Then, the condition `j<n` is evaluated, and it is true. As a result, the program executes the instruction block of the inner *for* loop, that is, it prints to the screen "(1,2) ". The next step is the increment of the variable `j`, that is, `++j`, so the variable has value 3. Since 3 is still less than `n`, the program executes the instruction block of the inner *for* loop again, printing "(1,3) ". After that, the variable `j` is incremented again, becoming 4, which is still less than n. This means the instruction block of the inner *for* is executed again, and the program prints "(1,4) ". The value of `j` is incremented again, becoming 5. Since 5 is no longer less than n, the inner loop ends and the program returns to the outer loop. Therefore, the variable `i` is incremented, now having the value 2. Since 2 is less than n, the program executes the second *for* again, initializing the variable `j` with the value `i+1`, which is 3. The process continues until the condition of the outer *for* is no longer satisfied. The final output of the program is: "(1,2) (1,3) (1,4) (2,3) (2,4) (3,4) ".

From this example, we can understand that in the case of nested loops, the inner loop will be executed each time an iteration of the outer loop is performed. It is also important to mention that the initialization of the control variable of the second loop depends on the control variable of the first loop. However, the reverse would not be possible since the scope of the variable `j` is between lines 8 and 10.

2.3.5 The *break* and *continue* Statements

Regardless of the structure used to implement a loop (*do-while*, *while*, or *for*), that loop will only terminate when its condition is no longer satisfied. However, in some situations, it may be useful to exit a loop before that happens. For this purpose, we can use the *break* statement inside the loop's instruction block. Upon encountering this statement, the program immediately exits the loop.

However, in nested loops, the *break* statement only exits the loop in which it is placed (depending on its location) and not all loops. Consider the following example:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int n = 5;
6.
7.     for(int i = 1; i < n; ++i) {
8.        for(int j = i + 1; j < n; ++j) {
9.            cout<<"("<<i<<","<<j<<") ";
10.               if ( j % i == 0 )
11.                    break;
12.       }
13.    }
14.
15.    return 0;
16. }
```

```
i = 1
    j = 2
        Writes on the screen "(1,2) "
        break (since 2%1==0)
i = 2
    j = 3
        Writes on the screen "(2,3) "
    j = 4
        Writes on the screen "(2,4) "
        break (since 4%2==0)
i = 3
    j = 4
        Writes on the screen "(3,4) "
    j = 5 (End of 2nd loop)
i = 4
    j = 5 (End of 2nd loop)
i = 5 (End of 1st loop)
```

This program differs from the previous one in that it contains the *break* statement, which only affects the inner loop (the loop of `j`). The use of this statement in the program means that the inner loop can terminate for two reasons: (i) when $j \geq n$; or (ii) when `j` is a multiple of `i`. The execution of the program is shown on the right-hand side.

The *continue* statement allows skipping an iteration of a loop (*while*, *do-while*, or *for*) without terminating it. More precisely, it causes the program to "jump" from the line where the *continue* statement appears to the end of the loop in which it is inserted. Consider the following example:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     for(int i = 1; i <= 9; ++i) {
6.        if(i % 4 == 0) {
7.             continue;
8.        }
9.       cout << i << endl;
10.    }
11.    return 0;
12. }
```

```
1
2
3
5
6
7
9
```

The program shown on the left prints all numbers from 1 to 9 that are not divisible by 4. Thus, the *for* loop iterates through all numbers from 1 to 9, and when a number is divisible by 4, it "jumps" to the end of the *for* loop, skipping the instruction that prints the numbers to the screen (line 9). Assume that the variable i has the value 3. In this case, the condition of the *if* is evaluated as false, and the program does not enter the *if* block. Therefore, the next instruction to be executed is on line 9, and the value 3 is printed on the screen. Next, the loop control variable i is incremented to 4, and the condition of the *if* is evaluated as true, executing the *continue* statement, which causes the program to "jump" from line 7 to line 10, ending the current iteration of the *for* loop without printing the value 4 to the screen. The output of the program is shown on the right.

One must be careful when using the *continue* statement in *while* and *do-while* loops since, in these loops, the value of the loop control variable is updated within the body of the loop. Because the use of the *continue* statement causes the program to "jump" over instructions, the loop control variable may not be updated, resulting in an infinite loop. Consider the following program:

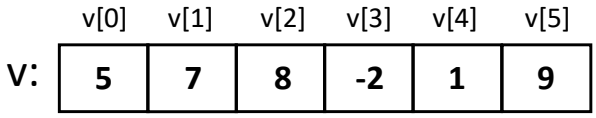
```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int i = 1;
6.     while(i <= 9) {
7.        if(i == 5) {
8.           continue;
9.        }
10.       cout << i << endl;
11.       ++i;
12.    }
13.
14.    return 0;
15. }
```

The previous program was supposed to print the integers from 1 to 9 on the screen; however, it only prints the numbers from 1 to 4 and then enters an infinite loop. When the loop control variable i takes the value 5, the condition of the *if* statement on line 7 is evaluated as true, and the *continue* statement on line 8 is executed, causing the program to "jump" to line 12. This means that the loop control variable i is never incremented again, thus resulting in an infinite loop.

Chapter 3

Indexed Variables – Vectors

A *vector* is a non-primitive data type that allows storing a sequence of variables of the same type, for example, several variables of type *int*. Each variable is an *element* of the vector and is identified by a non-negative integer called an *index*. The index of the first element is zero, so a vector with last index n will have $n + 1$ elements. The figure below shows an integer vector `v` with six elements (indices from 0 to 5).



The elements of a vector do not have names and are identified by their index. For example, the variable that contains the value 5, and occupies the first position of the vector `v`, is identified as `v[0]`, while the variable that contains the value -2 is identified as `v[3]`.

3.1 Vector Declaration

The use of vectors in C++ requires including the package *vector*[1]; thus, it is necessary to write the instruction: `#include<vector>` whenever we want to use vectors. There are two different ways to declare a vector, depending on whether we know its size in advance.

3.1.1 Vector Declaration with a Known Size

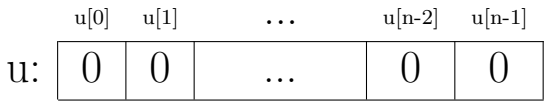
This form of declaration is used when we know exactly how many elements the vector will have. It can be done in one of the following ways:

```
vector<Data_type>  Vetor_name(n);
                     or
vector<Data_type>  Vetor_name(n, x);
```

Both statements create a vector named `Vector_name` with `n` positions (from 0 to `n-1`) to store elements of type *Data_Type*. The difference between them is that the first initializes all vector elements with the *default* value of the vector's *Data_Type*, while the second initializes all vector elements with

[1]See `https://cplusplus.com/reference/vector/vector/` for more information about the *vector* package.

the value `x`. Considering that the *Data_Type* is `int` and the *Vector_name* is `u`, i.e., `vector<int> u(n)`, the following figure represents what the first statement does.



As mentioned above, the vector `u` has n elements, but each element will have the *default* value of type `int`, which is 0. To construct the vector presented earlier, $v = (5, 7, 8, -2, 1, 9)$, it is necessary to assign specific values to each element of the vector, which is done in the following code example:

```
1.   #include <vector>
2.   using namespace std;
3.
4.   int main(){
5.       vector<int> v(6);   //Declaration
6.
7.       v[0] = 5;    //or v.at(0) = 5;
8.       v[1] = 7;    //or v.at(1) = 7;
9.       v[2] = 8;    //or v.at(2) = 8;
10.      v[3] = -2;   //or v.at(3) = -2;
11.      v[4] = 1;    //or v.at(4) = 1;
12.      v[5] = 9;    //or v.at(5) = 9;
13.
14.      return 0;
15.  }
```

3.1.2 Vector Declaration with an Unknown Size

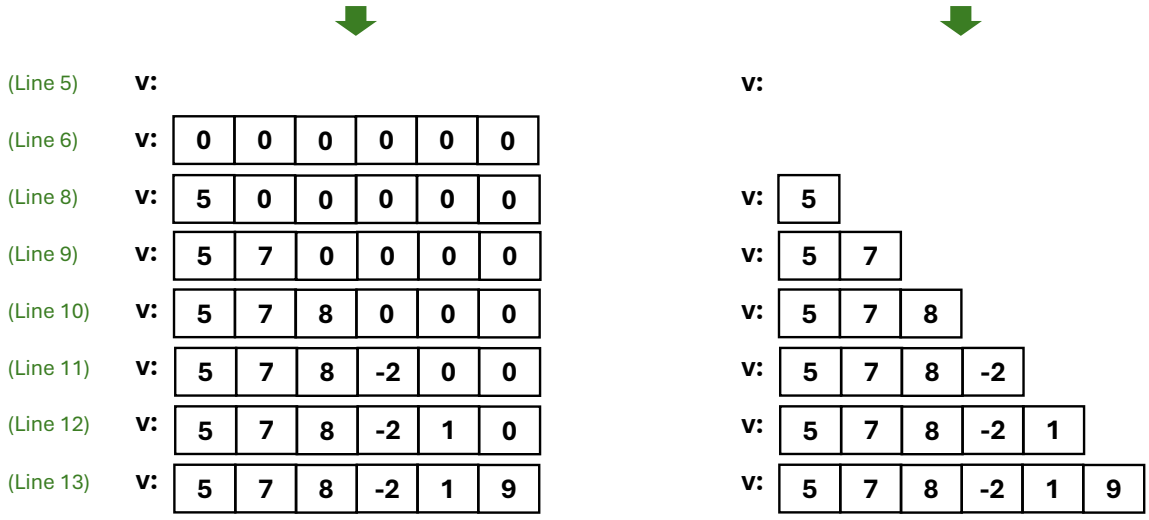
C++ also allows the creation of vectors without specifying their size at the time of creation. For this purpose, the following statement can be used:

```
vector<Data_Type>  Vector_name;
```

However, it should be noted that this statement alone is useless because it only declares the vector, that is, it creates a vector with no positions. Therefore, if we want to store elements in the vector, we must first create the necessary positions to store them. These positions can be created all at once by resizing the vector using the `resize` statement, or they can be created one by one using the `push_back` statement, as shown below.

```
1. #include <vector>
2. using namespace std;
3.
4. int main(){
5.     vector<int> v; //Declaration
6.     v.resize(6);   //Resizing
7.
8.     v[0] = 5;      //Filling
9.     v[1] = 7;
10.    v[2] = 8;
11.    v[3] = -2;
12.    v[4] = 1;
13.    v[5] = 9;
14.
15.    return 0;
17. }
```

```
1. #include <vector>
2. using namespace std;
3.
4. int main(){
5.     vector<int> v;
6.
7.    //Add new element/position to vector
8.     v.push_back(5);
9.     v.push_back(7);
10.    v.push_back(8);
11.    v.push_back(-2);
12.    v.push_back(1);
13.    v.push_back(9);
14.
15.    return 0;
17. }
```



In the program on the left, a vector without elements is initially created (line 5) and then immediately resized (line 6) to have six positions. During resizing, the *default* value of the vector's data type (zero in the case of numeric types) is automatically assigned to all elements of the vector. Once the positions are created, they are filled in lines 8–13. In the program on the right, the vector is also declared without elements. However, each time an element is to be added to the vector, a position for that element is first created and then filled with the designated element. All of this is done internally by the `push_back` instruction.

In terms of computational efficiency, the first code, where the `resize` instruction is used, is more efficient than the second, where the `push_back` instruction is used. Therefore, between these two instructions, we should choose the first one whenever possible.

The size of a vector can be changed multiple times during the execution of a program using the `resize` method. This method has the following behavior: (i) if the new size of the vector is greater than the previous one, all existing elements of the vector are preserved and the extra positions are created,

with the elements in these positions taking the *default* value of the vector's data type (0 in the case of numeric types); and (ii) if the new size of the vector is smaller than the current size, the vector is simply truncated, and the last positions are removed. It is also worth noting that the `resize` method can be called with two arguments, that is,

```
v.resize(n,x);
```

In this case, `n` will be the new size of the vector, and `x` will be the value assigned to all elements in the newly created positions. For example, if we have $v = (1, 2, 3)$, the instruction `v.resize(5,10)` changes the vector to $v = (1, 2, 3, 10, 10)$.

Similar to other data types, it is also possible to initialize a vector at the moment of its declaration. However, such initialization must be done using a list of elements. Considering $v = (5, 7, 8, -2, 1, 9)$, it would be enough to write:

```
1. #include <vector>
2. using namespace std;
3.
4.    int main(){
5.       vector<int> v = {5, 7, 8, -2, 1, 9};
6.       return 0;
7.    }
```

This would be the simplest way to create the desired vector; however, this type of initialization is not always possible because the values to be placed in the vector (as well as the vector's size) are often not known at the time the vector is declared.

3.2 Method `.at()` *vs* Operator [ ]

To access or fill a position in a vector, both the [ ] operator and the `.at()` method can be used. The main difference between them is that the `.at()` method validates the vector position being accessed, that is, it checks if that position exists in the vector. Consider the following example:

```
1. #include <vector>
2. using namespace std;
3.
4.  int main(){
5.      vector<int> v(2);
6.      v[0] = 5;  //or v.at(0) = 5;
7.      v[1] = 7;  //or v.at(1) = 7;
8.      v[2] = 8;      //ERROR  (The program may not be interrupted)
9.      v.at(2) = 8;   //ERROR  (The program is immediately interrupted)
10.     //...
11.     return 0;
12. }
```

The vector `v` declared on line 5 has only two positions (position 0 and position 1). Therefore, the access to these positions can be done using either the [ ] operator or the `.at()` method (lines 6 and 7). When we try to access a position in the vector that does not exist (position 2, for example) using the [ ] operator, the program does not inform us that such a position is invalid and instead accesses some arbitrary memory location, returning whatever garbage value is stored there. Depending on the context, the program may terminate immediately without displaying any error message, or it may continue executing with the error "hidden". In the latter case, the error can propagate through the program without being noticed. By using the `.at()` method to attempt access to a non-existent position, the program will terminate immediately and display an `out_of_range` error message.

But then, why not always use the `.at()` method since it is safer? The main reason is efficiency. Because the `.at()` method always validates the position being accessed; its computational effort is greater, which can have a significant impact on the program's performance. In addition, the [ ] operator is simpler to write. Hence, each option has its advantages and disadvantages, and both approaches can be used in this course.

3.3 Vector Manipulation

A vector `v` can be seen as a sequence of indexed variables `v[i]`, where `i` represents the position of the variable within the vector. This means that each of these variables can be manipulated using the operators or methods defined for its type, such as `cin`, `cout`, `+`, and `==` among others. However, it is important to keep in mind that these operators are not defined for vectors as a whole. That is, if `v` and `u` are two objects of type `vector`, it is not possible to perform operations such as `v+u`. This means that, for now, a vector must always be manipulated element by element, and not as a whole, as explained below.

3.3.1 Filling Vectors

We have already seen how to fill a vector with values known in advance. Suppose now that we want to create an integer vector `v` of size `3`, with values provided by the user. We already know that the `cin` instruction allows us to request values from the user, but this instruction is not defined for vectors, so it is not possible to do something like "`cin >> v`". However, it is possible to write "`cin >> v[0]`", "`cin >> v[1]`", and "`cin >> v[2]`" because each element `v[·]` is a variable of type `int`. If we do that, we will repeat the procedure of requesting values from the user several times; thus, we can use a loop control structure to fill the vector. In addition, since we know exactly how many values will be requested, we should use the *for* loop.

The program below shows the creation and filling of vector `v` with user-provided values, with and without using a `for` loop.

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. int main(){
6.    vector<int> v(3);
7.
8.    //cin >> v; //ERROR
9.    cin >> v[0]; //or cin>>v.at(0);
10.   cin >> v[1];
11.   cin >> v[2];
12.
13.   return 0;
14. }
```

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. int main(){
6.    vector<int> v(3);
7.
8.    for(int i = 0; i < v.size(); ++i){
9.       cin >> v[i]; //or cin>>v.at(i);
10.   }
11.
12.
13.   return 0;
14. }
```

The code on the right fills the vector automatically, because the program successively goes to each position (from 0 to 2) and assigns it a value read from the user. The *size()* method returns the size of the vector (in this case, 3). Vector manipulation often requires the use of a *for* loop - such as the one shown on line 8 of the code on the right - to go through all the positions of the vector (from 0 to *size()-1*).

When the size of the vector is not known in advance, it is recommended to use a *while* loop to fill it. Suppose that we want to repeatedly ask the user for numerical values until a non-numeric value is entered. In this case, we do not know beforehand how many numerical values the user will provide, so we should define a vector without specifying its size and use the `push_back` instruction to create a new position in the vector each time a numerical value is inserted. This procedure is exemplified in the code below.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.
5.  int main(){
6.      vector<double> v;
7.
8.      double x;
9.      while(cin >> x){ //while numeric values are read
10.        v.push_back(x);
11.     }
12.
13.     return 0;
14. }
```

Each time the *while* loop iterates, the instruction `cin >> x` attempts to assign to the variable `x` (which is of type *double*) the value entered by the user. If that value is numeric, the assignment succeeds, and therefore the program enters the *while* loop, creates a new position in the vector, and stores the value of the variable `x` there. If the user enters a non-numeric value, that value cannot be

assigned to the variable `x`, and the instruction `cin >> x` returns `false`, making the *while* loop terminate immediately. Additionally, `cin` enters the *with error* state and cannot be used until its state is cleared (see Section 10.3 for more information).

3.3.2 Printing Vectors

A vector must also be printed element by element, because the *cout* method is not defined for vectors. Hence, since the vector we want to print has a known size (because it has already been created), the most common way to print it is by using a *for* loop, as shown in the code below.

```
1.   #include <iostream>
2.   #include <vector>
3.   using namespace std;
4.
5.  int main(){
6.      vector<int> v = {5, 7, 8, -2, 1, 9};
7.
8.      cout << "(";
9.      for(int i = 0; i < v.size(); ++i){
10.         if( i < v.size() - 1)
11.           cout << v[i] << ", ";  //For all elements excepting the last
12.         else
13.           cout << v[i] << ")";   //For the last element
14.     }
15.     return 0;
16.  }
```

It should be noted that the *for* loop has exactly the same structure as the one used to fill the vector, since it is necessary to iterate through all the positions of the vector. The *if* statement inside the *for* loop aims to distinguish the printing of the last element from that of the others. This is because, after printing the last element of the vector, a parenthesis should be written instead of a comma, as is done for the other elements. The program output will then be:

(5, 7, 8, -2, 1, 9).

3.3.3 Vector Sorting

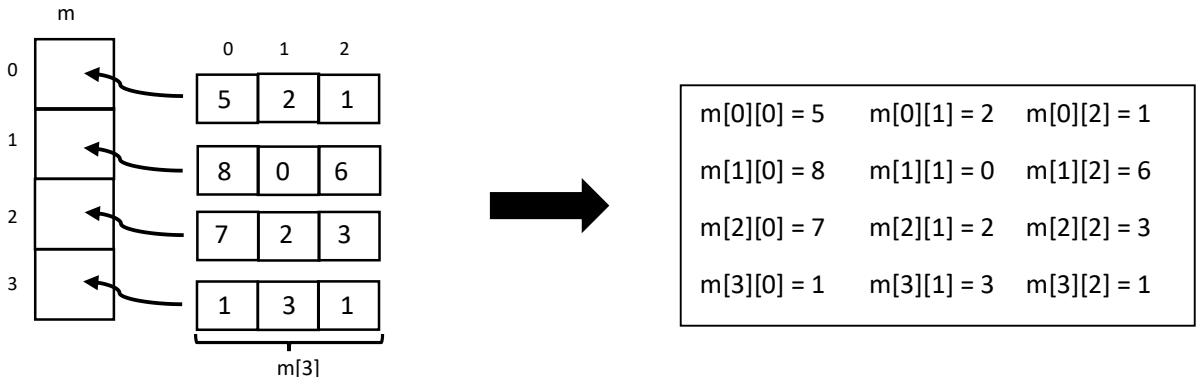
Sorting vectors is essential to simplify tasks that we commonly perform with vectors, such as searching for elements or obtaining descriptive statistics. There are several sorting algorithms, such as *Bubble Sort*, *Insertion Sort*, and *Sequential Sort*. C++ already provides a sorting method, *sort*, which we will use in this course whenever we need to sort vectors. This method belongs to the *algorithm* package, so its use requires including the instruction `#include<algorithm>` in the preamble. The code below demonstrates the sorting process of a vector in ascending and descending order.

```
1.  #include <vector>
2.  #include <algorithm>
3.  using namespace std;
4.
5.  int main(){
6.      vector<int> v = {5, 7, 8, -2, 1, 9};
8.
9.      //Sorting v by ascending order
10.        sort( v.begin(), v.end() );
11.
12.     //Sorting v by descending order
13.        sort( v.begin(), v.end(), greater <>() );
14.
15.     return 0;
16.  }
```

In line 6, for example, we have `v[0] = 5`. In line 10, the vector `v` is sorted in ascending order, so `v[0] = -2`. In line 13, the vector is sorted in descending order, so `v[0] = 9`.

3.4 Vectors of Vectors - Matrices

As we saw at the beginning of this chapter, an object of type `vector` stores variables of a given type. In particular, it can also store variables of type `vector`, which results in a *vector of vectors*. This data structure is the most natural way to represent a matrix in C++. Schematically, a matrix can be represented as follows:



The vector `m` is composed of four elements, and each of them is a new vector of size three. The vector `m` can then be seen as a 4×3 matrix, that is, a matrix with four rows and three columns. Each element of the matrix is identified by `m[i][j]`, where i is the index of the main vector `m` (row) and j is the index of the secondary vector `m[i]` (column). Since a matrix is a vector of vectors, it is necessary to define the size of all the vectors involved before filling the matrix. The creation of the matrix in the example above can be done as follows:

```cpp
#include <vector>
using namespace std;
int main(){
    //Option 1: Declare and fill the matrix
    vector<vector<int>> m = { {5, 2, 1}, {8, 0, 6}, {7, 2, 3}, {1, 3, 1} };

    //Option 2: Create the matrix without dimension and then resize it
    vector<vector<int>> m; //or vector<vector<int>> m(4); and remove line code 9
    m.resize(4); //Define the number of lines (dimension of the main vector)

    //Define the number of columns (dimension of each inner vector)
    for(int i = 0; i < m.size(); ++i)
          m[i].resize(3);

    //Fill the matrix
    m[0][0] = 5;
    //...
    m[3][2] = 1;

    return 0;
}
```

The first option to create the matrix is clearly the simplest one. However, this option is only possible when both the dimensions and the elements of the matrix are known at the time of its creation, which is not often the case. When the dimensions of the matrix and its elements are not known beforehand — for example, if this information is requested from the user during the program execution — the second option must be used. It is important to emphasize that we cannot write `vector<vector<int>> m(4,3)` to create a 4×3 matrix. The type `vector<vector<int>>` expects its second constructor argument to be a `vector<int>`, not an `int`. To create a 4×3 matrix (4 rows and 3 columns) initialized with zeros, we can write

```cpp
vector<vector<int>> m(4, vector<int>(3, 0));
```

This constructs an outer vector with four entries, where each entry is an inner `vector<int>` of length 3, with every element equal to 0.

In the example presented above, the *vector of vectors* was used to represent a matrix; thus, all the inner vectors must have the same size. However, it would also be possible to define a vector of vectors where the inner vectors have different sizes.

As in the case of simple vectors, matrices are also manipulated element by element, and not as a whole. This means that to manipulate a matrix, it is necessary to iterate through all its elements, that is, through all its rows and columns. The simplest way to do this is by using two nested *for* loops, where the first loop iterates through the "rows" of the matrix and the second through the "columns". The code below shows how to print a matrix in C++.

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

int main(){
    vector<vector<int>> m = { {5, 2, 1}, {8, 0, 6}, {7, 2, 3}, {1, 3, 1} };

    for(int i = 0; i < m.size(); ++i){           //Row i
        for(int j = 0; j < m[i].size(); ++j){    //Column j
           cout << m[i][j] << " ";
        }
        cout << endl;   //Write a line break before going to the next line i
     }

    return 0;
}
```

Note that we use the `cout` method applied to each element of the matrix, and not to the matrix as a whole. Doing something like "`cout << m`" is not possible because the `cout` command is not defined for objects of type `vector<vector<int>>`. The process of filling a matrix is similar to the writing process, that is, it requires the use of two *for* loops; thus, it will not be shown here.

Chapter 4

Functions

The term *function* intuitively brings us to the field of mathematics, where a function is characterized by a domain, a codomain, and an analytical expression. For example, function f

$$\begin{aligned} f: \ \mathbb{Z} \times \mathbb{Z} &\longrightarrow \mathbb{R} \\ (x, y) &\longrightarrow f(x, y) = \frac{x}{y^2 + 1} \end{aligned}$$

takes two integer arguments (x and y) and returns a real value, which is the result of $\frac{x}{y^2+1}$.

In programming, it is important to distinguish between three concepts associated with a function: *declaration*, *definition*, and *call*. The *declaration* of a function consists of specifying its name, the type of its arguments (the data types it receives — domain), and the return type (the data type it returns — codomain). The *definition* of a function consists of specifying what the function does (analytical expression). Finally, the *call* of the function consists of executing the function for specific values of its arguments. In this example, we have:

$$\begin{aligned} f: \ \mathbb{Z} \times \mathbb{Z} &\longrightarrow \mathbb{R} && \text{(declaration)} \\ (x, y) &\longrightarrow f(x, y) = \frac{x}{y^2 + 1} && \text{(initialization)} \\ f(2, 3),\ f(5^2, -8)\ ... & && \text{(call)} \end{aligned}$$

In C++, for this same example, we would have:

```
//declaration
     double f(int, int);

//declaration and definition
     double f(int x, int y){
          return x / (y * y + 1);
     }

//call
     cout << f(2,3);            //call 1
     double z1 = 5 * f(1,7);    //call 2
     int a = 6, b = 1;
     double z2 = f(a,b);        //call 3
```

In the first code block, only the function declaration is made. That is, it indicates that the function f takes two arguments of type *int* and returns a result of type *double*. In the second code block, both the declaration and the definition of the function are made. The definition of the function corresponds to the block of code that appears inside the brackets {...}. Finally, in the last code block, three *calls* to the function are made. Note that in these cases, just like in mathematics, we only need to provide the values as arguments to the function without specifying their types, since that was already made explicit in the function's declaration/definition.

In this case, the function f returns a real value that can either be printed directly to the screen (call 1) or used to assign a value to a variable (calls 2 and 3). In the first function call, the first argument `x` will take the value 2 and the second argument `y` the value 3. In the third function call, `x` will take the value of variable `a` (which is 6), `y` the value of variable `b` (which is 1), and the value returned by the function will be stored in the variable `z2`.

It is important to note that the names of a function's arguments are used only internally within the function and are therefore independent of the variable names used in the function call. Moreover, since this function was declared with two arguments of type *int*, it must always be called with two arguments of type *int*. Thus, for example, $f(1)$, $f(1, 8, 5)$, $f(\text{`}j\text{'}, 5)$, and f are not valid calls for this function.

4.1 General Syntax of a Function

The general structure for declaring and defining a function in C++ is as follows:

```
Return_type function_name(Type_a1 name_a1, ..., Type_an name_an){
   //...
   return ... ; //If "Return_type" different from "void"
}
```

in this declaration and definition:

- `function_name` is the name given to the function;
- `Return_type` is the data type returned by the function. If the function does not return any result, the return type will be *void*;
- `name_v1`, ..., `name_an` are the names of the function's arguments;
- `Type_a1`, ..., `Type_an` are the data types of the function's arguments.

When the *return* statement is executed, the program immediately exits the function, and the return value is passed back (via a copy) to the part of the program where the function was called. Therefore, this statement does not need to be used in functions of type *void*, since these do not return any value. *Void* functions are often used to print something to the screen, thus returning no result to be used by the program that called them, unlike the function f presented earlier.

Below is an example showing the declaration and definition of two functions: one with return type *int*, and the other with return type *void*. However, it is important to point out that functions do not necessarily need to have arguments, as is the case with the *main* function that we have always used.

Example 1

Suppose we want to implement a function that returns the maximum of two integers and another that prints two integers in ascending order. Both functions receive the same arguments: two integers

that will be referred to as `n1` and `n2` within the function. Since the first function (lines 4–9) aims to calculate the maximum of two integers, its return type is also an integer. However, as the second function (lines 11–16) only prints the two received integers in ascending order, it does not return any result when called. As such, this function is of type *void*, and for that reason, there is no *return* statement inside it.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int maximum(int n1, int n2){
5.      int max = n1;
6.      if( max < n2 )
7.          max = n2;
8.      return max;
9.   }
10.
11.  void order(int n1, int n2){
12.     if( n1 < n2 )
13.        cout << n1 << " <= " << n2;
14.     else
15.        cout << n2 << " <= " << n1;
16.  }
17.
18.  int main(){
19.     int a;
20.     int b;
21.     cout << "Insert a and b: ";
22.     cin >> a >> b;
23.
24.     //Function calls
25.     int x = maximum(5,7);
26.     int y = maximum(a,b) - 6;
27.     cout << "The maximum between 2 and 8 is " <<  maximum(2,8) << endl;
28.     int z = maximum( maximum(7,8) , maximum(1,6) );
29
30.     cout << "Order: ";
31.     order(7,5);
32.     cout << "\n Order: ";
33.     order( maximum(1,8) , 3 );
34.     return 0;
35.  }
```

Since a C++ program always begins execution with the *main* function, all other functions must be declared before it. That is why the functions `maximum` and `order` are declared/defined in lines 4–16.

As previously mentioned, when *calling* a function, it is mandatory to write its name and all of its arguments (without specifying their types). When calling the `maximum` function in line 25, we indicate that the values of its arguments `n1` and `n2` (line 4) are 5 and 7, respectively. Therefore, when the program reaches line 25, it jumps to line 4 and executes the instructions of the `maximum` function considering

`n1=5` and `n2=7`. Upon reaching line 8, the program returns the value of the variable `max` (which will be 7) to the place where the function was called, i.e., to line 25 in the `main` function, and thus the value of the variable `x` will be 7.

When calling the `maximum` function in line 26, the values of the variables `a` and `b` (previously requested from the user) are passed as arguments to the function, defining the values of `n1` and `n2`. The function `maximum` ends its execution by returning the maximum between `a` and `b` in line 26. From this maximum, the value 6 is subtracted, and the final result is stored in variable `y`.

The value returned by a function does not necessarily have to be stored in a variable as in lines 25 and 26. Since the `maximum` function returns an `int`, it can be printed directly to the screen, as done in line 27.

A function can also be called with arguments that are themselves functions, provided that the return types of the "inner" functions match the argument types expected by the "outer" function, as in line 28. The calls to `maximum(7,8)` and `maximum(1,6)` yield the values 8 and 6, respectively, both of type *int* since that is the return type of `maximum`. These values become the arguments of the "outer" function, so line 28 is equivalent to `int z = maximum(8,6);`. Therefore, the value of the variable `z` will be 8.

Unlike the *maximum* function, the *order* function does not return anything when called in the the `main` function; it only prints information to the screen and is thus a *void* function. As such, this function must be called as a standalone statement (lines 31 and 33), and it cannot be used to define variable values or be placed inside a *cout*.

The *order* function takes two *int*-type arguments, so it can be called as in line 33, since the function call `maximum(1,8)` returns an *int* value that will be used as the first argument of the *order* function.

In the code below, several examples of incorrect calls to functions `maximum` and `order` are presented.

```
int maximum(int n1, int n2){...}
void order(int n1, int n2){...}

//Incorrect function calls
    int w = maximum;                        //Arguments missing
    int a = 9, b = 4;
    int r = maximum(int a , int b);         //Argument types inserted
    int s = maximum(c, d);                  //Variables c and d not declared
    int t = maximum("A", 3);                //First argument is not of type int
    int y = maximum(n1, n2);                //Undefined arguments
    int u = maximum(5);                     //Arguments missing
    cout << "The maximum is " <<  maximum;  //Arguments missing
    int h = order(2,8);                  //Function order does not return an int
    cout << "Order: " <<  order(7,5);    //Function order return nothing
    order;                               //Arguments missing
   cout << maximum(order(1,6), 5);       //Function order does not return an int
```

Example 2

A function can have arguments of any data type, in particular, it can take arguments of type `vector`. In the program below (on the left), a *print* function is declared and defined, which takes a vector of integers as an argument and returns that vector written as a *string*.

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. string print(vector<int> x){
6.    string s = "(";
7.    for(int i = 0; i<x.size(); ++i){
8.       if( i < x.size() - 1)
9.          s+=to_string(x[i]) + ", ";
10.       else
11.          s+=to_string(x[i]) + ")";
12.   }
13.   return s;
14. }
15.
16. int main(){
17.    vector<int> v = {2, 3, 1, 7};
18.    vector<int> u = {3, 5, 1, 1};
19.    vector<int> w(4);
20.
21.    for(int i = 0; i<v.size(); ++i)
22.       w[i] = v[i] + u[i];
23.
24.    cout << print(v) << "+";
25.    cout << print(u);
26.    cout << "=" << print(w);
27.
28.    return 0;
29. }
```

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. int main(){
6.    vector<int> v = {2, 3, 1, 7};
7.    vector<int> u = {3, 5, 1, 1};
8.    vector<int> w(4);
9.
10.   for(int i = 0; i<v.size(); ++i)
11.      w[i] = v[i] + u[i];
12.
13.   string s1 = "(";
14.   for(int i = 0; i<v.size(); ++i){
15.      if( i < v.size() - 1)
16.         s1+=to_string(v[i]) + ", ";
17.      else
18.         s1+=to_string(v[i]) + ")";
19.   }
20.
21.   string s2 = "(";
22.   for(int i = 0; i<u.size(); ++i){
23.      if( i < u.size() - 1)
24.         s2+=to_string(u[i]) + ", ";
25.      else
26.         s2+=to_string(u[i]) + ")";
27.   }
28.
29.   string s3 = "(";
30.   for(int i = 0; i<w.size(); ++i){
31.      if( i < w.size() - 1)
32.         s3+=to_string(w[i]) + ", ";
33.      else
34.         s3+=to_string(w[i]) + ")";
35.   }
36.
37.   cout<< s1 << "+" << s2 << "=" << s3;
38.   return 0;
39. }
```

In the *print* function, the received vector is always referred to as `x`. This means that each time the *print* function is called with a given vector (`u`, `v`, or `w`, as in lines 24, 25, and 26), a copy of that vector (named `x`) will be created and used as the function's argument.

This example aims to demonstrate two major benefits of using functions: avoiding code repetition and simplifying the program where the function is called (the *main* function). Both code excerpts above

produce the same output, namely:

$$(2, 3, 1, 7) + (3, 5, 1, 1) = (5, 8, 2, 8).$$

However, the code on the right does not use functions, and therefore, each time a vector needs to be printed, lines 13–19 must be replicated for the respective vector. Note that the process carried out in lines 21–27 and 29–35 is the "same" as in lines 13–19, only applied to different vectors. In addition to resulting in longer code, repeating code snippets also increases the likelihood of errors. Defining a function to print a vector `x`, as done in the code on the left, allows that whenever an integer vector needs to be printed (regardless of its size), it is only necessary to call the function for that vector, as shown in lines 24, 25, and 26 of the code on the right. Note that the *main* function on the left is much easier to read than the one on the right.

4.2 Advantages of Functions

Functions are extremely useful in programming. As illustrated in the previous example, functions help avoid code repetition since they are implemented in a very generic way and can then be called multiple times with different arguments.

Another major advantage of functions is that they allow for code modularity. That is, with functions, it is possible to break the code into smaller pieces that are easier to organize, test, and use. This way, functions facilitate task division in programs involving multiple people, as they are completely independent structures.

Once defined, functions can be used by many people in various programs. From the user's point of view, it is only necessary to know how a function was declared — not how it was defined. In other words, one only needs to know the function's name, arguments, and return type. Note that, perhaps without realizing it, we have already used several functions without knowing how they were defined. Some examples of such functions include *size*, *resize*, and *at* for *vector* objects, as well as the function that converts numeric values to *string* (*to_string*).

4.3 Pass-by-Value, Pass-by-Reference, and Pass-by-Constant-Reference

The arguments of a function can be passed *by value*, *by reference*, or *by constant reference*, and the way to do so is as follows:

```
Type name(ArgumentType argument){...}        // Pass-by-value

Type name(ArgumentType& argument){...}        // Pass-by-reference

Type name(const ArgumentType& argument){...} // Pass-by-constant-reference
```

In *pass-by-value*, a copy of the argument used in the function call is passed to the function. This means that all changes made inside the function are applied to that copy and not to the original variable passed as an argument.

In *pass-by-reference*, what is passed to the function is not a copy of the variable's value, but rather the memory address where the variable is stored. Therefore, the function can "see" and "modify" the value of the original variable. This means that any change made inside the function will affect the original variable passed in the function call.

Pass-by-constant-reference is similar to *pass-by-reference* in that is the memory address of the variable that is passed as an argument. However, by using a constant reference, the function can only "see" the variable and cannot make any changes to it.

Consider the following example:

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void f(int a, int& b, const int& c){
5.      a += 10 + c;
6.      b += 10 + c;
7.      //c += 10;    ERROR!
8.      cout << a << " " << b << " " << c;  //a=12, b=12, c=1
9.  }
10.
11. int main(){
12.     int x = 1;
13.     int y = 1;
14.     int z = 1;
15.     f(x, y, z);
16.     cout << x << " " << y << " " << z;  //x=1, y=12, z=1
17.
18.     return 0;
19. }
```

The function f is called on line 15 with the arguments `x`, `y`, and `z`. This function receives three arguments: the first is passed *by value*, the second *by reference*, and the third *by constant reference*. Therefore, in the function call on line 15, the value of the variable `x` (which is 1) is passed as an argument, along with the memory address of variable `y` and the memory address of variable `z`. This means that `a=1`, `b` is exactly the variable `y`, and `c` is exactly the variable `z`. Therefore:

- changing the variable `a` inside the function does not change the variable `x`, since `a` is a copy of `x` and not the variable `x` itself. Note that at the end of the function execution, we have `a=12` and `x=1`.
- changing the variable `b` is the same as changing the variable `y`, and therefore, at the end of the function execution, we have `b=y=12`.
- `c` is a constant reference to the variable `z`, so the value of `c` cannot be changed by the function (line 7), and therefore we have `c=z=1`.

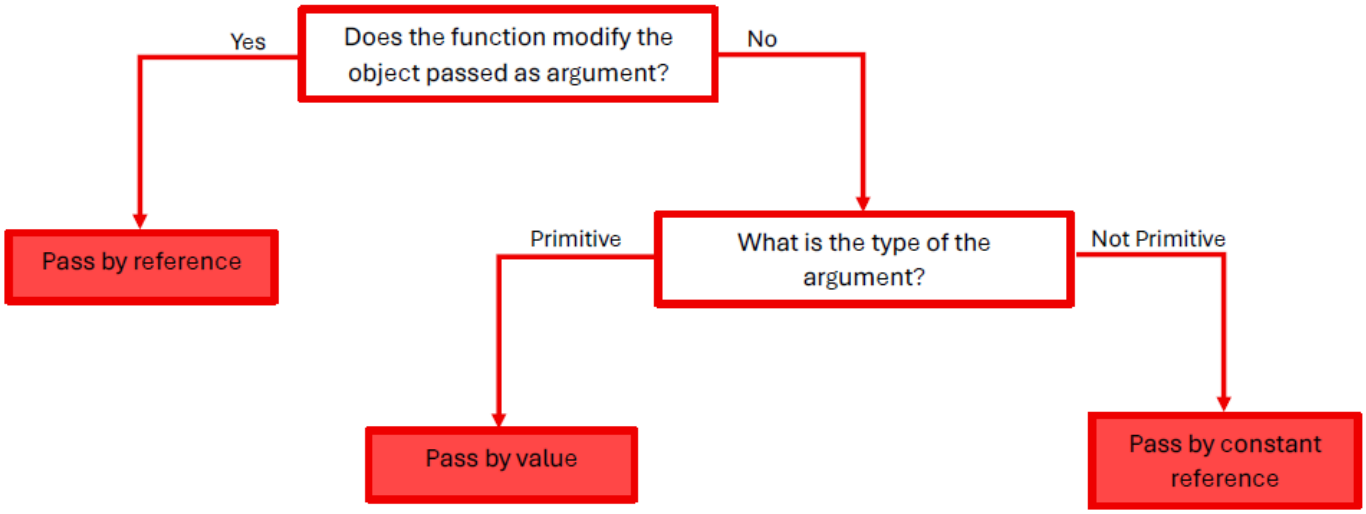
Which Passing Type Should Be Used?

The appropriate type of passing an argument to a function depends primarily on whether the function modifies that argument permanently. Permanently modifying an argument means that if it is changed

inside the function, it will remain changed outside the function as well. For this to happen, the function must receive the argument by *reference*, regardless of the argument's data type.

When a function does not permanently modify its argument, the passing type depends on the data type of that argument. Primitive data types such as *int*, *double*, *char*, and *bool* are considered "small" objects, whereas non-primitive types such as *vectors*, *strings*, and *classes* (which we will see later) are considered "large" objects. The time and computational effort required to create copies of primitive type objects is negligible, so these types of objects are generally passed *by value* to functions. The same does not apply to non-primitive objects, where creating copies can be time-consuming due to their potentially large size. Therefore, non-primitive objects that are not modified by the function should be passed *by constant reference*.

The diagram below illustrates how the appropriate argument passing method should be chosen.



Suppose we want to implement a function that receives a vector and prints it. This function only needs to access the elements of the vector (without modifying them), and since vectors are non-primitive data types, the function should receive a constant reference to a vector. Using constant references also provides an added layer of safety for the programmer, as it guarantees that a given object will not be modified within the function.

As we already know, a function can return only one object. However, non-constant references can be used as an "artificial" way to return more than one objects. Note that in the example from the previous section, even though the function *f* is of type *void* and does not return any value, the new value of the variable `b` (which is 12) is "returned" to the *main* function because a non-constant reference was used.

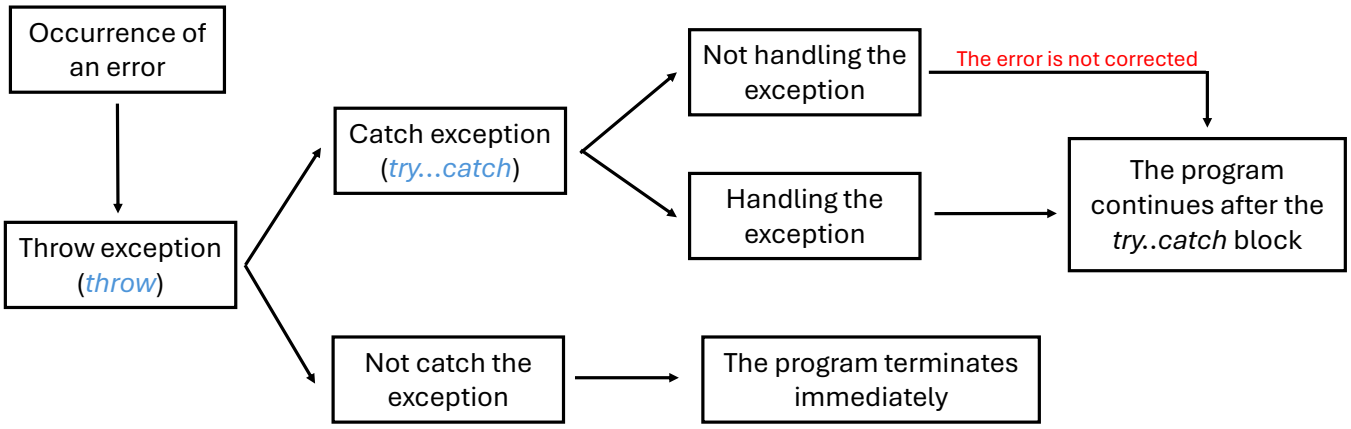
Chapter 5

Error handling

To be robust, a program should be able to handle the errors that may occur during execution in a safe and predictable way. There are several ways to handle errors, and the most appropriate choice depends on the type of error encountered. Errors can disrupt normal execution, so they must be detected and handled. A program should include mechanisms to recover when possible—such as correcting the input, retrying an operation, using a default value, or reporting the problem and continuing. If an error is not handled, the program may terminate immediately or continue in an invalid state, leading to additional failures that compromise its execution.

One of the most common mechanisms for handling errors is the use of *exceptions*. Exceptions are abnormal situations that occur during the execution of a program. Therefore, a program must be prepared to signal all exceptions that may occur. Signaling an exception means identifying a problem that might arise during program execution and informing the system of its existence — this is called *throwing* an exception. To throw an exception, we use the *throw* statement. Throwing an exception causes the program to terminate immediately, unless a mechanism is used to *catch* (and possibly handle) the thrown exception.

To catch an exception thrown to the system, we use a *try...catch* block. The *try* statement looks for exceptions that might be thrown by the code block associated with it. The *catch* statement defines the actions to handle the thrown exception and thus allows the program to continue its execution. These actions may be merely informative — that is, they may simply inform the user of the occurrence of the error without correcting it — or they may actually correct the existing error. The flowchart below summarizes what happens to the program depending on the mechanisms used to catch or handle an exception.



The general syntax of the *throw* and *try...catch* structures is as follows:

```
//Instruction_block_1

if(error_condition)
     throw  Exception_to_throw;

//Instruction_block_2

//Instruction_block_3
```

```
try{

    //Instruction_block_1

    if(error_condition)
        throw  Exception_to_throw;

    //Instruction_block_2

}catch( Exception_to_catch ){
     //Handling the exception
}

//Instruction_block_3
```

The throwing of exceptions is generally performed within conditional instructions because it only occurs if a specific error condition is met. In the code on the left, the program starts by executing `Instruction_block_1`. If the `error_condition` is verified, an exception is thrown and the program terminates immediately. Otherwise, the program will execute the two subsequent instruction blocks.

For an exception to be caught by a *catch* block, it must be thrown within the *try* block associated with that *catch*. Inside the *catch* block, the programmer specifies what should be done if the `Exception_to_catch` has been thrown within the corresponding *try* block. In the code on the right, the program enters the *try* block directly and begins by executing `Instruction_block_1`. Next, it checks whether the `error_condition` is true, and if so, an exception is thrown. After throwing the exception, the program skips `Instruction_block_2` and jumps immediately to the *catch* block. Upon reaching the *catch* block, the program checks whether the thrown exception is of the type `Exception_to_catch`. If not, the program terminates immediately. Otherwise, the instructions inside the block *catch* are executed to handle the exception, and the program continues its execution by proceeding to `Instruction_block_3`.

If, after entering the *try* block and executing `Instruction_block_1` in the code on the right, the `error_condition` is not verified — meaning that no error has occurred — the program will execute `Instruction_block_2` and then `Instruction_block_3`, ignoring the block*catch*.

A *try* statement can be associated with multiple *catch* blocks, one for each exception that needs to be handled. If an exception is thrown inside the *try* block and the corresponding *catch* block is not prepared to handle it, the exception will not be caught, and the program terminates immediately. To prevent this, the last *catch* block associated with the *try* statement should be a general block capable of catching all exceptions that were not handled by the previous *catch* blocks. To catch any type of exception, ellipses (`...`) must be used as the argument of the *catch* statement. The example below presents a *try* block with three associated *catch* blocks — the first and second handle two specific exceptions (`Exception_1` and `Exception_2`), while the last is a general block that catches any other exception not covered by the previous ones. It should be noted that when the *throw* statement is executed inside the *try* block, the program immediately jumps to the corresponding *catch* block. Since only one exception is thrown, only one of the *catch* blocks (`Exception_1`, `Exception_2`, or `...`) will be executed.

```
try{

    //...

}catch( Exception_1 ){
      //Handling_Exception_1
}catch( Exception_2 ){
      //Handling_Exception_2
}catch( ... ){
      //Handling_the_remaining_Exception
}
```

Exceptions are objects of a certain type (`int`, `string`, among others). However, to clearly identify the exceptions that are thrown and to handle them differently, we will define a separate *class* for each exception. Classes will only be introduced in detail in Chapter 7. For now, it is enough to understand that a class is a user-defined, non-primitive data type. Next, we will explain how we can use classes defined by the programmer to throw an exception and classes already available in the *standard* library for the same purpose.

5.1 Empty Classes

Using empty classes as exception types is especially useful when a program can raise different kinds of errors and you want to handle each one separately. With this approach, you first declare an "empty class" for each type of exception that may occur. These declarations must appear before any function that may throw those exceptions. In the following example, the class declarations are placed before the *main* function, which is the function that throws the exceptions. In this example, two empty classes are defined, each representing a different error condition. The try block has two corresponding catch blocks, each handling a different exception type. Also note the use of constant references in the arguments of the *catch* instruction, which is justified by the fact that we are dealing with objects of non-primitive data types (classes). T

```
//Preamble
class Name_for_exception_1{};
class Name_for_exception_2{};

int main{

    //...

    try{
        if(error_condition_1)
            throw  Name_for_exception_1();

        //...

        if(error_condition_2)
            throw  Name_for_exception_2();

        //...

    }catch( const Name_for_exception_1& ){
            //Handle exception_1
    }catch( const Name_for_exception_2& ){
            //Handle exception_2
    }catch( ... ){
            //Handle other exceptions
    }

    //...
    return 0;
}
```

In the example below, the user is required to introduce the values of the variables n and m, and a new variable *result* is created to store the value of the integer division of n by m. If any of the values of n or m are not read correctly — for example, if a non-numeric value is entered — an exception of type `Wrong_Read` is thrown, and the division between n and m is not performed because the program immediately jumps to the first *catch* block. In that block, the exception is handled by assigning the value 1 to the variable *result*, and the program continues executing, moving on to the final *cout* statement where the value 2 is printed on the screen. Note that the verification of the success of the input read process for a given variable is done through the statement `if(!cin)`, which means "if the correct data type was not read in the previous *cin*". The `cin` instruction evaluates to `false` when the input operation fails.

If the reading of the values of n and m is successful but the value of m is zero, an exception of type `Null_Value` is thrown, and the program does not perform the division between n and m. In this case, the program immediately jumps to the last *catch* block, where only an error message is displayed. This is therefore an example in which we handle the exception (by displaying an error message) but do not actually solve it — that is, the program continues executing and the error is ignored. In this situation,

the variable *result* will keep its initial value (zero), and the final *cout* statement will print the value 1 on the screen.

It should be noted that when we handled the `Wrong_Read` exception, we assigned the value 1 to the variable `result`, which is an arbitrary choice. The question that arises is: what value should we assign to the integer division when one of the operands is a non-numeric value? The answer to this question is unclear, so this is the main reason why we usually do not handle exceptions by assigning values, but rather only deal with them by displaying an error message.

```
//Preamble
class Null_Value{};
class Wrong_Read{};

int main(){
  int n, m, result = 0;

  try{
      cout << "Value for n: ";
      cin >> n;
      if(!cin)
          throw  Wrong_Read();

      cout << "Value for m: ";
      cin >> m;
      if(!cin)
          throw  Wrong_Read();

      if(m == 0)
          throw  Null_Value();

      result = n/m;

  }catch( const Wrong_Read& ){
            result = 1;
  }catch( const Null_Value& ){
            cout << "Attention! the value of m is zero...";
            cout << "... but the program carries on executing!";
  }

  cout << result + 1;
  return 0;
}
```

5.2 Classes from the *Standard* Library

In C++ there are several predefined classes for error handling[1]. Among them, we highlight the `runtime_error` and the `out_of_range` classes, which we will explore next.

5.2.1 Class `runtime_error`

The `runtime_error` class from the standard library is one of the most commonly used for error handling. This class takes as an argument an object of type *string*, where an appropriate error message can be provided. Therefore, this is the simplest option for error handling when the goal is merely to display specific error messages for each type of error.

Any exception of type `runtime_error` that is thrown but not caught causes the immediate termination of the program, just as previously explained for the *throw* statement. Therefore, if we want to handle the exception, a *try...catch* block must be used. The general structure of a *try...catch* block for handling exceptions of type `runtime_error` is as follows:

```
try{
    //...
    if(error_condition_1)
        throw  runtime_error("Error_message_1");
    //...
    if(error_condition_2)
        throw  runtime_error("Error_message_2");
    //...
 }catch( const runtime_error& e){
        cout << e.what();
 }
```

Depending on the type of error, a specific message is passed as an argument to the `runtime_error` class. When an exception of type `runtime_error` is thrown, it is then caught by the *catch* block. The *catch* block uses the *.what()* method to return the message that was passed as an argument when the exception was thrown. That message is then displayed on the screen. The *.what()* method must be associated with an object of type `runtime_error`. In this case, that object was stored in the variable named `e`, although any other name could be used. The code below illustrates the use of the `runtime_error` class for handling errors in the program presented in the previous section.

[1]More information about these classes can be found at https://cplusplus.com/reference/exception/exception/.

```
int n, m, result = 0;

try{
    cout << "Value for n: ";
    cin >> n;
    if(!cin)
        throw  runtime_error("Incorrect reading of n");

    cout << "Value of m: ";
    cin >> m;
    if(!cin)
        throw  runtime_error("Incorrect reading of m");

    if(m == 0)
        throw  runtime_error("The value of m is zero");

    result = n/m;

}catch( const runtime_error& x){
    cout << x.what();
}
```

Since this example is very similar to the previous one, we will just examine the main differences. Start by noting the error messages defined in the various `runtime_error` instances. If the value of `n` is non-numeric, the error message is "Incorrect reading of n". Note that here, unlike in the previous example, the error message makes it possible to identify which of the entered values is non-numeric. Finally, if the value of `m` is 0, the error message is "The value of m is zero." There is only one *catch* block, the exception is stored in the variable `x`, and the error message is displayed on the screen using the *.what()* method.

5.2.2 Class `out_of_range`

The `out_of_range` class is used to handle errors related to accessing non-existent positions. Among the data types we have studied, the only ones that have associated positions are the `string` and `vector` types.

As we have seen earlier, we can use either the [ ] operator or the *.at()* method to access vector elements, and the main difference between them is that the *.at()* method performs validation of the position we are trying to access. When attempting to access a position in the vector that does not exist using the *.at()* method, an exception of type `out_of_range` is thrown, which may or may not be caught using a *try...catch* block.

Consider the following example:

```
vector<int> v(2);

try{
    v.at(0) = 7;
    v.at(1) = 2;
    v.at(2) = 5; // An exception is thrown

}catch( const out_of_range& e){
    cout << "Error: Inexistent position";
    // ou cout << e.what();
}
```

In this example, a vector of size two (with positions 0 and 1) is created. When attempting to access position 2, which does not exist, an exception of type `out_of_range` is thrown and caught by the *catch* block. To handle the exception, a custom error message can be printed (as in the first case), or the *.what()* method can be used, which provides information about the size of the vector and the position that was attempted to be accessed. Finally, it is important to note that there is no need to check the error condition using an *if* statement, since this validation is already performed in the implementation of the *.at()* method.

Chapter 6

Splitting a Project into Files

As a program grows, it becomes useful to split the code into different files, each containing independent components, to make the code more *modular*. The files created can be compiled individually, which allows new functionalities to be added to the program without needing to recompile the entire project. As a result, any new errors that may arise will be more easily detected, since they will likely be related to the newly added features and, therefore, confined to a single file.

One of the key advantages of modularity is that each of the different files created can be reused in multiple programs, avoiding the need to reimplement processes. For example, the packages from the *standard* library are implemented in a single module, which we cannot modify but can consult. Thus, whenever we want to use an existing feature from the *standard* library, we simply include it in the preamble of our program and directly call the available methods. As it will be explained later, we can also define our own modules, which can be shared across multiple programs.

To separate a project into files, we should start by creating a project and adding two files: a *header* file (also called *.h*) and a *source* file (or *.cpp*). To add the header file, right-click on the created project, select *Add new*, then choose *C/C++ Header File*, and finally define the module name. To add the source file, repeat the same process, select *C/C++ Source File* and assign the same name used for the header file. This process is illustrated in Figure 6.1.

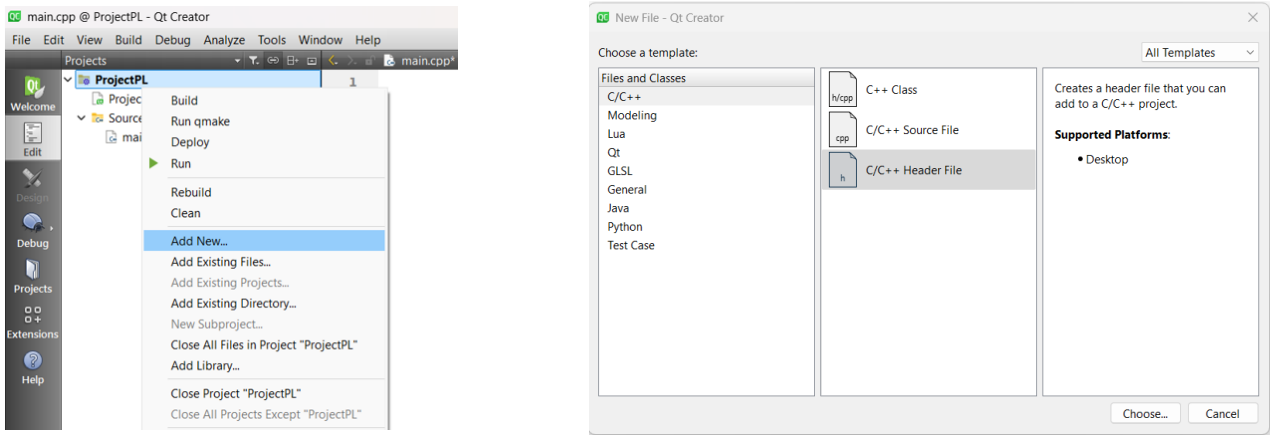


Figure 6.1: How to add a header file and a source file to a project.

After creating the header file and the source file, the structure of the project will be as shown in Figure 6.2. In this case, the created module is named *VectorsPL*. The header file has the structure shown in the figure, and our code should be written in the location indicated in the image. The directives `#ifndef`

`VECTORSPL_H` and `#define VECTORSPL_H` are used to define the content of the header file if it has not already been defined. If it has already been defined (via the instruction `#include "VectorsPL.h"`), the content of this file is ignored. The source file is created empty.

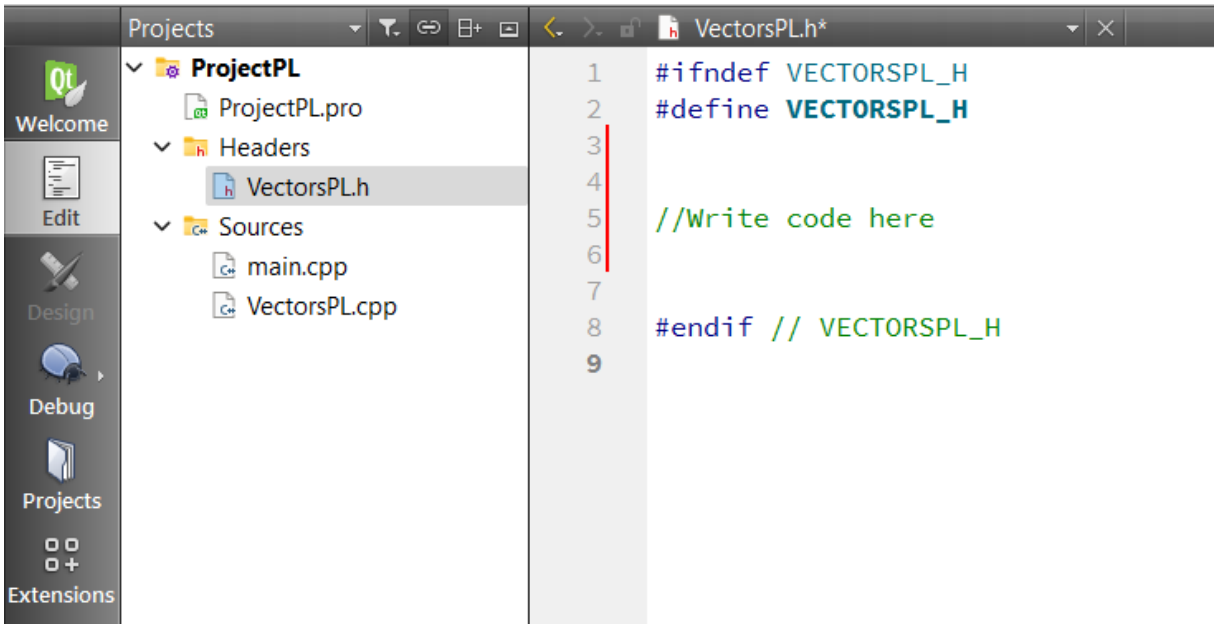


Figure 6.2: Project structure with header and source files.

The header file should contain only the method declarations, which is justified by two main reasons: reducing compilation time and allowing the user to easily identify the components of the module. Thus, the header file can be seen as a "table of contents" of a book. The "content" of the book, which in our case corresponds to the methods' definitions, is found in the source file. Consider the following example.

Header File

```
#ifndef VECTORSPL_H
#define VECTORSPL_H

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

void print( const vector<int>& );

#endif // VECTORSPL_H
```

Source File

```
#include "VectorsPL.h"

void print( const vector<int>& v){
    for(int i = 0; i<v.size(); ++i){
        if(i == 0)
           cout << "(" << v[i];
        else if(i == v.size() - 1)
           cout << ", " << v[i] << ")";
        else
           cout << ", " << v[i];
    }
}
```

Main Program

```
#include "VectorsPL.h"

int main(){
    vector<int> u = {1, 2, 3};
    print(u);
    return 0;
}
```

In this example, a module (*VectorsPL*) is created that contains, in the header file, the declaration of a function *print* with return type *void* that receives a constant reference to a vector as an argument. The definition of this function is written in the source file. The `#include "VectorsPL.h"` directive is required at the top of this file to establish the link between the header and source files.

Once the module is defined, it can be used in any program by simply including it using the instruction `#include "VectorsPL.h"`. Note that it is not necessary to include in the main program the packages that are already included in the header file, because by including the header file, we are automatically including all the packages it contains.

After including the module, all the methods it contains can be called directly, as demonstrated with the instruction `print(u)`. Modularization makes the main program much more compact and readable, as illustrated in this example.

6.1 Namespaces

More complex programs often require the inclusion of multiple modules, which may be created independently by different people. This can lead to elements being declared with the same name in different modules — for example, two functions with the same name. When including multiple header files in the main program, there is the risk of having repeated declarations, which is not allowed by the compiler due to a naming conflict. One way to avoid this problem is by using *namespaces*.

A *namespace* is a named scope where various elements can be declared. When using *namespaces*, access to their elements is done by explicitly indicating which *namespace* they belong to. This way, even if there are two elements with the same name in different modules, access to each one is done differently, thereby avoiding naming conflicts.

Consider the following modules `M1` and `M2`, where a *print* function is declared. The first function belongs to *namespace* `X`, and the second to *namespace* `Y`. Therefore, to define these functions in their respective source files, it is necessary to specify which *namespace* they belong to by using the `X::` and `Y::` prefixes before their names.

Header File M1

```
#ifndef M1_H
#define M1_H

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

namespace X{
     void print(const vector<int>&);
}

#endif // M1_H
```

Header File M2

```
#ifndef M2_H
#define M2_H

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

namespace Y{
     void print(const vector<int>&);
}

#endif // M2_H
```

Source File M1

```
#include "M1.h"

void X::print(const vector<int>& v){
     // ...
}
```

Source File M2

```
#include "M2.h"

void Y::print(const vector<int>& v){
     // ...
}
```

Main Program

```
#include "M1.h"
#include "M2.h"

int main(){
     vector<int> u = {1, 2, 3};
     print(u); // ERROR
     X::print(u); // CORRECT
     Y::print(u); // CORRECT
     return 0;
}
```

In the main program, where both header files are included, we can then call both functions by specifying the *namespace* to which they belong. This way, the function we want to use is clearly identified.

Always specifying the *namespace* an element belongs to makes the program more verbose and harder

to read, but it is essential when there are elements with the same name in different *namespaces*. However, when this is not the case, we can simplify the syntax with the *using* directive. Adding the statement `using namespace X;` to the preamble allows direct access to the elements of the *namespace* `X` without needing to use `X::` for each of them, since we have already indicated that we are using elements from *namespace* `X`. Note that this is exactly what we have been doing with the elements from the *namespace* `std` when we use the instruction `using namespace std;`. This *namespace* includes elements such as *cout*, *cin*, *string*, and *vector*. Therefore, to write something like

```
vector<string> v;
// ...
cout << v[0];
```

Without using the `using namespace std;` instruction, we would have to write

```
std::vector<std::string> v;
// ...
std::cout << v[0];
```

which clearly makes the program harder to read. Using the `std::` instruction only has the advantage of clearly identifying which *namespace* the elements belong to.

In the same header file, we can define multiple *namespaces*. Furthermore, we can also define *namespaces* inside other *namespaces*, as illustrated in the example below.

Header File M1

```
#ifndef M1_H
#define M1_H

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

namespace X{
    void print1(const vector<int>&);

     namespace Z{
        void print2(const vector<int>&);
     }
}

namespace W{
     void print3(const vector<int>&);
}

#endif // M1_H
```

Main Program

```
#include "M1.h"

int main(){
    vector<int> u = {1, 2, 3};
    X::print1(u);
    X::Z::print2(u);
    W::print3(u);
    return 0;
}
```

Note that the functions `print1`, `print2`, and `print3` could have the same name since they are associated with different *namespaces*. Furthermore, all these functions could be defined in the same source file, with each one being identified in a similar way as in the main program, that is, using the directives `X::`, `X::Z::`, and `W::`.

6.2 Redefinition of Data Types - Type aliases

In C++, the *using* keyword allows you to define an alternative name (an alias), usually simpler, for a given data type. This alias can then be used throughout the code instead of the original data type. The general syntax of the *using* keyword to define aliases is:

```
using alternative_name = data_type;
```

In the example below, we present a header file that contains the declaration of two functions for handling matrices. The first (of type *void*) prints a matrix, while the second returns a matrix that is the sum of the two matrices received as arguments. In the first code snippet, an alias was not used, so `vector<vector<int>>` must be written every time that data type is referenced. In the second code snippet, an alias named `matrix` matrix is defined for the type `vector<vector<int>>`. This makes the code shorter and easier to read than the first snippet, which is the main benefit of using type aliases.

Header File M1 - Without type aliases

```
#ifndef M1_H
#define M1_H

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

void print(const vector<vector<int>>&);
vector<vector<int>> add(const vector<vector<int>>&, const vector<vector<int>>&);

#endif // M1_H
```

Header File M1 - With type aliases

```
#ifndef M1_H
#define M1_H

#include <iostream>
#include <vector>
using namespace std;
using matrix = vector<vector<int>>; //Defines "matrix" as "vector<vector<int>>"

void print(const matrix&);
matrix add(const matrix&; const matrix&);

#endif // M1_H
```

Chapter 7

Classes

As we know, C++ provides several primitive data types such as *int*, *double*, *char*, etc. Due to their simplicity, these data types do not allow us to represent *objects* that we frequently encounter, such as vectors, fractions, complex numbers, cars, books, and so on. These objects have: (i) attributes (for example, license plate, color, and number of doors in the case of a car); and (ii) functionalities (start, brake, etc.), which can be represented in C++ through *classes*. A class is a new user-defined data type used to represent and manipulate objects that cannot be represented or manipulated using primitive data types. Classes make clearer which object we are applying a certain functionality to, and are therefore the foundation of any object-oriented programming language. Classes do not replace what we have learned so far; rather, they provide us with an additional tool to deal with code complexity, allowing code to be written in a more modular way.

An example of a class that we already know well is the *vector* class. This class was created to represent vectors and therefore contains methods to access their properties, such as the *size()* method, as well as functionalities to manipulate them, such as the *push_back()* method.

Suppose we want to write a program that deals with complex numbers. Although the C++ standard library already provides a type for complex numbers[1], we will define our own class, `Complex`, to represent and manipulate complex numbers. This class will serve as a running example throughout this chapter.

A class should be declared in a header file and defined in a source file so that it can be easily reused. To create a class, we right-click on the project and then select *Add New*, as illustrated in Figure 6.1 shown in the previous chapter. After that, we select the *C++ Class* option (see Figure 6.1) and give the class a name. The code below illustrates the structure of the header and source files created for the `Complex` class.

[1]For more information, see `https://en.cppreference.com/w/cpp/numeric/complex.html`.

Header File

```
1.  #ifndef COMPLEX_H
2.  #define COMPLEX_H
3.
4.  class Complex{
5.
6.  public:
7.      Complex();
8.  };
9.
10. #endif // COMPLEX_H
```

Source File

```
1.  #include "complex.h"
2.
3.  Complex::Complex(){
4.
5.  }
```

Note that the class declaration ends with a ";" (see line 8 of the header file) and that when creating a *C++ Class*, the source file begins with the *include* of the header file.

Any new data type created by the user is based on other data types. A complex number has the form $a+bi$, where a is the real part and b is the imaginary part. Thus, a complex number can be represented by two variables of type *double* corresponding to these real and imaginary parts. In a class, the variables used to represent an object are the class *attributes*, which can be of any type, including other classes.

A class can also have several *methods* that allow manipulating the object of the class. Moreover, a class always has at least one *constructor*, which initializes its attributes when a new object is created. The constructor has the same name as the class and may or may not have arguments. Thus, a constructor can be seen as a function without a return type. The attributes, constructor(s), and methods of a class are referred to as the class *members*.

A class can have *public* and *private* members. Public members can be accessed both inside and outside the class, while private members can only be accessed within the class, that is, inside its methods. The declaration of a class member as public or private depends on its purpose. However, since the attributes are the structural elements of the class, they should be private to prevent them from being modified directly by the user. To access and modify (get and set) private attributes, it is therefore necessary to create public methods, as illustrated in the code below.

Header File

```
#ifndef COMPLEX_H
#define COMPLEX_H

  //Write all necessary includes

  class Complex
  {
  private:
     //Attributes
      double Real;
      double Im;

  public:
     //Constructors
      Complex();
      Complex(double , double);

     //Methods
      void SetReal(double);
      void SetIm(double);
      double GetReal() const;
      double GetIm() const;

  };

#endif // COMPLEX_H
```

Source File

```
#include "complex.h"

// Default constructor
Complex::Complex(){
      Real = 0;
      Im = 0;
}

// Other constructor
Complex::Complex(double a, double b){
      Real = a;
      Im = b;
}

 //Methods
 void Complex::SetReal(double x){
      Real = x;
 }

 void Complex::SetIm(double x){
       Im = x;
 }

 double Complex::GetReal() const{
       return Real;
 }

 double Complex::GetIm() const{
      return Im;
 }
```

The `Complex` class contains two variables of type *double* as private attributes, one named `Real` and the other named `Im`, which store the real and imaginary parts of the complex number, respectively. Note that class attributes appear in red in QT Creator. Since the attributes are private, the class provides two public methods that allow modifying each of these attributes: the `SetReal` and `SetIm` methods. There are also two other public methods (`GetReal` and `GetIm`) that allow accessing the values of the class attributes. It is important to mention that these two methods, as well as all methods that do not modify the class attributes, should be defined as constant. To define a method as constant, we write the keyword *const* after its arguments.

This class contains two constructors: the *default constructor*, which takes no arguments, and another constructor that takes two arguments of type *double*. The first constructor receives no arguments and is therefore defined to initialize the class attributes with their *default* value, zero. The second constructor takes two arguments and uses them to initialize the class attributes. The two constructors mentioned can alternatively be implemented by listing the class attributes as follows:

Source File

```
// Default constructor
Complex::Complex(): Real(0), Im(0){ }

// Other constructor
Complex::Complex(double a, double b): Real(a), Im(b){ }
```

where the meaning of, for example, `Real(a)` is similar to `Real = a`. The purpose of the constructors is to initialize the attributes of the object the class represents at the time of its creation. As we saw in the example, a class can have multiple constructors, as long as they have different arguments.

Now we will see how this class can be used in the main program, within the *main* function.

Main Program

```
1.  #include "complex.h"
2.
3.  int main(){
4.     Complex z1;
5.     Complex z2(3, 2);
6.
7.      z1.Real = 1;    //ERROR
8.      z1.Im = 5;      //ERROR
9.      z1.SetReal(7);
10.     z1.SetIm(0);
11.
12.     //Print z2
13.        cout << z2.GetReal() << "+" <<  z2.GetIm() << "i";
14.
15.     return 0;
16.  }
```

To use a class in a file where it was not defined, you must *include* its header file (line 1). On line 4, an object `z1` of type `Complex` is created. At this point, the default constructor is implicitly called, so the real and imaginary parts of `z1` will be initialized to zero. On line 5, a new object `z2` of type `Complex` is created. However, in this case, since two arguments are provided, the second constructor is implicitly called, resulting in `z2 = 3 + 2i`.

To access the public methods of a class, we use a dot (".") after the class object name, followed by the method name, as done in lines 9 and 10. In line 9, the `SetReal` method is called with the argument 7, which allows changing the real part of the complex number `z1` to 7. Since the class attributes are private, they cannot be accessed outside the class, as illustrated in lines 7 and 8. In line 13, the complex number `z2` is printed on the screen in the form $a+bi$, so the methods `GetReal()` and `GetIm()` are used to access its attributes.

It is important to emphasize that, when used outside the class, a class's methods are always associated with an object of that class and therefore cannot be called without being applied to a class object. For example, the only way to use the `GetReal` method is by writing `x.GetReal()`, where `x` is an object of type `Complex`. Thus, writing something like `.GetReal()` or `GetReal()` outside the class is not possible.

A class can contain methods to manipulate the type of object it represents. In the case of the `Complex` class, it may make sense to have, for example, a public method that allows printing a complex number in the form $a+bi$. This class can also include public methods to calculate the modulus of a complex number, to check whether a complex number is purely imaginary, among others. These methods can take objects of the class as arguments and return objects of the class. The code below illustrates the inclusion of four functions (`Print`, `PureIm`, `Symmetric`, and `Sum`) in the `Complex` class.

Header File

```
#ifndef COMPLEX_H
#define COMPLEX_H

//Write all the necessary includes

class Complex{
private:
  //Attributes
  double Real;
  double Im;

public:
  //Constructors
  Complex();
  Complex(double , double);

  //Methods
  void SetReal(double);
  void SetIm(double);
  double GetReal() const;
  double GetIm() const;
  void Print() const;
  bool PurelyIm() const;
  Complex Symmetric() const;
  Complex Sum(const Complex&) const;
};




#endif // COMPLEX_H
```

Source File

```
#include "complex.h"

// ...
// Remaining definitions
// ...

void Complex::Print() const{
    cout << Real;
    if ( Im >= 0 )
        cout << " + " << Im << "i";
    else
        cout << Im << "i";
}

bool Complex::PureIm() const{
    if (Real == 0 and Im != 0)
        return true;
    else
        return false;
}

Complex Complex::Symmetric() const{
    Complex z;
    z.Real = -1*Real;
    z.Im = -1*Im;
    //or
    Complex z(-1*Real, -1*Im);

    return z;
}

Complex Complex::Sum(const Complex& z) const{
    Complex zSum(Real + z.Real, Im + z.Im);
    return zSum;
}
```

The `Print` function, as the name suggests, prints the `Complex` object in the form "a+bi". The `PureIm` method checks whether the complex number is a pure imaginary number. The `Symmetric`

function returns a new complex that is the symmetric of the class object. Finally, the `Sum` method returns a complex number that is the sum of the class object with another `Complex`.

These functions do not modify the attributes of the class and, therefore, are defined as constant. The `Symmetric` function returns an object `z` of type `Complex` that is the symmetric of the object it is associated with. In this function, the `Complex z` is created using the class constructor and is then returned. The `Sum` function takes as an argument a constant reference to an object of type `Complex`, because it is a non-primitive data type that is not modified by the function. This function creates a new object of type `Complex`, which results from the sum of the class object with the `Complex z`, and returns it.

The main program below illustrates the use of the `Print`, `Symmetric`, and `Sum` methods. Initially, the constructor is used to create the `Complex` object `z1`, which is printed on the screen as *3+2i*. Next, a new `Complex z2` is created, which is the symmetric of `Complex z1`. This new `Complex` is then printed in the form *-3-2i*. Finally, a new object `z3` of type `Complex` is created, resulting from the sum of `z1` with `z2`, and it is printed on the screen.

Main Program

```
#include "complex.h"

int main(){
   Complex z1(3, 2);
   z1.Print();
   Complex z2 = z1.Symmetric();
   z2.Print();
   Complex z3 = z1.Sum(z2);
   z3.Print();
   return 0;
}
```

Since the sum is commutative, we would obtain the same result in the previous program by writing `Complex z3 = z2.Add(z1);`. A more intuitive `Sum` function would take two complex numbers as arguments and return their sum. In that case, it could be called as follows: `Complex z3 = Add(z1, z2);`. This is possible by creating global functions, that is, functions declared in the class files but not as class members.

Header File

```
#ifndef COMPLEX_H
#define COMPLEX_H

//Write all necessary includes

class Complex{
  //Class Complex previous defined without the function Sum
};

Complex Sum(const Complex&, const Complex&);

#endif // COMPLEX_H
```

Source File

```
#include "complex.h"

// ...
// Definition of class members
// ...

Complex Sum(const Complex& z1, const Complex& z2){
    Complex zSum(z1.GetReal()+z2.GetReal(), z1.GetIm()+z2.GetIm());
    return zSum;
}
```

Since the `Sum` function is not a class member, its declaration is made outside the class. Therefore, it is not necessary to include the identifier "`Complex::`" before its name in the source file. For the same reason, access to the class attributes (defined as private) must be done using the *Get* functions. The call of the `Sum` function in the main program is made as follows:

```
Complex z3 = Sum(z1, z2);.
```

To conclude this chapter, consider a new class that represents a person. This class has as attributes the person's name and age. In addition to the default constructor, the `Person` class has a parameterized constructor and functions to access its attributes.

Header File

```
#ifndef PERSON_H
#define PERSON_H

//Write all necessary includes

class Person{
private:
    string Name;
    int Age;

public:
    //Constructors
    Person();
    Person(const string&, int);

    //Methods
    const string& GetName() const;
    int GetAge() const;
};

#endif // PERSON_H
```

Source File

```
#include "person.h"

Person::Person(): Name(" "), Age(-1){ }

Person::Person(const string& name, int a):
   Name(name), Age(a){ }

const string& Person::GetName() const{
    return Name;
}

int Person::GetAge() const{
    return Age;
}
```

In the analysis of this example, we will focus only on the main differences compared to the previous examples. We start by looking at the definition of the `GetName` method, more specifically at its return type. This method returns a constant reference to a *string*, because the function should return the `Name` attribute itself and not a copy of it (which would be the case if the return type were simply *string* instead of a constant reference). Thus, attributes of non-primitive types should be returned as constant references to avoid unnecessary copies. Note that the `Age` attribute is of type *int* (a primitive data type) and therefore it does not need to be returned by reference in the `GetAge` function.

To summarize, in the context of classes, the reserved word *const* can appear in three different situations:

i. in the arguments of the methods, where its purpose is similar to that discussed in Chapter 4;

ii. associated with methods, indicating that the method will not modify the class attributes;

iii. in the return type of methods that return class attributes of a non-primitive data type, to ensure that these attributes are returned without being modified and without creating unnecessary copies.

Chapter 8

Operator Overloading

Most of the operators we saw in Chapter 1 are only defined for primitive data types. In fact, we have already used the operators "+", "=", "==" and "<<" with variables of type *int*, for example. Since a class is a new data type created by the user, the usual operators are not defined for objects of those classes. However, it is possible to define "versions" of the operators that allow them to work when applied to objects of a particular class. This is called *operator overloading*.

Consider the `Complex` class created in the previous chapter as an example, which includes the method `Symmetric`. The use of this method outside the class is done with the following instruction:

```
Complex w = z.Symmetric();
```

where `z` is an object of type `Complex` and `w` is its symmetric (or additive inverse). Now, the unary minus operator "-" already exists for numerical data types, but not for the `Complex` data type, although it can be overloaded for that purpose. After overloading the operator, it becomes possible to write

```
Complex w = -z;
```

making the code more readable and intuitive. The same applies, for example, to the method `Print`, which is also part of the `Complex` class. The direct alternative to this method is the output operator `<<` which, once overloaded, allows printing a `Complex` object using the second notation shown below.

```
//Using Print method
    z.Print();

//Using operator <<
    cout << z;
```

Several operators can be overloaded in C++, some of which are presented in the following table.

Unary Operators		**Binary Operators**	
Increment	`++`	Binary Arithmetic	`+, -, *, /,%`
Decrement	`--`	Arithmetic Assignment	`+=, -=, *=, /=, %=`
Unary Minus (or symmetric)	`-`	Relational	`<, >, <=, >=, !=, ==`
Logical Not	`!`	Output and Input	`<<, >>`
		Subscript and Parenthesis	`[], ()`

Unary and binary operators can be overloaded. Unary operators have a single operand, which will be an object of the class. Binary operators act on two operands, with at least one of them being an object of the class.

We will now see how to overload some of these operators using the `Complex` class as an example. It is important to note that, as with functions, operator declarations should be placed in the header file, and their definitions in the source file. The code below illustrates a possible header file for the `Complex` class where several operators are declared.

Header File

```
//Instructions to define the header file and the necessary includes
class Complex {
private:
    double Real;
    double Im;

public:
    Complex( );
    Complex(double , double);

    void SetReal(double);
    void SetIm(double);
    double GetReal( ) const;
    double GetIm( ) const;

    //Unary minus operator: to write -z
      Complex operator-( ) const;
    //Plus assignment operator: to write z1+=z2
      Complex& operator+=( const Complex& );
    //(Prefix) Increment operator: to write ++z
      Complex& operator++( );
    //(Postfix) Increment operator: to write z++
      Complex operator++( int );
    //Parenthesis operator: to write z(a)
       double operator()( double a) const;
    //Logical Not operator: to write !z
       bool operator!( ) const;
};

 //Binary plus operator: to write z1+z2
    Complex operator+( const Complex& , const Complex& );
 //Comparison operator: to write z1==z2
    bool operator==( const Complex& , const Complex& );
 //Output operator: to write cout << z
    ostream& operator<<( ostream& , const Complex& );
 //Input operator: to write cin >> z
    istream& operator>>( istream& , Complex& );
```

Note that some operators are members of the class and others are not. There are operators that can be defined either outside or inside the class, and the way they are declared and defined depends on

the chosen approach. However, to avoid making this chapter overly lengthy, we will adopt the following logic:

- binary arithmetic, relational, output, and input operators will be defined outside the class;
- arithmetic assignment, increment/decrement, unary minus, logical not, subscript, and parentheses operators will be defined inside the class.

We will now see how to define each of these operators in the class's implementation file.

Unary Minus (Symmetric) Operator

The addictive inverse (symmetric) of a complex number is also a complex number, so the return type of this operator is an object of type `Complex`. This operator is defined inside the class, which is why we need to use the identifier `Complex::`. When writing `z1 = -z2`, where `z1` and `z2` are objects of type `Complex`, we are applying the unary minus operator "-" to `z2`. The result of `-z2` is stored in `z1`, but the value of `z2` remains unchanged. Therefore, when implementing the unary minus operator, we must create a new `Complex` with real and imaginary parts equal to the negatives of the corresponding parts of the object to which the operator was applied.

Source File

```
Complex Complex::operator-( ) const{
      Complex new;
      new.Real = -1*Real;
      new.Im = -1*Im;
      return new;

      //or
      Complex new = Complex( -1*Real, -1*Im );
      return new;

      //or simply
      return Complex( -1*Real, -1*Im );
}
```

Plus Assignment Operator

Unlike the unary minus operator, the assignment arithmetic operator for addition/assignment requires two operands, where one is modified and the other remains unchanged. For example, when writing `a += b`, we are adding `b` to `a`, and thus only the value of `a` is modified. Therefore, this operator receives as an argument a constant reference to the object that will not be changed (operand `b`) and returns a (non-constant) reference to the object that was modified (object `a`). The instruction `return *this;` means "return a reference to this", where "this" refers to the object the operator was applied to, that is, object `a`.

Source File

```
Complex& Complex::operator+=(const Complex& b ){
     Real += b.GetReal();
     Im += b.GetIm();
     return *this;
}
```

The declaration and definition of the other assignment arithmetic operators are similar to what was presented here for the `+=` operator and therefore will be omitted from this lecture notes. It should be noted, however, that the remainder operator (`%`) is not valid for complex numbers since their attributes are of type *double*, meaning that overloading the `%=` operator does not make sense for this data type.

Increment Operators

As we saw earlier, there is the prefix increment operator and the postfix increment operator. When we use the prefix increment operator (`++a`), the object is first incremented and then returned. When we use the postfix (`a++`) increment operator, a copy of the object is made first, and only then is the object incremented, with the copy of the object (which was not incremented) being returned in the end. As illustrated in the code below, what distinguishes the declaration of the increment operators is the *int* argument in the postfix operator. Note that the *int* argument serves only to distinguish which operator we want to overload and not to indicate that the postfix operator requires an *int* argument.

Unlike with other data types, the meaning of the increment operators for objects of type `Complex` is not clear and may not even make sense. However, to explain how these operators should be declared and defined, we will assume that they increase both the real and imaginary parts of the `Complex` object by one unit.

Source File

```
//Prefix increment operator (++a)
    Complex& Complex::operator++( ){
         ++Real;
         ++Im;
         return *this;
    }

//Postfix increment operator (a++)
    Complex Complex::operator++( int ){
         Complex aux(Real,Im);
          ++Real;
          ++Im;
         return aux;
    }
```

The implementation of the decrement operators is similar and will therefore be omitted.

Arithmetic Binary Operators

Arithmetic operators perform an arithmetic operation between two objects of the class, which they receive as arguments, and return a new object of the class. Since these operators do not modify the objects passed as arguments, such objects should be passed by constant reference. It is also important to note that, as these operators are defined outside the class, the `Complex::` identifier is no longer needed.

In the code below, the addition operator is defined, and the definition of the other arithmetic operators (minus, multiplication, division) is similar.

Source File

```
Complex operator+( const Complex& z1, const Complex& z2 ){
     double new_real = z1.GetReal() + z2.GetReal();
     double new_im = z1.GetIm() + z2.GetIm();
     Complex aux( new_real, new_im );
     return aux;

      // or simply...
      return Complex( z1.GetReal()+z2.GetReal(), z1.GetIm()+z2.GetIm() );
}
```

Relational Operators

Relational operators allow comparison between two objects of the class and return a result of type *bool*. Just like the arithmetic operators, relational operators also do not modify the objects they operate on. Thus, these objects should be passed by constant reference. In the code below, the `==` operator is defined for objects of type `Complex`. The definition of the remaining relational operators would follow a similar structure. However, it is important to mention that the operators `<`, `>`, `<=`, and `>=` do not have a clear meaning for objects of type `Complex`.

Source File

```
bool operator==( const Complex& z1, const Complex& z2 ){
     if ( z1.GetReal()==z2.GetReal() && z1.GetIm()==z2.GetIm() )
         return true;
     else
         return false;
}
```

To evaluate whether two `Complex` objects are different, we cannot use the statement `z1 != z2`, since we have not overloaded the `!=` operator. However, since the result of this operator is the negation of the `==` operator, we could use `==` to perform this check as follows: `!(z1 == z2)`.

Output Operator

The output operator (`<<`) is a binary operator that takes two arguments, namely a reference to an `ostream` object and a constant reference to an object of the class, and returns a reference to an `ostream` object. The class `ostream` is from the *standard* library and stands for *output stream*. This class enables

writing and formatting character sequences. Note that the definition of the output operator is quite similar to the definition of the `Print` function, with the only difference being the use of an `ostream` object instead of the traditional *cout*.

Source File

```
ostream& operator<<( ostream& output, const Complex& z){
     if ( z.GetIm() >= 0)
          output << z.GetReal() << "+" << z.GetIm() << "i";
     else
          output << z.GetReal() << z.GetIm() << "i";

    return output;
}
```

Given what we have learned about functions, a question arises: why is the `<<` operator not of type *void* if it takes an argument by reference? The answer is simple: the `<<` operator returns a reference so that we can chain instructions. That is, we can do something like `cout << z << endl;`, where `z` is an object of type `Complex`. If the return type of the operator was *void*, we would end up with `void << endl;`, which the computer cannot interpret. By returning the *ostream* reference, we get `cout << endl;`, which the computer does recognize.

Input Operator

The input operator (`>>`) is also a binary operator that receives two arguments, namely a reference to an `istream` object and a reference to an object of the class, and returns a reference to an `istream` object. The class `istream` is from the *standard* library and stands for *input stream*. This class allows reading sequences of characters. In the code below, the input operator is defined for objects of type `Complex`, assuming that these objects are entered by the user in the form $a \pm bi$. According to that format, the user should first input the real part of the complex number, which is stored in variable `a`. Then, a character is entered, expected to be either the `+` or `-` sign, and it is stored in `c1`. Finally, the imaginary part of the complex number is entered (stored in variable `b`) along with a character, expected to be '*i*', stored in `c2`.

Source File

```
istream& operator>>( istream& input, Complex& z){
      char c1, c2;
      double a, b;
      input >> a >> c1 >> b >> c2;
      if( c1 == '-')
          b = -b;

     if( !input || (c1!='-' && c1!='+') || c2!='i' )
         //throw exception

      z.SetReal(a);
      z.SetIm(b);
      return input;
}
```

The declaration of the input operator >> is very similar to that of the output operator <<. The only differences are the use of the *istream* class instead of *ostream*, and the `Complex` object being passed-by-reference instead of constant reference. This happens because the input operator will modify the `Complex` object, as it changes the values of its real and imaginary parts to those entered by the user.

Parenthesis Operator

The parenthesis operator `()` can take any number of arguments and return any data type. This flexibility allows it to be used in many situations, two of which are illustrated in the example below for the previously defined `Complex` class.

Source File

```
double Complex::operator()( double a) const{
      return Real*a + Im;
}

Complex Complex::operator()( double a, double b ) const{
      return Complex( Real*a , Im*b );
}
```

In the first example, which corresponds to the one declared in the header file, the `()` operator takes only one argument of type *double* and returns the result obtained by multiplying the real part by the received value and adding the imaginary part of the `Complex` object. Therefore, the return type is *double*.

In the second case, the `()` operator is used to compute a new `Complex` object with real and imaginary parts equal to the original object's parts multiplied by real constants. Thus, it takes two *double* values, which are multiplied by the real and imaginary parts of the original `Complex`, resulting in a new `Complex` object.

Logical Not Operator

The logical not operator `!` does not take any arguments and its return type is *bool*. In the example below, the negation operator is defined for the `Complex` class.

Source File

```
bool Complex::operator!( ) const{
      if (Real == 0 and Im == 0 )
            return true;
      else
            return false;
}
```

In the example shown, the negation operator is used to check whether a `Complex` object is zero, meaning that both its real and imaginary parts are zero.

After overloading the previous operators, they can be used as illustrated below.

Main Program

```
#include "complex.h"

int main(){
   Complex z1(3,2), z2;
   cout << "z2: ";
   cin >> z2;                                    //Input operator

   Complex z3 = z1 + z2;                         //Plus operator
   cout << z1 << " + " << z2 << " = " << z3;     //Output operator
   ++z1;                                         //Increment operator
   Complex z4 = -z1;                             //Unary minus operator
   z4 += z2;                                     //Plus-assignment operator

   if(z1 == z2)                                  //Equality operator
        cout << "They are equal";
   else
        cout << "They are different";

    double a = z1(2);                             //Parenthesis operator example 1
    Complex z5 = z2(2, 3);                        //Parenthesis operator example 2

    if ( !z1 )                                    //Logical not operator
        cout << "Zero complex";
    else
        cout << "Non-zero complex";

   return 0;
}
```

Subscript Operator

Usually, we use the subscript operator `[]` to access elements of vectors, so overloading this operator is particularly useful when the class contains an attribute of type *vector*. To illustrate the overloading of the `[]` operator, consider a fictional class `VectorPL` that has a private vector of *strings* as an attribute, with the header file shown below.

Header File

```
#ifndef VECTORPL_H
#define VECTORPL_H
//Include necessary headers

  class VectorPL {
  private:
      vector<string> V;

  public:
      //Constructor and other public members

      //Subscript operator - non-constant version
        string& operator[]( int );

      //Subscript operator - constant version
        const string& operator[]( int ) const;
  };

#endif // VECTORPL_H
```

As previously mentioned, the overloading of the `[]` operator is done inside the class. Typically, two versions are implemented: a constant version and a non-constant version. We begin by analyzing the non-constant version. This version takes an integer argument that corresponds to a position in the vector and returns a reference to the vector element at that position. Thus, this version allows modification of the vector elements and is implemented as shown below.

```
//non-constant version
string& VectorPL::operator[]( int i ){
   return V[i];
}
```

It is important to emphasize that the use of a reference in the return type of this version is not related to the fact that we are returning a *string* (a non-primitive type), but rather to the fact that we want the returned object to be modifiable. Therefore, this version of the `[]` operator always returns a reference, even if the return type is a primitive data type.

The `[]` operator, once overloaded, is often used in the definition of other methods/operators. Some of those methods receive constant objects as arguments, i.e., constant references to class objects. Constant objects can only be handled by methods/operators that are also constant, and so they cannot be

manipulated by the non-constant version of the `[]` operator. To overcome this situation, a constant version of the `[]` operator must be implemented. This version — shown below — returns a constant reference to an element of the vector and therefore does not allow modification of that element.

```
//constant version
const string& VectorPL::operator[]( int i ) const{
   return V[i];
}
```

Finally, it is important to note that no validation of the argument `i` is required in the implementation of either version of the `[]` operator, since this operator does not perform such validation, as we have seen in the case of standard vectors. Keep in mind that when overloading an operator, we should always preserve its original properties.

Subscript Operator for Matrices

As we have seen, a matrix is a vector of vectors, that is, a vector where each of its elements is itself a vector. For example, the matrix

$$m = \begin{bmatrix} 5 & 2 & 1 \\ 8 & 0 & 6 \\ 7 & 2 & 3 \\ 1 & 0 & 1 \end{bmatrix}$$

can be defined in C++ as:

```
vector<vector<int>> m = { {5, 2, 1}, {8, 0, 6}, {7, 2, 3}, {1, 0, 1} }
```

This means that `m` is actually the "principal" vector with four elements (`m[0] = {5, 2, 1}`, `m[1] = {8, 0, 6}`, `m[2] = {7, 2, 3}`, and `m[3] = {1, 0, 1}`), each of which is a "secondary" vector of size three. The overload of the `[]` operator for an object of a class that has a matrix as an attribute only allows access to the principal vector of that matrix, so the return type of the operator will be a vector of elements with the same type as the secondary vector, as illustrated in the next example.

In the header and source files of the example, a class `XPTO` is created that contains a matrix of *doubles* as an attribute, and the `[]` operator is defined in both its constant and non-constant versions. This operator allows direct access to the `Matrix` attribute of an object of type `XPTO`, as shown in the definition of the output operator. Note that `x` is an object of type `XPTO` and not a vector or matrix, so without overloading the `[]` operator, it would not be possible to write something like `x[i]`. However, once the `[]` operator is defined, it becomes possible to write `x[i]`, and consequently, `x[i][j]`. When writing `x[i][j]`, the `[]` operator defined in the class is first called, which returns a vector of *doubles*, i.e., it returns `x[i]`. Since `x[i]` is a vector of *doubles*, access to its elements can be done directly using the standard library's built-in `[]` subscript operator for vectors, which then allows retrieval of the element at position j of the vector `x[i]`, that is, `x[i][j]`. Thus, there is no need to implement the subscript operator for the secondary vector.

Header File

```
#ifndef XPTO_H
#define XPTO_H
//Include necessary headers

  class XPTO {
  private:
      vector<vector<double>> Matrix;

  public:
      //Constructor and other public members

      //Subscript operator - non-constant version
        vector<double>& operator[]( int );

      //Subscript operator - constant version
        const vector<double>&  operator[]( int ) const;
  };

  ostream& operator<<( ostream& , const XPTO& );

#endif // XPTO_H
```

Source File

```
//Subscript operator - non-constant version
    vector<double>& XPTO::operator[]( int i ){
        return Matrix[i];
    }

//Subscript operator - constant version
    const vector<double>& XPTO::operator[]( int i ) const{
        return Matrix[i];
    }

//Output operator
    ostream& operator<<( ostream& output, const XPTO& x){
        for(int i = 0; i < x.N_rows(); ++i){ //Method N_rows() must be defined
            for(int j = 0; j < x[i].size(); ++j){
                output << x[i][j] << " ";
            }
            output << endl;
        }
        return output;
    }
```

Chapter 9

Inheritance and Polymorphism

Inheritance is the ability to create new classes (called derived or child classes) from existing ones (called base or parent classes), with the members of the parent classes being *inherited* by the derived classes. In a parent class, we can have *public*, *private*, or *protected* members. Protected members cannot be accessed outside the class they were defined, similarly to private members. The difference between private and protected members only exists in the context of inheritance: derived classes have access to the protected members of the parent class, but not to its private members.

The indication that a class (derived class) inherits from another class (parent class) is done as follows:

Header File

```
#ifndef DERIVED_CLASS_H
#define DERIVED_CLASS_H

class  Derived_Class: access_type Parent_Class {
    //...
};

#endif // DERIVED_CLASS_H
```

The `access_type` defines the type of access to the members of the parent class and can be *public*, *private*, or *protected*. Private members of the parent class can never be accessed by derived classes, regardless of the access type used. Therefore, the difference between the three access types only concerns the public and protected members of the parent class:

i) *private*: the derived class inherits all public and protected members of the parent class, but these members are defined as private in the derived class.

ii) *protected*: the derived class inherits all public and protected members of the parent class, but these members are defined as protected in the derived class.

iii) *public*: the derived class inherits all public and protected members of the parent class, and their access type is not changed. That is, public members of the parent class remain public in the derived class, and protected members of the parent class remain protected in the derived class.

Consider as an example a `Polygon` class to represent and manipulate a polygon. Since there are characteristics common to all polygons, we can implement a parent class representing those characteristics. For instance, all polygons can be defined through a vector containing the length of each of their

sides; thus, this vector can be the only attribute of the `Polygon` class. The calculation of a polygon's perimeter corresponds to the sum of all its sides, which is also independent of the specific polygon. The same does not apply to the area, since its calculation depends on the type of polygon.

The parent class `Polygon` could then be implemented as shown below.

Header File

```
#ifndef POLYGON_H
#define POLYGON_H
//Include necessary headers

class Polygon{

protected:
    //attributes
    vector<double> Sides;

public:
    //Constructor
    Polygon(const vector<double>& );

    //Methods
    double Perimeter() const;
    double Area() const;
};

#endif // POLYGON_H
```

Source File

```
#include "polygon.h"

Polygon::Polygon(const vector<double>& v){
   Sides = v;
}

double Polygon::Perimeter() const{
    double p = 0;
    for(int i = 0; i<Sides.size(); ++i){
        p += Sides[i];
    }
    return p;
}

double Polygon::Area() const{
    throw runtime_error("ERROR!");
}
```

Suppose we want to create two new classes to represent triangles and squares. Both the triangle and the square are polygons and, therefore, share the characteristics of the `Polygon` class. These characteristics can be *inherited* from the parent class `Polygon` instead of being redefined in the `Triangle` and `Square` classes, thus avoiding code repetition. In addition to the members inherited from the parent class, the derived classes can also include other specific members.

Consider now the declaration and definition of these two classes. To simplify, we present only the header file of the classes, where we include the definitions (which should normally be placed in the source file). Note that the calculation of the area is well-defined for any square or triangle, so the respective classes include a method to perform this calculation.

Header File

```
#ifndef SQUARE_H
#define SQUARE_H
//Include necessary headers

class Square: public Polygon{

public:
    //Constructor
    Square(double x): Polygon(vector<double>(4,x)){ }

    //Methods
    double Area() const{ return Sides[0]*Sides[0]; }
};

#endif // SQUARE_H
```

Header File

```
#ifndef TRIANGLE_H
#define TRIANGLE_H
//Include necessary headers

class Triangle: public Polygon{

private:
    double Base;
    double Height;

public:
    //Constructor
    Triangle(double a, double b, double c): Polygon({a,b,c}){
        Base = a;
        double s = (a+b+c)/2; //Heron's formula
        Height = 2*sqrt(s*(s-a)*(s-b)*(s-c))/Base;
    }

    //Methods
    double Area() const{ return Base*Height/2; }
};

#endif // TRIANGLE_H
```

Both classes publicly inherit from the `Polygon` class and each defines a specific method to calculate the area. The `Square` class presented does not include any additional attributes. In this class, there is a constructor that takes the length of the square's side as an argument. The constructor of a derived

class is always defined through the constructor of the parent class. In this case, since the parent class constructor receives a vector as an argument, it is necessary to create a vector with four elements, each of them having the value equal to the side of the square (`x`). The `Triangle` class has two specific attributes in addition to the general ones inherited from the `Polygon` class. Thus, any object of type `Triangle` will have three attributes: the base and height defined in the `Triangle` class, and the vector containing the side lengths inherited from the `Polygon` class. The constructor of the `Triangle` class receives the three side lengths of the triangle and uses them to initialize its specific attributes (base and height) as well as the vector *Sides* through the constructor of the `Polygon` class.

Although the *Perimeter* method is not explicitly declared in these two classes, it is inherited from the `Polygon` class and can therefore be used by objects of type `Square` and `Triangle`, as illustrated in the example below.

```
//Include necessary headers

int main(){
    Polygon P( {3, 1, 3, 5, 7} );
    cout << "PerimeterP: " << P.Perimeter();

    Square Q(3);
    cout << "\nPerimeterQ: " << Q.Perimeter();
    cout << "\nAreaQ: " << Q.Area();

    Triangle T(3, 4, 5);
    cout << "\nPerimeterT: " << T.Perimeter();
    cout << "\nAreaT: " << T.Area();

    return 0;
}
```

leading to the expected output

```
PerimeterP: 19
PerimeterQ: 12
AreaQ: 9
PerimeterT: 12
AreaT: 6
```

Let us now consider the program below. In this program, a global function f is created, which takes as an argument a constant reference to a `Polygon`. Since `Square` and `Triangle` are classes derived from the `Polygon` class, they are also of type `Polygon` and can therefore be passed as arguments to the function f. The call to the *Perimeter* method in function f does not raise any problem when the function receives a `Polygon`, a `Square`, or a `Triangle` because it is defined in the `Polygon` class for any polygon. However, the same does not happen with the method *Area*. Although the `Square` and `Triangle` classes have their own *Area* methods, when an object of one of these types is passed to the function f, the *Area* method that is called will always be the one from the `Polygon` class, which, in this case, throws an exception.

```
//Include necessary headers

void f(const Polygon& P){
     cout << "\nPerimeter: " << P.Perimeter();
     cout << "\nArea: " << P.Area();
}

int main(){
    Square Q(3);
    f(Q);
    Triangle T(3, 4, 5);
    f(T);
}
```

This is not what is desirable here, because the *Area* method called for objects of type `Square` and `Triangle` should be the one defined in their respective classes. To achieve this, the *Area* method in the `Polygon` class must be declared as a *virtual* or *pure virtual* method. That is, like a method that will be redefined in the derived classes. When a virtual method is called for an object of a derived class, the method of the derived class (if it exists) is used instead of that of the base class. The virtual or pure virtual *Area* method can be declared and defined in the header file of the `Polygon` class, as shown below, with no changes needed in the derived classes.

```
// Declaration as a virtual method
    virtual double Area() const;

// Declaration as a pure virtual method
    virtual double Area() const = 0;
```

The main difference between virtual and pure virtual methods is the possibility of creating objects of the class in which they are defined. If the *Area* method is declared as a pure virtual method, it is no longer possible to create objects of the parent class `Polygon`; only objects of the derived classes can be created. That is, if the *Area* function is pure virtual, we have:

```
Polygon P({1, 4, 7, 3})  //ERROR!
Square Q(3)              //OK
Triangle T(3, 4, 5)      //OK
```

The same does not apply to a virtual method. In this case, creating objects of type `Polygon` is also possible. Furthermore, a pure virtual method must be implemented in all derived classes, whereas a virtual method does not have this requirement.

Defining the *Area* as a virtual or pure virtual method allows the previous function f to work correctly; that is, the `Polygon` received as an argument is treated as a `Square` when the function is called with a `Square`, and as a `Triangle` when the function is called with a `Triangle`. This ability of an object to behave as if it were of another type is called *polymorphism*.

Chapter 10

Writing and Reading Files

The use of files is essential for importing and exporting large amounts of data in a program. The `fstream` package from the C++ *standard* library — which stands for *file stream* — contains methods for file manipulation, and must therefore be included in the preamble using the instruction:

```
#include <fstream>
```

This package contains the classes `ofstream` and `ifstream`, which allow, respectively, writing to and reading from files. Their meanings are *output file stream* and *input file stream*, respectively.

10.1 Writing to Files

To write to a file, it is necessary to create an object of the `ofstream` class, which creates a channel to send information to a file. To do so, we should use the instruction:

```
ofstream name(Path, open_mode);
```

or the instructions:

```
ofstream name;
name.open(Path, open_mode);
```

where `name` is the name of the `ofstream` object to be associated with the file, `Path` is the location of the file to be written to (including its name), and `open_mode` is the option that indicates what to do with the file's existing content (if any). For `open_mode`, there are two options: `ios_base::out` and `ios_base::app`. Both options create the file if it does not already exist; however, if the file does exist, the first option erases its contents, while the second preserves the content and appends the new information at the end of the file. If the open mode is not specified, the default is `ios_base::out`.

The `Path` is a *string* that indicates the full path[1] to the file, including its name. When only the file name is provided in the `Path`, the *default* location is assumed, which is the project's *build* folder.

After creating the *ofstream* object `name`, we should check whether the file was successfully opened. To do this, we can use the *is_open* method (`name.is_open()`) or the logical not operator (`!name`).

[1]To access a file's location in Windows, go to the folder containing the file, right-click the file, and select *Properties*. Then simply copy the full path shown there, replacing each backslash ("\") with a double backslash ("\\"). On Mac, the forward slash ("/") is used as a separator instead of double backslashes.

Once we have confirmed that the file is open, we can write to it using the `name` variable. The writing process is similar to writing to the screen, but instead of using the *cout* instruction, we use the `name` variable. Consider the example below.

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

int main(){
    ofstream file("C:\\Users\\A\\New.txt", ios_base::out);

    if(!file)   //or  if( !file.is_open() )
         throw runtime_error("ERROR: The file could not be opened.");

    file << "Text";

    file.close();
    return 0;
}
```

In this example, an `ofstream` object named `file` is created to send information to the file named *New.txt*, located at *'C:\Users\A"*. The opening mode is `ios_base::out`, so any content that exists in the file — if it exists — will be erased. Then, we check whether the file was successfully opened, and if not, an exception is thrown. If the exception is not thrown, which means the file was successfully opened, the word *"Text"* is written to the file. Finally, the writing channel to the file is closed using the *close()* method.

10.2 Reading from Files

To read a file, an object of the `ifstream` class must be created, which establishes a communication channel to read from a file. To do this, we must use the instruction:

```
ifstream name(Path)
```

where, as before, `name` is the name of the `ifstream` object and `Path` is the location of the file to be read (including its name). After creating the *ifstream* object `name`, we should also check if the file was successfully opened, just as we do when writing to files. Reading the file's content is similar to reading input from the screen, but instead of using the *cin* instruction, we use the variable `name`. Consider the example below.

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

int main(){
    ifstream file;
    file.open("Data.txt");
    //or simply: ifstream file("Data.txt");

    if(!file)   //or  if( !file.is_open() )
         throw runtime_error("ERROR: The file could not be opened.");

    string s;
    file >> s;

    file.close();
    return 0;
}
```

In this example, an `ifstream` object named `file` is created to read data from the file named *Data.txt*, located in the project's *build* folder, since no specific path was provided. Next, we check whether the file was successfully opened and, if so, the first character sequence in the file is read and stored in the variable `s`. Finally, the input channel used to read the file is closed using the *close()* method.

Note that only the first character sequence in the file is read, because reading a *string* stops when a space or a newline character is encountered. For example, if the contents of the file *Data.txt* are:

```
Data.txt
Tomorrow it will rain a lot.
Today it is not raining.
```

only the word *Tomorrow* is read and stored in the variable `s`. To read a complete line, that is, to read until a newline is found, we can use the *getline* function, which takes either two or three arguments. In the case where it takes three arguments, its syntax is as follows:

```cpp
getline(InputStream, storeInformation, delimiter)
```

where `InputStream` is an object of type `ifstream` (e.g., *cin*) used to indicate the source from which the information will be read (console, file, etc.). The second argument is the *string* variable (named `storeInformation`) where we want to store the read data. The third argument is the delimiter character of type *char*, which indicates up to where the *string* should be read. If the specified character does not exist in the line to be read, the reading continues to the following lines until the delimiter character is found or, in the case of files, until the end of the file is reached. In the two-argument version of the *getline* function, the last argument (delimiter character) is omitted, and it is assumed by default to be a newline character. Consider the example below:

```
int main(){
    ifstream file("Data.txt");
    if(!file)
        throw runtime_error("ERROR: File could not be opened.");

    string s1;
    getline(file, s1, 'i');

    string s2;
    getline(file, s2);

    file.close();

    string s3, s4;
    getline(cin, s3);
    getline(cin, s4, '/');
    return 0;
}
```

The content of the *string* `s1` is read from the file *Data.txt*. Thus, the first *getline* will read the first line of the file until it encounters the character `'i'`, meaning it reads *"Tomorrow "*, which becomes the content of `s1`. The second *getline* will also read from the file *Data.txt*. The reading of the second *getline* starts where the previous read ended, that is, right after the first character `'i'` in the first line. Since in this case no delimiter character is specified, the reading will stop at the end of the first line, resulting in `s2 =` *"t will rain a lot."*.

In the last two *getline* calls, *cin* is used, so the content of the *strings* `s3` and `s4` will be read from the screen. The reading of the *string* `s3` ends when a newline character is entered. Then, the reading of the *string* `s4` begins, and it will store all information entered before the character `'/'`.

To read all the information from a file, we can use a *while* loop similar to the one we use to repeatedly request values from the user. The statement `while(getline(file, s))` can be interpreted as "while there are lines to read in the file, store them in s".

```
//read lines successively

int main(){
    ifstream file("Data.txt");

    string s;
    while( getline(file, s) ){
        //...
    }

    file.close();
    return 0;
}
```

```
//read words successively

int main(){
    ifstream file("Data.txt");

    string s;
    while( file >> s ){
        //...
    }

    file.close();
    return 0;
}
```

10.3 Instructions `clear()` and `ignore()`

The `clear()` and `ignore()` instructions are used to manipulate input streams and are very important for ensuring the proper functioning of a program. The *cin* stream is an input channel from the screen, from which information is extracted, and it has two possible states: *"with error"* and *"without error"*. One reason that can cause the *cin* stream to enter an error state is reading a data type that differs from the type of the variable where the input is to be stored. Once in an error state, the input stream will not return to the *"without error"* state unless the program is restarted or the stream is "cleared", which would otherwise make it impossible to use the stream during the rest of the program execution.

The instruction `cin.clear()` serves to "clear" the stream, restoring it to the *without error* state and thus allowing it to continue being used in the program. Consider the example below.

```
int n1;
int n2;
string s;

cout << "First number: ";
cin >> n1;



cout << "Second number: ";
cin >> n2;

cout << "Text: ";
cin >> s;

cout << n1 << " "<< n2 << " " << s;
```

```
int n1;
int n2;
string s;

cout << "First number: ";
cin >> n1;
cin.clear();
cin.ignore(10000, '\n');

cout << "Second number: ";
cin >> n2;

cout << "Text: ";
cin >> s;

cout << n1 << " " << n2 << " " << s;
```

First, consider the code on the left. Suppose that, when reading the variable `n1` (numeric), the user enters a non-numeric character. In this case, the variable `n1` will hold a *garbage value*, and the input stream will enter in an *error state*, which prevents the reading of the second and third variables (which will also receive *garbage values*).

In the code on the right, the instructions `cin.clear()` and `cin.ignore(10000, '\n')` were added to the program. We start by analyzing what happens if we only have the instruction `cin.clear()`. If a non-numeric value, such as `'g'`, is entered when reading `n1`, the *cin* stream enters an error state. Upon reaching the instruction `cin.clear()`, the program changes the state of the *cin* input stream to *no error*, allowing information to be read from the stream again. However, the content still present in the stream at this point is the character entered during the failed read of `n1` and a newline character automatically added when the *enter* key was pressed after entering `g` in the console — that is, "g\n". Since this information still exists in the stream and has not been read into any variable, the user will not have the opportunity to enter new values. This means the program will attempt to associate the content already present in the stream with the variable `n2`, causing the same issue again. Note that if the reading of the variable `s` (of type *string*) were performed before reading `n2`, there would be no problem with the input stream, since the existing content would be assigned to variable `s`, that is, we would have `s=''g''`, and the user would then be able to enter a new value for `n2`.

We can thus conclude that merely changing the state of the input stream to *no error* may not be sufficient to solve reading issues, since the information present in the stream is not removed until it is stored in a variable. To erase all content currently in the input stream, we should use the instruction `cin.ignore(10000, '\n')`. This instruction aims to delete all characters (up to a maximum of 10000) that were entered before a newline character (caused by pressing the *enter* key) is encountered.

In the code on the right, when the user incorrectly enters a non-numeric character while reading the variable `n1`, the *cin* stream enters in an *error state*, which is then reset to *no error* using the `cin.clear()` instruction. Next, the instruction `cin.ignore(10000, '\n')` clears all content from the input stream. As a result, since the input stream is no longer in an error state and contains no leftover data, a new value is requested from the user for the variable `n2`, and then for the variable `s`, assuming there was no error when reading `n2`.

Another situation where the use of `cin.ignore()` may be necessary is in programs where the instructions `cin >>` and `getline(cin, ...)` are used together. Consider the following examples:

```
int n1;
string s;

cout << "Name: ";
getline(cin, s);

cout << "Number: ";
cin >> n1;

cout << n1 << " " << s;
```

```
int n1;
string s;

cout << "Number: ";
cin >> n1;

cin.clear();
cin.ignore(10000, '\n');

cout << "Name: ";
getline(cin, s);

cout << n1 << " " << s;
```

As we already know, the reading of `getline` ends when a newline character is found, which, in the case of this example, is implicitly inserted when the *enter* key is pressed, and this newline character is also read but ignored. Therefore, in the code on the left, after reading the string `s`, the *cin* stream will be error-free and empty, allowing the user to enter new information into the stream during the reading of the variable `n1`. This means that in this case, the instructions `cin.clear()` and `cin.ignore()` are not necessary.

In the code on the right, the variable `n1` is read first, and during this reading, the newline character `"\n"` automatically included by pressing the *enter* key is stored in the stream. When a `getline` follows immediately after, the existing information in the stream (`"\n"`) is read by `getline`, resulting in `s="\n"`. The use of the instruction `cin.ignore(10000, '\n')` allows the deletion of all content from the stream, including the newline character. Therefore, the user will have the opportunity to re-enter information that will be stored in the variable `s`. It is important to note that in this example `cin.clear()` is only necessary to handle the case in which the user inputs a non-numeric value during the reading of `n1`, and thus has no effect if that does not occur.

Finally, it is important to emphasize that the instructions `.clear()` and `.ignore()` can be used in exactly the same way for input streams other than *cin*. Such streams include, for example, file input streams or *strings* streams (which will be introduced in the following section).

10.4 String Streams

A *stream* is a "channel" through which information can be inserted or extracted. As we saw in the previous sections, an object of type *ofstream* is a channel for inserting information into a file, whereas an object of type *ifstream* is a channel for extracting information from a file. There are also channels for inserting and extracting information from the screen, namely *cout* and *cin*, respectively. In this section, we will look at how to insert and extract information from *strings*.

To use a *string* as an input or output stream, we must use objects of type `istringstream` and `ostringstream`, respectively. Both are available in the `sstream` package (*string stream*) from the *standard* library. Thus, it is necessary to include this package in the preamble using the instruction

```
#include <sstream>
```

To create an input stream for a *string*, we use an object of type `istringstream`. After that, it becomes possible to use the `>>` operator to extract information from the *string*. Consider the example below.

```
string s = "I have 30 chocolates";
istringstream iss(s);

string s1, s2, s3;
int n;

iss >> s1 >> s2 >> n >> s3;
```

In this example, an object of type `istringstream` named `iss` is created, which serves as an input stream for the *string* `s`. Thus, it becomes possible to extract each element from the *string* `s` and store it in a variable of the appropriate type, as illustrated by the variable `n` of type *int*. That variable will store the integer value 30. Note that reading from a *string* without using *getline* ends as soon as a space or newline character is encountered.

An object of type `ostringstream` creates an output stream for a *string* to which we can easily add information using the `<<` operator. The great advantage of using `ostringstream` objects is that it becomes possible to easily concatenate objects of different types into a single *string*. Recall that, until now, to concatenate a numeric variable into a *string*, it was necessary to use the `to_string` function. After building the information stream for the desired *string* using the `ostringstream` object, it is necessary to extract the created *string*, which is done using the `.str()` method. Consider the example below.

```
string s = "Mary weights ";
double x = 60.25;

ostringstream oss;
oss << s << x << "kg and measure " << 1.7 << 'm';

string new = oss.str();
cout << new;
```

In this example, an object of type `ostringstream` named `oss` is created, which will be used to concatenate information. This information results from different data types, namely a *string* variable, numeric variables, *strings*, numbers and characters. The content of the created *string* is then returned and stored in a new variable of type *string* (called `new`) using the `str()` method, and that variable is printed to the screen.

Bibliography

[1] Stroustrup, B. (2014). *Programming: principles and practice using C++.* Pearson Education.
[2] Stroustrup, B. (2018). *A Tour of C++.* Addison-Wesley Professional.
[3] `https://www.learncpp.com/`