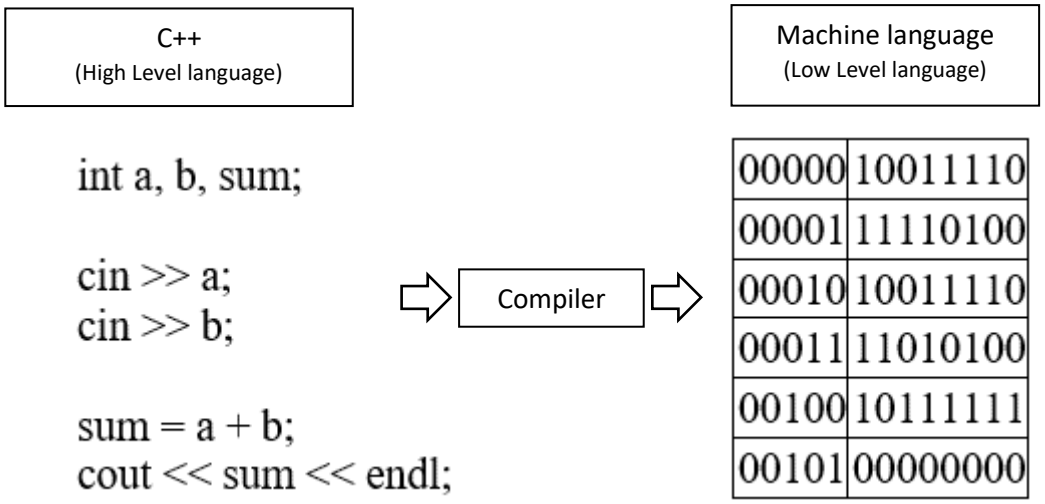
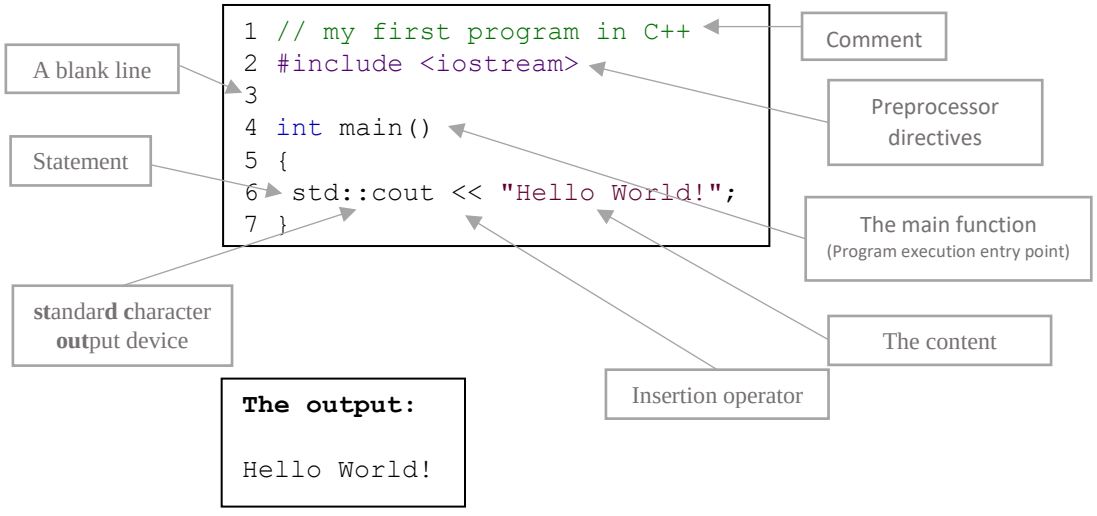
Introduction

Computers understand only one language and that language consists of sets of instructions made of ones and zeros. This computer language is appropriately called machine language.



Structure of a program



Remarks:

- comment has no effect on the behavior of the program and their purpose is to add explanations to the code they can written either as a *line comment* (`// comment`) or a *block comment* (`/* comment */`)
- Lines beginning with a hash sign (`#`) are directives read and interpreted by what is known as the *preprocessor*. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive `#include <iostream>`, instructs the preprocessor to include a section of standard C++ code, known as *header iostream*, that allows to perform standard input and output operations, such as writing the output of this program (`Hello World`) to the screen.
- Blank lines have no effect on a program. They simply improve readability of the code alongside with the indentation.
- Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a type (int), a name (main) and a pair of parentheses (()), optionally including parameters.
- All functions use braces to indicate the beginning and end of their definitions. Everything between these braces is the function's body that defines what happens when the function is called.
- A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body. The statement must ends with a semicolon (`;`).

`cout` is part of the standard library, and all the elements in the standard C++ library are declared within what is called a *namespace*: the namespace `std`. In order to refer to the elements in the `std` namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing `cout` with `std::`), or introduce visibility of its components.

```
// my second program in C++
#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
}
```

The output:

```
Hello World! I'm a C++ program
```

At which circumstance we use prefixing `cout` with `std::`over declaration and vice versa?

Programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work, we need to introduce the concept of *variables*.

Let's imagine that I ask you to remember the number 5, and then I ask you to also memorize the number 2 at the same time. You have just stored two different values in your memory (5 and 2). Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Then we could, for example, subtract these values and obtain 4 as result.

```
// operating with variables
#include <iostream>
using namespace std;
int main ()
{
  // declaring variables:
  int a, b;
  int result;
  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;
  // print out the result:
  cout << result;
  // terminate the program:
  return 0;
}
```

We can now define *variable* as a portion of memory to store a value. Each variable needs a name that identifies it and distinguishes it from the others. For example, in the previous code the variable names were `a`, `b`, and `result`, but we could have called the variables any names we could have come up with, as long as they were valid C++ identifiers.

A *valid identifier* is:

- A sequence of one or more letters, digits, or underscore characters (_).
- Spaces, punctuation marks, and symbols cannot be part of an identifier.
- Identifiers shall always begin with a letter. They can also begin with an underline character (_), but such identifiers are -on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case can they begin with a digit.
- The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the RESULT variable is not the same as the result variable or the Result variable. These are three different identifiers identifying three different variables.

C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords. Sample of the standard reserved keywords that cannot be used for programmer created identifiers are: `bool, char, if, int.`
Specific compilers may also have additional specific reserved keywords.

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name. What the program needs to be aware of is the kind of data stored in the variable. It's not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they don't occupy the same amount of memory.

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

- **Character types:** They can represent a single character, such as `'A'` or `'$'`. The most basic type is `char`, which is a one-byte character. Other types are also provided for wider characters.
- **Numerical integer types:** They can store a whole number value, such as `7` or `1024`. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values or not.
- **Floating-point types:** They can represent real values, such as `3.14` or `0.01`, with different levels of precision, depending on which of the three floating-point types is used.
- **Boolean type:** The boolean type, known in C++ as `bool`, can only represent one of two states, `true` or `false`.

The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: `void`, which identifies the lack of type; and the type `nullptr`, which is a special type of pointer. Except Boolean, Void, and Null pointer, each group has other sub category which has the same properties but with different sizes. For example **`char`** has an 8 bits memory size while **`char16_t`** has 16 bits memory size. These sizes are not standard and depends on the used compiler and the machine architecture. And to see the size of the variable we use the function `sizeof (data type)`. The number of unique values a data type can represent is calculated by 2^{bits} for example datatype with the size of 8-bit can store $2^8 = 256$ unique values.

Why there are different size of the data types?

For integer types, having more representable values means that the range of values they can represent is greater; for example, a 16-bit unsigned integer would be able to represent 65536 distinct values in the range 0 to 65535, while its signed counterpart would be able to represent, on most cases, values between -32768 and 32767. Note that the range of positive values is approximately halved in signed types compared to unsigned types, due to the fact that one of the 16 bits is used for the sign; this is a relatively modest difference in range, and seldom justifies the use of unsigned types based purely on the range of positive values they can represent. For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.

C++ requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // c-like initialization
  int b(3);              // constructor initialization
  int c{2};              // uniform initialization
  int result;            // initial value undetermined
  a = a + b;
  result = a - c;
  cout << result;
  return 0;
}
```

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks. An example of compound type is the `string` class. Variables of this type are able to store sequences of characters, such as words or sentences. A very useful feature! A first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header `<string>`):

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

As you can see in the previous example, strings can be initialized with any valid string literal, just like numerical type variables can be initialized to any valid numerical literal. Note: inserting the `endl` manipulator **end**s the **l**ine (printing a newline character and flushing the stream).

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

Literals are the most obvious kind of constants. They are used to express particular values within the source code of a program. We have already used some in previous chapters to give specific values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

The `5` in this piece of code was a *literal constant*. In addition to decimal numbers (those that most of us use every day), C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16) as literal constants. For octal literals, the digits are preceded with a `0` (zero) character. And for hexadecimal, they are preceded by the characters `0x` (zero, x). Any of the letters that can be part of a floating-point numerical constant (`e`, `f`, `l`) can be written using either lower or uppercase letters with no difference in meaning.

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


int main ()
{
  // literals
    cout << 75;         // decimal
  //cout << 0113;       // octal
  //cout << 0x4b;       // hexadecimal

  //cout << 75;         // int
  //cout << 75u;        // unsigned int

  //cout << 3.14159;    // 3.14159
  //cout << 6.02e23;    // 6.02 x 10^23
  //cout << 1.6E-19;    // 1.6 x 10^-19
  //cout << 3.0;        // 3.0
  //cout << 3.14159L   // long double
  //cout << 6.02e23f   // float
}
```

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{

cout << '\n';
cout << '\t';
cout << "Left \t Right";
cout << "one\ntwo\nthree";
cout<< newline;
cout << "string expressed in \
two lines"
;
cout<< newline;
cout<< pi;
cout<< newline;
cout<< "pi";
}
```

Notice that to represent a single character, we enclose it between single quotes (`'`), and to express a string (which generally consists of more than one character), we enclose the characters between double quotes (`"`). Both single-character and string literals require quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

Here, `pi` alone would refer to an identifier, such as the name of a variable or a compound type, whereas `"pi"` (enclosed within single quotation marks) would refer to the string literal `"pi"`. Character and string literals can also represent special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (`\n`) or tab (`\t`). These special characters are all of them preceded by a backslash character (`\`).

Internally, computers represent characters as numerical codes: most typically, they use one extension of the ASCII character encoding system (see ASCII code for more info). Characters can also be represented in literals using its numerical code by writing a backslash character (`\`) followed by the code expressed as an octal (base-8) or hexadecimal (base-16) number. For an octal value, the backslash is followed directly by the digits; while for hexadecimal, an `x` character is inserted between the backslash and the hexadecimal digits themselves (for example: `\x20` or `\x4A`).

Some programmers also use a trick to include long string literals in multiple lines: In C++, a backslash (`\`) at the end of line is considered a *line-continuation* character that merges both that line and the next into a single line.

Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:

```
#define identifier replacement
```

After this directive, any occurrence of `identifier` in the code is interpreted as `replacement`, where replacement is any sequence of characters (until the end of the line). This replacement is performed by the preprocessor, and happens before the program is compiled, thus causing a sort of blind replacement: the validity of the types or syntax involved is not checked in any way.

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

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

}
```

Note that the `#define` lines are preprocessor directives, and as such are single-line instructions that -unlike C++ statements- do not require semicolons (;) at the end; the directive extends automatically until the end of the line. If a semicolon is included in the line, it is part of the replacement sequence and is also included in all replaced occurrences.

References:

http://www.cplusplus.com/doc/tutorial/introduction/