PRACTICAL GOAL: Writing your first C++ program.

Programing: An Introduction

Rafael Vicente

The Computer

There are many books that already provide an adequate introduction to basic computer architecture. The typical textbook offers a broad description of the typical hardware components and a view of their interaction. The computer is thus described as a machine capable of carrying out instructions with the support of its hardware.

The task of programming can be viewed as the art and technique of writing the instructions the computer must follow to execute a task. Of course, in order to for those instructions to be effective, they must be written in a language understood by the computer, and the "sentences" written in that language must make sense.

If we accept the view of the computer as a very obedient extraterrestrial being, programming would be akin to providing our E.T. with detailed directions as to how to accomplish a task in an language understandable to him. Of course, without a common language, our directions, however precise, would be useless. Therefore, the first step in making effective use of our alien collaborator is to design a language that both he and we can use to communicate. Once such a tool is in place, communication is possible. But that's no guarantee that this communication will be effective. A complete knowledge of the new language would not necessarily enable us to write instructions, for example, to build a flying saucer. In other words, you would have the means of expression, but, in order for your commands to be effective, you would have to know what to say.

Therefore, the task of programming requires both the ability to "speak" a shared language, and an exact knowledge of the task you are programming. That knowledge must be acquired first, and then translated from a human language into the dialect your computer understands.

The Programmer

According to the above description, we could consider programming as a special case of communication. Special because we must use a specialized language; special also because, for this type of communication to achieve its goals, our commands must be accurate enough to accomplish the task and must be expressed without ambiguity. This way, the task, and nothing but that task, will be guaranteed.

If we found a way to instruct our alien on the steps needed to cure a disease, we would certainly like other human beings to understand it and to be able to follow the same instructions. Ideally, our programs --initially a way to give directions to our computers --should also be understandable to our colleagues.

From this point of view, programming is more than merely preparing a computer to accomplish a given task; a program is a way to express how to accomplish the task in a way that is clear, precise, expressible in the terse language of the machine, but still understandable to humans. In other words, it's a way to express the solution to a problem.

Memory

We write programs as a set of instructions translated into programming language. Therefore, it should be evident that the language itself is one of the primary tools needed for programming.

Once our instructions are correctly translated into that language, the program is finished. If the instructions are also correct, we have a method to solve the problem, or carry out the task we had set out to accomplish. The program itself is a description of the solution we devised.

But the program, by itself, is of little use. On one hand, we have a solution to our problem; on the other hand, to be effective, this solution cannot be merely on paper: it must refer to, and act, on something physical: the computer.

Imagine we want to teach our E.T. how to climb a staircase. Our instructions -our program- must refer to our E.T.'s legs as the tools to be used, and must assume a staircase. These assumptions must be present when the program is being written, so that our instructions use those "tools" to accomplish the task of climbing the staircase. It should be clear that different assumptions on the shape of our E.T., might require a totally different set of instructions. In other words, our programs are tailored to an "architecture". If we envisioned our alien as a legless being, we would have to devise our "climbing" program to use crawling instead of walking.

Since our programs will use computers, we will have to assume a basic architecture. Just as our stair-climbing instructions would refer to the alien's legs, specifying how they should be moved, a computer program will need a part of the computer to act upon and change in order for the program to be executed. If nothing changes inside the computer, our program has no effect.

The part of the computer our programs will use, and change, is memory. Without getting into details, we can adopt a metaphorical view of memory as a set of cells our programs can access and change. The figure below would depict the memory configuration of a ridiculously small computer.

		'A'	
56			
	3.45678910		

The type of computer determines the "capacity" of each cell, in *bits*. You can imagine memory composed of rows of very tiny light bulbs. Each light bulb can be turned on or off, and a code can be associated with sequences of bulbs. For example, assuming each memory cell in our computer was made up of eight light bulbs, we could represent integers by adopting the following code: 00000001 (seven bulbs off, one on) Number 1

00000010 (six off, one on, one off) Number 2

00000011 (six off, two on) Number 3

where each digit, 0 or 1, represents a bulb that is off or on, respectively. Therefore, in our metaphorical view, a memory cell would have eight of these bulbs, some of them on and some off. Total memory would be the set of all these memory cells. A code used to translate from the bulb configuration to actual numbers would be adopted and agreed upon.

Although actual computer memory is not really made up of bulbs, it is made up of components that can be turned on and off in a similar way. Each one of these elements can be 0 (off) or 1 (on), and is called a bit.

This suggests that our somewhat primitive picture of memory is not quite accurate, because we would never find a 56 in a memory location, but rather the equivalent set of zeroes and ones ("bulbs" turned on and off) that, in the code, represents that number. In fact, the number stored in that memory location would indeed be 56, but written in an different encoding.

This memory configuration is what enables the computer to store information: the instructions we load into it as well as the data we want to manipulate, change and act upon.

We said above that for our programs to execute, we would need to assume an architecture, and write instructions that acted on it. Now we know that our computer programs will act on memory; this is the part of the computer's "organism" that we can act upon, change and manipulate as the program is running. Unlike our alien, we are not writing instructions to move a set of legs so they climb a staircase, we are writing instructions to change and operate on the contents of memory to achieve a result.

This discussion has shunned technical language on purpose. However, some terminology is necessary. We will call byte the number of bits needed to represent a character, like the letter 'A.'

If we refer back to our picture, we can see that our character occupies a memory cell, a byte, the number 56 also needs a byte, but that the long decimal number requires two bytes to be stored. This is the case with real computers; the amount of bits allocated to represent different types of data is different. Decimal numbers require more 'bulbs" that integers because their internal encoding is more complex. A real computer might need a byte for characters, two to four for integers and even more for decimals. In our toy example, that would mean that characters would be represented in 8 bits, but decimals in 16.

One last question that might be puzzling to the careful reader is this: if all the data in the computer is represented by bits (zeros and ones), how can the computer tell what types of data are stored in different memory locations?

In our example: we know 56 is stored as a series of zeroes and ones, and so is character 'A'. Given that both are represented internally as numbers, how can we tell which one is the character and which the numerical value? Do we run the risk of misinterpreting the contents of memory locations unless we make a note of what we stored in them?

This question should be answered in the coming chapters.

Algorithms

Now we know that program instructions must be written in a language intelligible to the computer, and that they use and act on the computer's memory.

Like a writer needs a language and a blank page, a programmer needs the programming language and memory. But this is not enough; the tools don't create the programs. Much as a writer does, a programmer needs to be clear about what it is to be written, must have a precise knowledge of the program's "subject."

A good command of the programming language is a necessary but not a sufficient condition to create a program. If a writer tells stories, a programmer writes methods to achieve results. These methods, called **algorithms**, must be unambiguous and must, of course, be finite; that is, the set of steps described by the algorithm must take us to the result and provide an ending point.

This means that, given a problem to solve or a result to achieve, in order to write the program the programmer must have:

* A method to solve it, expressed in any language. This set of unambiguous steps with an ending point is called an algorithm.

* A programming language to translate our algorithm into. The result of the translation constitutes the program

* A computer whose memory the program will use during execution.

More on Algorithms

Imagine we are asked to determine whether a number is evenly divisible by three. We could devise many different algorithms. The context of our algorithms will be the following: we assume we have pencil and paper, and a good command of arithmetic. Given these assumptions, the following algorithms are possible:

Algorithm 1

```
1. Write the number
2. Toss a coin.
3. If tails come up, say YES. The number is divisible by
   three. 4. If heads come up, say NO.
```

Algorithm 2

```
1. Write the number
2. Divide by 3.
3. If the remainder is zero, say YES.4. If the remainder
   is not zero, say NO.
```

Algorithm 3

```
1. Write the number
```

```
2. Add the digits.

3. Divide the result by three.

4. If the remainder is zero, say YES. The original was
   too.

5. If the remainder is not zero, say NO
```

Obviously, the first algorithm has everything an algorithm requires, but it does not solve the proposed problem. Formally, it provides a set of unambiguous steps with an ending point; but unambiguous steps and an ending point are not enough. Unless that set of steps leads us to the solution, our algorithm is useless.

The task of finding a correct algorithm is at the center of programming. It is the equivalent of a writer finding a good story to tell. If we look at the second algorithm, we will agree that the set of steps indeed allow us to determine divisibility by three. At least for a human, the procedure is possible and yields the desired results. But algorithms are to be translated into a programming language, where they must be adapted to the 'architecture" of the computer. We can't expect our computer to have a piece of paper to write the number on any more than we can expect a human to do long divisions in their minds.

Clearly, our task, now that we know algorithm 2 is correct, will be to translate it into our programming language of choice. Where the algorithm written in human language, uses "tools" available in the human world (pencil, paper, arithmetic skills), our programming language must refer to the tools available in the computer (memory, arithmetic operations). Our program will then use the capabilities of the computer to perform the steps of our algorithm.

As for algorithm 3, we can certainly say that it is also correct. It is the most efficient way <u>for a human to</u> test divisibility by 3, because it replaces the division of potentially long numbers with a sequence of additions, which are much easier operations for humans. On the other hand, algorithm three assumes easy access to each and every digit of the given number, so they can be added together.

In principle, we might assume that this substitution of division with addition would also benefit the computer. However, the operation that allows humans to isolate a number's digits is not as immediate or trivial to a computer. We can certainly program the computer to grab every digit of a number, but it is much more difficult to do than just issuing a simple division instruction. Therefore, due to the differences between the human "architecture" and the computer "architecture", method 2, while being the least efficient for humans, is the easiest to program.

This delicate balance between correctness, efficiency and ease of programming is what makes the design and writing of programs, and the choice of algorithms, such a challenging game.

Design

In order to program a task, we have seen, one must be able to break it down into a sequence of instructions. For complex tasks, the algorithm can be complicated and lengthy, potentially resulting in a tangle of instructions that are difficult for humans to follow.

Let's say your algorithm were described in a thousand lines. The mere task of reviewing it for correctness seems daunting. By line 500 it would be difficult to keep a clear view of what the preceding lines were doing, and how they interacted with each other, not to mention how they might interact with the instructions that follow.

The situation is similar to that of a child learning long division. If the division includes fairly large numbers, the piece of paper becomes too crowded, making it difficult to find a mistake and correct it.

The best way to overcome this pitfall is to design our algorithms, and therefore our programs, in a modular way. We can describe the solution to our problem by isolating subtasks that can be performed independently, and then describe the whole program as a combination of such tasks.

In the long division example, we could, for example, describe the division as a repetition of the following sequence:

1. Divide the divisor of x digits into number formed by the first x digits of the dividend.
2. Write the partial answer resulting from this initial division
3. Multiply that answer by the divisor
4. Subtract the result from the first x digits in the dividend
5. Bring down the next digit
6. Repeat the same PROCESS until all digits in the dividend have been used.

You can see that line 6 uses the word PROCESS. Clearly, our method to divide is repetitive, and we can reduce, and simplify, the description by referring to the repeated steps by one single word. Therefore. we could very well isolate steps 1-5 into its own module, and write a new algorithm, informally:

"Module" division

```
a.    Divide the divisor of  x digits intonumber
      formed by the first x digits of the dividend.
b.    Write the partial answer resulting fromthis
      initial division
c.    Multiply that answer by the divisor
d.    Subtract the result from the first x digitsin
      the dividend
e.    Return the partial answer and  remainder.
```

Our program then could be written as a main body that simply uses the module called division repeatedly until the task is finished.

Main Algorithm

```
1.    Send the divisor of x digits and the first x
      digits
of the number to be divided to module divide
2.    Append the digit of the partial answer
      calculated(returned) by module divide to the
      previous partail answer.     3.    If  we are out
      out of digits to bring down, go to step 5.
3.    Append the next digit of the original dividend
      tothe remainder returned by divide     4.    Go to
      step 1.
    5.    STOP. Partial answer and remainder collected so
far is the final answer.
```

As you can see, our main program assumes the division performed by module *divide* is correct. We could imagine the child in the example above deciding to collaborate with a classmate, each with their own sheet of paper, one performing always the same steps and passing the partial results to the other.

Similarly, in designing our programs, we can often isolate portions that are repetitive, and give a name to that sequence of instructions. Our program is then able to refer to those intructions by using their name instead of the instructions themselves.

The main advantage of this approach is the clarity it can afford, and the ease gained when it comes to correcting it. If we write our programs in this way, we can check the modules, much smaller, for correctness, and then assemble them into the final program.

Functions: main

One of the ways C++ facilitates modularity is by means of functions. A function is a part of the program that can be used by the rest by invoking or calling its name.

A function is defined by listing within {   } the C++ commands to be executed. Functions are essential in C++; in fact, a C++ program is starts as a function whose name is always the same: *main*.   In fact, a C++ program always starts by executing a function named main. A line like:

int main ( )

will be included in your program, and will signal to the compiler the point where execution will begin. In addition to function main, you can define other functions in your program , these can be called by main and even other functions.

Each C++ function must include a header (*int main ()*), and a body (the commands within { } that specify what the function does). The header must specify, in addition to its name, the values provided to the function, if any, and the type of value the function will return. In our example, main is a function which does not expect any values to be provided to it, no inputs sent to it, thus the empty (), and "promises" to return an integer at the end of its execution. If instead of the word *int* the word *void* had been used, the function would not be expected to return any results.

The idea is that functions are specialists, performing one simple task. The machine expects a type of input --the parameters in parenthesis--, and yields an output. The definition of a function must include a description of the inputs it expects in order to be able to perform its task, and of the type of output it will return when its task is completed.

A proper definition of a function requires a *header* and a *body*. The headerand contains the name of the function and the names of the inputs, if any, (in parentheses) and the return type, that is, the type of data the function will return. The body includes the function code within curly braces ({, }).

A function *prototype* is a description of the function (name, type of inputs and return type). Its mission is to alert the compiler of the characteristics of the function, so that, when the function is used in the program, the compiler can ascertain that its use is consistent with its name, inputs and type of output.

In our example, function *main* expects no inputs and returns an integer value as its output. That's the reason why you will see in the programming examples that function main ends often with the statement *return 0*, meaning that the result returned by *main* is always zero.

Your programs will have the following structure :

1) # include ....

2) prototypes of the functions you defined

3) int main ( )
  {
    //body of main, where you can call the other functions

  }   4)   definition of the functions

you use.

A Simple Example

However illuminating the theory might be, programming is also a technique that requires practice. In the previous section we talked about long division. In this section we'll use an even simpler problem to illustrate the structure of your typical C++ program.

We'll tackle a trivial case so we can concentrate in the form of our program. The program should compute the result of doubling an integer.

First, we'll design the program. We'll structure it so that function *main* uses a separate function that will perform the calculation and then will return the result to the main program. Of course, *main* will have to provide this other function with the value to be doubled. In this way, the function can be used as a, rather dumb, calculating machine specialized in doubling integers.

Therefore, the structure of our program will be:

* A main function (mandatory)
* Our function to double integers. We'll call it *twice*,for example.

Once we have this elementary design, it's time to figure out in more detail what method our function *twice* is to follow in order to compute its result. In order to describe that method we'll need to specify the information our function will be given. That is, the input to our function, and the results it must produce: its output. *Function twice*

Inputs: An integer, let's call it x.
Output: One integer, the result of doubling the input

Algorithm:

```
1.    Multiply the number x by 2
2.    Return the result.
```

Now that we have the function well thought out, we can start coding it. The result is shown in the following

Program

```
#include <iostream> //include modules necessary for input/output. Provided by C++
using namespace std;      // tell C++ to use the standard names for input/output
int   twice ( int  x);   // prototype of the function,  describes the inputs and outputs


int main()
 {

cout << “The result of calling twice 5 is: “;

cout<<twice (5); return 0;

}



int   twice ( int  x)
 {                                                   definition of the function
 return  2 * x;
}
```

The words in **bold** are part of C++, and they are in bold only for the purpose of this explanation.

The first two lines are needed to use C++'s input and output facilities. Any program needing to output to the screen or get input from the keyboard, will need to be preceded by those two lines. Their mission is to specify that the predefined input/output C++ modules be included in our program.

The keyword **int** indicates to the language translator (compiler) that you need to reserve in memory space for an integer, and associate that space with the name x. We say that x is a parameter of function *twice*.

The *int* at the beginning of the function prototype, and definition, indicate that *twice* returns a value of type *int* to the calling function.
Thus the keyword return in the statement *return 2 * x tells* the compiler that the result of 2 * x should be made available (be returned) to the calling function, *main* in this case.

The words "made available" mean that when *main* calls function twice, *as the* program is executing, the instruction(s) in *twice* will begin execution while main waits, As soon as *twice* is done, main will continue execution of the statement *cout <<twice (5).*

The result of *twice (5)* (which is returned to this point) is now displayed by main on the screen, which is what *cout* <<does.

The double quotes indicate a *string*, which is the way C++ represents sequences of characters treated as a unit. Therefore, cout << “Hello there”; would print *Hello there* to the screen.

You might be wondering what 5 does in parenthesis. That is in fact the way we send input to our functions. Of course, since twice was defined as expecting an integer value as input --it has an integer parameter-- the value we sent to it from *main* had to agree with the type expected by *twice.*