Computing with Integers

Data Types and Assignment

As we saw, our programs must store their data in computer memory. In a previous section we imagined memory as a set of lightbulbs which can be set either ON or OFF. A code, we said, could be based on combinations of groups of bulbs, let's say eight, so that a configuration like "00000101" would represent, for example, the integer 5.

Of course, the decision as to how many bulbs (**bits**) make up a group (**byte**) varies among different computer architectures, but the basic mechanism is the same. The idea is hardly new; the Morse code, based on transmission of long/short signals, was used many years before the appearance of the computer. Therefore, there's nothing new about the idea of representing numbers and letters using **binary** (on/off, long/short) methods to encode information.

When it comes to our programs, we will see there are several ways that data can be committed to memory. We will leave to the compiler the task of translating that data into the computer internal code. Therefore, if we want to save a number like 35, we will use C++ to issue the command without having to know how 35 will be represented internally.

Unfortunately, the internal representation used by the computer is not uniform for all types of data. This means that integers are represented using a different encoding from decimals, and also different from letters. There are many reasons for this; in the case of numerical values, decimals will require more *bits* (more lightbulbs, in our metaphor) because of their higher precision. It should be clear that 5.000001, although very close to 5, will need more space and, a different code, to be represented accurately. A similar arrangement must be provided to represent letters, which also require a different encoding.

What this means is that when we save data to memory, we will have to specify the type of data we are saving, so that C++ (the compiler) can reserve the number of *bits* needed, and apply the proper encoding. This way, if we want to save to memory the number 35, we will use a C++ instruction to reserve the amount of memory needed for an integer:

```
int myValue;
```

and we'll specify a name for that region of memory, so that we can store and retrieve values into and from it.

The line above **declares** a **variable** name "myValue" as an integer (**int**). By including this line in our program we are asking the compiler to reserve for our **variable** the number of **bits** needed to store and integer, and to apply the encoding for integers on any value we store in that variable.

When you execute your program, the appropriate (computer dependent) number of **bits** will be set aside for your **variable**, so that a line like

```
myValue = 35;
```

will store the value 35, encoded as a series of *bits*, in the area of memory associated by the compiler with the name "myValue."

The command used to store values in variables is called **assignment**. Your "=" sign does not have the usual mathematical meaning; it stands for a store operation by which the value on the right is encoded and placed into the memory location that appears on the left-hand side.

Expressions and Assignments

The idea of assignment is simple: the result of the expression on the right-hand side is stored into the memory location whose name appears on the left-hand side of "=".

Any expression that can produce a result, like operations or function calls, may be written on the right-hand side of "=" to signify that, after the expression is calculated, its result must be stored on the variable named on the left.

For example, the following lines:

```
int j;

j = 3 + 2;
```

will cause the memory location named "j" to be created. Then the operation 3 + 2 will be calculated and its result assigned to "j." The line:

```
j = j + 4
```

would calculate first the result of j + 4 and then assign it back to "j." Notice that, on the right hand side, the occurrence of "j" means "get the value stored in j" while on the left-hand side means the location called "j." The complete process would be:

1. Find the result of j + 4 by
    1. retrieving the value in memory location j. (5 in this case) 2. adding that value to 4 and getting the answer (9 in this case)
    2. Assign the result to the memory location "j", which now will contain 9.

You have seen examples of two expressions, both involving arithmetic, used in assignments.

The assignment operation is itself an expression which yields a result. Therefore, it should be possible to write it on the right-hand side, like:

```
int j , k; //define two variables j and k.

j = 3;

k = 7;

 j = k = 5;
```

This example does not accomplish much other than, we hope, puzzling you a little and inviting you to think. What is the final value contained in "j" after the above lines are executed?

Operations with Integers

We have seen that integers are one of the basic **data types** in C++. They are represented in a a predefined (depending on the architecture of the computer) number of bits, and they are encoded as binary numbers, in other words, in a way similar to our light bulb example.

As we have seen, data types like **int** or **char** (used to represent characters, like letters, digits, spaces, etc.), are the way the language uses to specify the number of bits (i.e. the amount of memory) to be used in order to store each kind of data.

Therefore, a C++ statement like

```
int a;

char b;
```

instructs C++ to create two variables a and b, and reserve the amount of memory necessary to store a as an integer, and, in a different location, reserve enough memory to represent a character (like 'a', 'Z', ' ', '=').

We can see that a data type uses the same amount of memory every time is declared. That is, variables of type int will be allocated the same number of bits internally. Of course, this implies that integers beyond a certain value cannot be accurately represented, because they need more bits than allocated by the default definition of int.

Values of the same data type share not only the number of bits, but also a more abstract characteristic: they may share a set of operations.

We are familiar with the mathematical fact that adding integers is always done in the same way, and that decimals also share a common addition method, different from the one for integers. The notion of type in programming languages has the same connotation. All int values share the same operations with the same meaning, which might be different from the operations shared by chars or decimals (called **float** and **double**). For example, a programming language might include an operation for decimals that calculates the decimal portion (2.3 would yield 0.3), but that same operation would not make sense --and would not be available-- for integers.

The integer operations defined by C++ are detailed in the textbooks. We will not list them here again. One aspect worth noting is the importance of the order of operations, called **precedence**. In general C++ will follow the order of operations in mathematics, but will also introduce the idea of **associativity**, a big word that essentially refers to the direction (left to right, or viceversa) that is used to compute a result.

The order of operations in basic arithmetic follows a left-to-right association. C++ uses left-to-right for some operations and right-to-left for others.

An example of right-to-left is the assignment operator that we saw in the previous page. Statements like `int a =5; int b=12; a = b = 7;` would have a different meaning if the = operator was not performed right-to-left. Variable b gets the value 7, and then variable a gets the contents of b, which results in a and b being 7.