Selection: Boolean, Conditions, if ...then...else

So far we have been using variables of only one type: integer. Now we will introduce a new type that is both more abstract and more restricted in its range of values.

A variable of type "bool" (boolean) is able to store only two values: true or false. In C, the values true and false were represented by integers: 0 was false, and any other integer would mean true. C++ added the values *true* and *false* to the language, and the type bool, thus allowing the following statement:

```
bool result = false;
```

where variable result is of type boolean and can be assigned values *true* or *false* the same way int variables can be assigned any integer value.

One might wonder why we need to manipulate in a programming language the notions of *true* and *false*. Comparison is the simple answer.

In previous Programming with Words assignments we realized that in order to provide our algorithm with the ability to choose, we included instructions like: "Are both feet at the same level?" , and we allowed to proceed to different parts of the programs depending on the Yes/No answer; that is, depending on the answer being *true* or *false.*

In a typical programing language we will perform comparisons using relational operators similar to the ones used in Math. For example: bool x = (3 > 2); would assign the value *true* to the variable x. If you printed x now (cout << x) your screen would display a 1, which is the way the value *true* is displayed , but not necessarily the way it is stored. This difference between internal and display representation is not new: we saw that integers are represented internally in a binary encoding, but when we print them they appear to us in their familiar code.

A sign of the C legacy is that many C++ implementations will allow you to assign an integer to a boolean variable (bool x = 2;). If this int is not zero, then it will be assumed to mean *true*. Therefore, when writing cout << x, the value printed will be the display representation of *true,* the number 1.

Relational and Logical Operators

We have seen that the Math relational symbols can be used in C++. In some cases the notation is not exactly the same (<=, >=, ==, !=, for less than or equal to, greater than or equal to, equal to and not equal to).

The important notion to keep in mind is that these relational symbols are indeed operators; this means that -- as standard +, - , *, / -- they perform an operation (a logical one in this case) and return a result (a boolean *true* or *false*).

The relational operators can be combined with the usual logical operators (&&, ||, !: and, or and not respectively) . The discussion as to how logical operators work is described very simply in any textbook. To us it is important to emphasize that use of any of these operations result in a boolean.

For example:

bool x = (3 > 2) && (4 <= 5);

The value of x in this case should be *true*, since both parts of the && (and) operator are relations that evaluate to *true.* Since the logical operator && (and) requires that both conditions are true for the whole expression to be true, we can see that the final value must be true.

Had we written x = (3 > 2) && (4 > 5). the value of x wold be set to *false,* since in this case one of the values combined by && is false, and the && is only true when both are true.

A similar situation occurs with the operator or (||), with the only difference that || is true if at least one of its operands is true. In this case, x would be set to *true* for both our examples.

The rules for these logical operators, and further examples, can be found in the textbooks. We will study how these relational/logical operators are used express conditions, and how in turn these conditions are used to select different paths of execution in our programs.

Branching

As we have seen, boolean expressions allow us to test a variety of conditions by using relational and logical operators.

Normally, a program executes the statements in the order they are written. This sequential execution model would not be enough to express many algorithms: for example, the instructions for our step-climbing alien would have been useless without our ability to include a boolean test ("Are both my feet at the same level?") as part of a selection mechanism to decide whether is time to stop or to continue climbing.

C++ provides a construct to express this type of selection based on a boolean expression (a condition). The statement *if (*condition*)* ..... *else*..... allows us to test for conditions and execute a portion of the program is they are true or a different portion (the *else* part *)* if they are false.

In the alien's example, we could have written something like:

if ("Are both feet at the same level?" is true)

stop climbing;

else

keep climbing.

Unfortunately, the reduced language we used for our alien did not include an *if....else....* statement, so we were forced to express the branching operation in a more primitive way.

An example of a C++ use of *if...else...* could be:

```
#include <iostream> //or iostream.h

int main (void)

{ int number;

cin >> number;

if (number > 0)

cout << "Number " << number << " is already positive\n";

else

{ number = number * -1;

cout << "Converted number is "<< number << " \n";

}

return 0;

}
```

This simple program gets an integer form the keyboard and tests whether is positive or negative. If it is positive, it just prints it, otherwise (*else*) it changes it to positive and prints it.

You might notice the braces ({ }) after the else. They are necessary because the *else* part must execute more than one statement; in cases like that the group of statements to be executed are in a block enclosed in braces. This is not needed in the first part of the *if...else...* because, in our example, if the condition is true only one statement is executed.

Those adventurous minds wondering what type of statements can be used inside the *if...else...* construct will be happy to learn that there are no restrictions. Any C++ statement or group of statements may be used, including the *if...else...* itself. This implies that we can test conditions within conditions, nesting them down to as many levels as we need.

In our example we could have written

```
if (number > 0)

if (number > 5)
```

```
cout << number << " \n";

else

cout << number << " between zero en 5\n";

else

cout << "Number is negative\n"
```

Notice how each *if* is paired with the closest *else*, which means that if number contained a 3, then " 3 between zero and 5" would be printed, and if number contained -3, then "Number is negative" would be printed.

Lastly, one might get the impression that every *if* must be paired to an *else*; but that os not so. An *if* may be all we need, without expressing any *else* alternative. In that case, if the condition fails to be true, then the statement(s) tied to the condition will not be executed, and the program will continue with the next statement following the *if*.

We see that the *if...else...* statement can be used to alter the path of execution depending on conditions we set up. Without this capability, our programs would execute only sequentially, which would severely limit our ability to express useful algorithms.