Repeating sequences of instructions: the *while* loop

In a past exercise, we gave our alien instructions to climb a staircase. Without knowing the number of steps in advance, the only way to write the "program" was to set an ending condition and to identify the set of instructions necessary to climb one step. By **repeating** those instructions until the condition was met, we could express, in a compact way, the algorithm that would allow our alien to climb the staircase.

The ability to express repetition is therefore essential. Without it we would have had to write explicitly the instructions for every single step to be climbed, followed by our "end-of-the-stairs" check. Unfortunately, this would be impossible, since, without knowledge of the number of steps, we wouldn't know how many times to include in our program the instructions to climb one step.

Looping constructs allow us to express this type of repetition. We can list the set of instructions to be repeated and indicate how many times, or on what condition, the repetition will take place. This means that we can put a stop to the repetition by either specifying the number of iterations (if we know it in advance) or by setting a condition, to be checked after each round through the loop, that will decide whether to continue or to stop looping.

The *while* loop allows to list the set of instructions to be repeated, and to control this repetitions by setting an initial condition that will be also checked every time we come to the end of the loop. If the condition is met, we'll continue looping; otherwise the loop will end and the program will go on to the next instruction after the *while* loop.

As a simple example, imagine you wanted to print to the terminal all numbers between one and twenty, twenty included. A possible *while* loop to do this would be:

```
int number = 1;

while ( number < = 20)

{
```

```
cout << number << " ";

number = number +1;

}

cout << "Finished" ;
```

The condition is written inside parenthesis after the *while* keyword, and the set of instructions to be repeated (called the **body** of the loop) follow in braces. Braces are only needed if the body of the loop contains, like in this case, more than one instruction.

Whe the above while loop is executed, the condition is checked first. If it is true, then the body of the loop is executed once after which the condition at the top of the loop is rechecked. If it is still true, we will enter again the body of the loop; otherwise, the loop will be skipped and execution will jump to the first instruction after the loop; in our example, the line that prints "Finished" to the screen.

As an example where we don't know in advance the number of repetitions, imagine we wanted to write a program that prints all integers between one and some number entered form the keyboard. Our program could be written like this:

```
#include <iostram.h>

int get_number(void);

void print_sequence(int );

int main (void)

{

int number;

number = get_number();

print_sequence(number);

return 0;

}




int get_number(void)

{ int n;

cout << "Enter an integer greater or equal to 1\n" ;
```

```
cin >> n;

return n;

}



void print_sequence (int x)

//output to screen all integers from 1 through x

{

int i = 1;

while ( i <= x)

{ //begin loop body

cout << i << " ";

i = i +1;

} // end of loop body

} // end of function
```

As you can see, the *while* loop checks the condition before the body of the loop is executed. This implies that, in cases where that condition fails, the body of the loop will not be executed at all.

What if you wanted the body to be executed once and have the condition checked at the end? Then a slightly different looping statement would be used: the *do-while* loop that we'll cover in the next page.

The do-while loop

There are cases where we want to execute the body of the loop at least once, irrespective of whether the condition fails initially or not.

The use of a regular *while* loop in those cases would be possible but awkward. It would require that we wrote twice the body of the loop: once before the *while*, to be executed before testing the condition, and the other as part of the *while*, to be executed if and while the condition is met.

Imagine we wanted to count the number of items entered from the keyboard and print a message after each item stating its order in the input sequence. We will stop doing this if the number entered is not positive, and that number will be the last (or only, if it is the first one) value listed. The while version follows:

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

int main (void)

{

int n, count = 0;

cin >> n ;
cout << "We just entered item number " << ++count<< endl;;

while (n > 0)

{
 cin >> n;
 cout << "We just entered item number " << ++count<<endl;
 }
```

return 0;

}

As you can see the body of the loop is repeated before the loop to guarantee that evenif the first input is not positive, its order (number) in the inout sequecne is displayed.

The same program using the do-while loop could have been written:

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

int main (void)

{

int n, count = 0;


do

{
 cin >> n;
 cout << "We just entered item number " << ++count<<endl;

}while (n>0);

return 0;

}
```

As you can see, the program looks simpler. The loop executes its body at least once, since the condition is now tested at the end; therefore we don't have to write the same instructions more than once.

The do-while can be interpreted as a way to express the idea of repetition until a condition is met, since we stop executing its body when the condition fails.

The break keyword

In some cases we may want to exit a lop before the loop condition becomes true. The **break** keyword can be used to achieve this.

For example, if we wanted to alow the user to enter at most 20 integers, but stop the process as soon as an even integer is entered, we could use **break** to exit the loop as soon as we see an even number (divisible by 2):

```
int n, counter = 0;


while (counter < 20)
 {
   cin >> n;

   if ( n %2 == 0) // if n is even
       break; //stop the loop
  else

      counter = counter + 1;
 }
```

Note that the use of **break** is not inevitable. You can often find an equivalent way to achieve the same effect without using **break**.

The continue keyword

If , while in the middle of a loop iteration, you want the program to skip the remaining statements and jump back to the next iteration, you can use **continue.**
In a do or while loop, the next iteration starts by reevaluating the controlling expression of the do or while statement.

In a for loop (using the syntax for(init-expr; cond-expr; loop-expr)), continue causes loop-expr to be executed. Then cond-expr is reevaluated and, depending on the result, the loop either terminates or another iteration occurs.

For example, if we wanted the user to keep typing integers, print only those divisible by 5, and stop only when three integers divisible by 5 have been entered , we could write:

```
 int n, counter = 0; //counter counts how many integers divisible by 5 have been
entered

while ( counter <  3 )
   {
     cin >> n ;

   if (n % 5 !=  0) //do nothing if the n is NOT divisible by 5
      continue; // go back to the condition, skipping the following lines


        cout << n << ' ';
        counter = counter + 1;


    }
```

The following for loop version would limit the number of integers typed to 3 and would print only those divisible by 5:

```
int n, counter = 0;

for  (int counter = 0;   counter <  3; counter++)
   {
     cin >> n ;

    if (n % 5 !=  0)
      continue; // jump bck to the thord part of the for-loop header


        cout << n << ' ';


    }
```