Lecture-9

JUMP STATEMENTS.

The break statement

Using `break` we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown
aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown
aborted!
```

The continue statement

The `continue` statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

```
}
```

The goto statement

`goto` allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.
The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (`:`).

Generally speaking, this instruction has no concrete uses in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using `goto`:

```
// goto loop example

#include <iostream>
using namespace std;

int main ()
{
  int n=10;           10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!\n";
  return 0;
}
```

The exit function

`exit` is a function defined in the `cstdlib` library.

The purpose of `exit` is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of `0` means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several `if` and `else if` instructions. Its form is the following:

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
  .
  .
  default:
     default group of statements
}
```

It works in the following way: switch evaluates `expression` and checks if it is equivalent to `constant1`, if it is, it executes `group of statements 1` until it finds the `break` statement. When it finds this `break` statement the program jumps to the end of the `switch` selective structure.

If expression was not equal to `constant1` it will be checked against `constant2`. If it is equal to this, it will execute `group of statements 2` until a break keyword is found, and then will jump to the end of the `switch` selective structure.

Finally, if the value of `expression` did not match any of the previously specified constants (you can include as many `case` labels as values you want to check), the program will execute the statements included after the `default:` label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

if-else equivalent

```
if (x == 1) {
  cout << "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
  cout << "value of x unknown";
  }
```

The `switch` statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put `break` statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the `switch` selective block or a `break` statement is reached.

For example, if we did not include a `break` statement after the first group for case one, the program will not automatically jump to the end of the `switch` selective block and it would continue executing the rest of statements until it reaches either a `break` instruction or the end of the `switch` selective block. This makes unnecessary to include braces `{ }` surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example `case n:` where `n` is a variable) or ranges (`case (1..3):`) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of `if` and `else if` statements

Program:9.1 Use of continue command

```
// Display even numbers 2,4,6,8,...

#include <iostream>
using namespace std;
void main()
{
      int num=0;
      while(num++<=100)
      {
            if (num%2!=0)
                  continue;//contine statement skips the step
                  cout<<'\t'<<num;
      }
}
/*2 4 6 8 10
12 14 11111116 18
........
*/
```