C++ CONTROL STRUCTURES

Slide credit to MELJUN CORTES
Revised by I.S SHEHU
Lecture Presentations: Object-Oriented Programming I (CPT 211)
(2014/2015)

Objectives

- At the end of the chapter the students should be able to:
  - Know what control structures mean in C++ and why it's important in writing programs
  - Describe the different control structures in C++
  - Create programs using the different conditional structures
  - Create programs using the different loop structures
  - Differentiate and use counters and accumulators
  - Differentiate and use the *break* and *continue* statements

What are control structures

- Just like in other object-oriented programming languages the idea of control structures has to do with ***the use of expression(s) to control the flow of execution in a program***
- The expression(s) enables us to control the behavior of our program like *what kind of function it will perform*, *when it will terminate* or *continue under certain circumstances or conditions*.

Why do we need to understand control structures?

- We need to understand control structures because it enables a programmer to write dynamic or sophisticated programs that;
  - Executes statement(s) depending on some condition (s).
  - Allows a statement or group of statements to be executed several times…
- All programs use control structures to implement the program logic.

Other words for *Control Structures* you may come across

- Program structures
- Program control statements
- Program flow statements
- Control flow statements

Relational Operators

- The expressions which determine
  - Selection or
  - Repetition

  } are usually comparisons
- Comparisons are done with relational operators

Operator	Description
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Beware of mistaking the assignment = for the equality ==

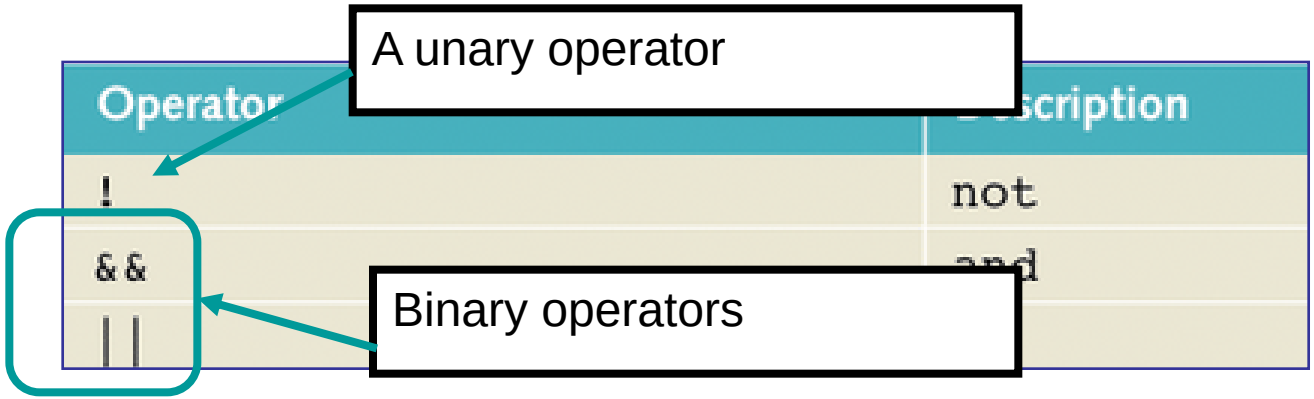
Relational Operators

Examples:

Expression	Meaning	Value
`8 < 15`	`8` is less than `15`	`true`
`6 != 6`	`6` is not equal to `6`	`false`
`2.5 > 5.8`	`2.5` is greater than `5.8`	`false`
`5.9 <= 7.5`	`5.9` is less than or equal to `7.5`	`true`

Logical (Boolean) Operators

- Logical or Boolean operators enable you to combine logical expressions



- Operands must be logical values
- The results are logical values (true or false)

Logical (Boolean) Operators

- The && operator (logical and)
  - If both operands are true, the result is true
  - If either or both operands is false, the comparison is false
- The || operator (logical or)
  - If either or both of the operands are true, the comparison is true
  - The comparison is false only if both operands are false
- The ! operator (logical not)
  - The not operator reverses the logical value of the one operand

3 Basic Control Structures

There are three types of Control Structures

1. Sequence
2. Selection/Conditional/Decision
3. Looping/ Repetition/ Iteration

1. Sequential Structure

- Default structure in programming
  - statements are executed one after another in the order of their appearance in the source code, i.e. execute first statement, then second statement, then third statement, etc
  - *Example:*

```
#include <iostream>
using namespace std;
    main()
      {
      int a = 5;                  //first statement
      int b = 6;                  //Second statement
      int sum;                    //Third statement

      sum = a+b;                  //fourth statement
      cout<<sum;                  //fifth statement
    }
```

2. Conditional Structure

- Selection structure in programming
  - execute statements depending on some condition
  - It is organized in such a way that there is always a condition or a comparison of two expressions that has to be evaluated first, which will decide the course of action of the program
  - In C++, the condition will either evaluate to a boolean value *true* or *false*, or integer values *1*(for true) or *0*(for false)
  - types
    - *if* statement
      - simple *if* statement
      - simple *if-else* statement
      - nested *if-else* statements
    - *switch-case* statement

THE SIMPLE *if* STATEMENT

- The syntax for the *if* statement is as follows:

```
if (<expression>)
        <statement>;
```

OR

```
if (<expression>)
{
        <statement1>;
        <statement2>;
        …
        <statementN>;
}
```

- The value of <*expression*> is evaluated first, if it results to a non-zero or true value, then <*statement*> is executed. If <*expression*> results to a zero or false value, then the program flow jumps to the next statement after the *if* structure.

THE SIMPLE *if* STATEMENT (continued…)

- There are some important things we must remember in using the *if* structure:
  - The *expression* must always be enclosed within a pair of parentheses; forgetting the parentheses will result into a syntax error.
  - If there is more than one statement that needs to be executed when the condition is non-zero or true, then these statements must be grouped in a pair of curly brackets.
  - Do not place a semi-colon (;) after the <*expression*> for this will cause a logical error.

Example #1

```
#include <iostream>

using namespace std;

main()
{
    // declare variable num
    int num;

    // input a value for num
    cout << "Input an integer value: ";
    cin >> num;

    // test if num value is positive
    if (num > 0)
            cout << num << " is POSITIVE\n";
}
```

Example #2

```
#include <iostream>

using namespace std;

main()
{
    int num;

    cout << "Input an integer value: ";
    cin >> num;

    // test if num value is even
    if (num % 2 == 0)
        cout << num << " is an even number.\n";
}
```

Example #3

```
#include <iostream>

using namespace std;

main()
{
      int month;

      cout << "Input month (1-12): ";
      cin >> month;

      if (month == 3 || month == 4 || month == 5)
         cout << "It is summer season.\n";
      if (month == 6 || month == 7 || month == 8 || month == 9 || month == 10)
         cout << "It is rainy season.\n";
      if (month == 11 || month == 12 || month == 1 || month == 2)
         cout << "It is cold season.\n";
}
```

THE SIMPLE *if-else* STATEMENT

- The syntax for the if-else statement is as follows:

  if (<expression>)

  <statement-1>;

  else

  <statement-2>;

- Like in the simple *if* statement, the value of <*expression*> is evaluated first, if it results to a non-zero or a true value, then <*statement-1*> is executed. Otherwise, if it is evaluated as zero or false, then the *else* part, i.e., <*statement-2*> is executed.

Example

```
#include <iostream>

using namespace std;

main()
{
    int num;

    cout << "Input an integer value: ";
    cin >> num;

    if (num >= 0)              // assume 0 is Positive
       cout << num << " is POSITIVE\n";
    else
       cout << num << " is NEGATIVE\n";
}
```

THE NESTED *if-else* STATEMENT

- Since *if-else* statements are statements by themselves, they can actually be used as statement(s) inside an *if-else* statement.
- We will refer to this construction as nested *if-else* statements.

Example #1

```
#include <iostream>

using namespace std;

main()
{
	int num;

	cout << "Input an integer value: ";
	cin >> num;

	if (num > 0)
	    cout << num << " is POSITIVE\n";
	else
	    if (num < 0)
	        cout << num << " is NEGATIVE\n";
	    else
	        cout << num << " is ZERO\n";
}
```

Example #2

```
#include <iostream>
using namespace std;
main()
{
	int scores;
	cout<<"enter scores: ";
	cin>>scores;

	if (scores>=70)
	cout<<"A";
	else
	if(scores>=60)
	cout<<"B";
	else
	if(scores>=50)
	cout<<"C";
	else
	if(scores>=45)
	cout<<"D";
	else
	if(scores>=40)
	cout<<"E";
	else
	cout<<"F";
}
```

CASCADED *if-else* STATEMENT

- If nesting is carried out to too deep a level and indenting is not consistent then deeply nested *if-else* statements can be confusing to read and interpret.
- Thus, a more consistent layout based on the syntax below is used, which we can also refer to as cascaded *if-else* statement:

```
if ( condition1 )
        statement1 ;
else if ( condition2 )
        statement2 ;
. . .
else if ( condition-n )
        statement-n ;
else
        statement-e ;
```

Example #2

```
#include <iostream>

using namespace std;

main()
{
    int num;

    cout << "Input an integer value: ";
    cin >> num;

    if (num > 0)
        cout << num << " is POSITIVE\n";
    else if (num < 0)
        cout << num << " is NEGATIVE\n";
    else
        cout << num << " is ZERO\n";
}
```

Example #3

```
#include <iostream>

using namespace std;

 main()
{
     int month;

     cout << "Input month (1-12): ";
     cin >> month;

     if (month == 3 || month == 4 || month == 5)
        cout << "It is summer season.\n";
     else if (month == 6 || month == 7 || month == 8 || month == 9 || month == 10)
        cout << "It is rainy season.\n";
     else if (month == 11 || month == 12 || month == 1 || month == 2)
        cout << "It is cold season.\n";
              else
              cout<<“wrong month entered.\n”;
}
```

THE *switch-case* STATEMENT

- The switch-case is a good alternative to cascading if-else statements. The syntax for the switch-case statement is as follows:

```
switch(<expression>)
{
        case <label-1> : <statement-1>;
                         [break;]
        case <label-2> : <statement-2>;
                         [break;]
        …
        case <label-n>: <statement-n>;
                         [break]
        [default : <statement-d>; ]
}
```

- The expression may be an *integer* or *character* variable or, as the name suggests, an expression that evaluates to an integer or a character value
  - the use of float and double data type values will result into an error.
  - The <*expression*> is evaluated and the value compared is with each of the *case labels*. The case labels must have the same type as the <*expression*> and they must all be different.
  - If a match is found between the selector and one of the case labels, say <*label-1*>, then the statement from <*statement-1*> will be executed. The same applies to other *cases*.
- The *[break]* statement is optional.
  - If it is present, it will cause the program to "break" or "jump" out of the switch-case, and to execute the next statement following switch-case.
  - If the break is not present, it will cause the program to execute the statement in the following case, i.e., <*statement- 2*> above, causing a **waterfall effect**, the same behavior applies to the other *cases*.
- If the value of the *expression* does not match with any of the case labels then the statement <*statement-d*> associated with *[default]* is executed.
  - The *[default]* is optional but it should only be left out if it is certain that the *expression* will always take the value of one of the *case labels*.
- Note that the statement associated with a case label can be a single statement or a sequence of statements (without being enclosed in curly brackets).

Example #1

```
#include <iostream>

using namespace std;

main()
{
      int month;
      cout << "Input month (1-12): ";
      cin >> month;
      switch(month)
      {
         case 1:  cout << "It is cold season.\n";
                  break;
         case 2:  cout << "It is cold season.\n";
                  break;
         case 3:  cout << "It is summer season.\n";
                  break;
         case 4:  cout << "It is summer season.\n";
                  break;
         case 5:  cout << "It is summer season.\n";
                  break;
         case 6:  cout << "It is rainy season.\n";
                  break;
         case 7:  cout << "It is rainy season.\n";
                  break;
         case 8:  cout << "It is rainy season.\n";
                  break;
         case 9:  cout << "It is rainy season.\n";
                  break;
         case 10: cout << "It is rainy season.\n";
                  break;
         case 11:  cout << "It is cold season.\n";
                  break;
         case 12:  cout << "It is cold season.\n";
                  break;
         default:
                  cout<<“"Input month out of
         range”;
         }
}
```

“Waterfall Effect” Example

```
#include <iostream>

using namespace std;

main()
{
      int month;
      cout << "Input month: ";
      cin >> month;
      switch(month)
      {
          case 1:
          case 2:
          case 11:
          case 12:
                    cout << "It is cold season.\n";
                    break;
          case 3:
          case 4:
          case 5:
                   cout << "It is summer season.\n";
                   break;
          case 6:
          case 7:
          case 8:
          case 9:
          case 10: cout << "It is rainy season.\n";
                         break;
         default:
              cout << "Input month out of range.\n";
     }
}
```

3. Looping Structure

- Repetition structure in programming
  - A loop is a control structure that allows a statement or a group of statements to be executed several times.
  - Components:
    - initialization of a variable or of several variables.
    - condition (that would evaluate to either true or false); the condition check is usually made on the current value of the variable initialized in (1) above
    - body (which maybe a single statement or a group of statements)
    - a change of state which is usually a statement inside the body of the loop that changes the contents of the variable(s)
  - There are three types of loop control structures in C++, namely:
    - *for* loop
    - *while* loop
    - *do-while* loop

THE *for* STATEMENT

- The *for* loop is the most compact looping structure.
- In this structure, all loop components are defined separately.
- The syntax of a *for* loop is as follows:

for ([initialization]; [condition]; [change of state])
<statement>;

- Based on the syntax presented above, the *for* loop is executed as follows:
  1. Perform the <*initialization*>
  2. Check the <*condition*>. If it is true, execute the <statement>. Otherwise, exit the *for* loop.
  3. Perform <*change the state*>, usually an increment or decrement operation. Go back to step (2).
- Note that <*statement*> can be a single or a block of statements.

  If what we have is the latter, we need to enclose the block of statements in a pair of curly brackets, i.e., {}.

Example #1

```
#include <iostream>

using namespace std;

main()
{
    int ctr;        // initialize counter

    //  start of the for loop structure
    for (ctr = 1; ctr <= 5; ctr++)
            // statement under for loop
           cout << "Jose Rizal University\n";
}
```

Example #2

```
#include <iostream>

using namespace std;

main()
{
    int ctr;

    for (ctr = 10; ctr > 0; ctr--)
        cout << ctr << endl;
}
```

THE *while* STATEMENT

- The syntax for the while loop is as follows:

```
        <initialization>;
while (<expression>)
        <statement>;
```

- Observe that in the *while* syntax, we did not include the <*change of state*>, simply because it is written as part of the <*statement*>.
- This implies that the <*change of state*> can be written at the beginning, in the middle or at the last line of <*statement*>.
- Note that <*statement*> can be a single or a block of statements. If what we have is the latter, we need to enclose the block of statements in a pair of curly brackets, i.e., {}.

Example #1

```
#include <iostream>

using namespace std;

 main()
{
    int ctr;

    ctr = 1;                                    // initialization
    while (ctr <= 5)                            // conditional check
    {
        cout << "Jose Rizal University\n";
        ctr++;                                  // change of state
    }
}
```

Example #2

```
#include <iostream>

using namespace std;


 main()
{
    int ctr;

    ctr = 2;                        // initialization
    while (ctr <= 20)               // conditional check
    {
        cout <<  ctr << "\n";
        ctr = ctr + 2;              // change of state
    }
}
```

Example #3

```
#include <iostream>

using namespace std;

void main(void)
{
    int ctr;
    int num;

    ctr = 1;
    while (ctr <= 10)
    {
        cout << "Input integer number " << ctr << ": ";
        cin >> num;
        cout << "Integer number " << ctr << "= " << num << "\n";
        ctr++;
    }
}
```

THE *do-while* STATEMENT

- The syntax for a *do-while* loop is as follows:

```
        <initialization>;
do
        <statement>;
        while (<expression>);
```

- <*statement*> is going to be executed as long as <*expression*> results into a condition that evaluates to *true* (or *1*). If it evaluates to *false* (or *0*), then the processing of the loop is terminated and the statement after the *while* loop is executed next.
- It is important to note that since <*expression*> is tested at the end of the loop, <*statement*> will be executed at least once.

Example #1

```
#include <iostream>

using namespace std;

 main()
{
    int ctr;

    ctr = 0;                            // initialization

    do                                  // body of the loop
    {
        cout << "Jose Rizal University\n";
        ctr++;                          // change of state
    } while (ctr < 5);                  // condition
}
```

Example #2

```
#include <iostream>

using namespace std;

main()
{
    int num=10;

    do
    {
        cout << num << "\n";
        num--;
    } while (num > 0);
}
```

Counters and Accumulators

- A counter is a variable that is used to keep track of the count (as the name suggest) of a certain group of items. Usually,
  - its data type is int
  - it is initialized to a value of 0
  - incremented by 1 inside a loop

Example (Counter)

```
 main()
{
    int ctr;                    // this is for the loop counter
    int num;                            // this is for the input value
    int ctr_positive;           // this is for the counter

    ctr_positive = 0;                   // initialization of counter

    for (ctr = 1; ctr <= 10; ctr++)
    {
        cout << "Input integer number " << ctr << ": ";
        cin >> num;

        // count the positive
        if (num > 0)
            ctr_positive = ctr_positive + 1;
    }

    cout << "Positive Numbers = " << ctr_positive << endl;
}
```

Counters and Accumulators (continued…)

- An accumulator is a variable that is used to keep track of the accumulated value of a certain group of items. An accumulator
  - may have a data type of *int*, *float* or *double*
  - it is usually initialized to a value of 0
  - changes by assuming the sum of the current value of the accumulator and the value of another variable

Example (Accumulator)

```
main()
{
    int ctr;                    // this is for the loop counter
    int num;                    // this is for the input value
    int sum;                    // this is for the accumulator

    sum = 0;        // initialization

    for (ctr = 1; ctr <= 10; ctr++)
    {
        cout << "Input integer number " << ctr << ": ";
        cin >> num;
        sum = sum + num;            // accumulate sum of all inputs
    }

    cout << "The sum of the integers is " << sum << "\n";
}
```

break and *continue* Statements

- break
  - We have already seen the use of the *break* statement when we discussed the *switch-case* statement.
  - There is actually no difference in the semantics if being used in a loop structure.
  - If executed, it terminates the loop structure that contains it.

Example (*break*)

```
main()
{
    int num;                    // this is for the input value
    int sum;                    // this is for the accumulator

    sum = 0;                    // initialization

    for (;;)                    // this will result into an infinite loop
    {
        cout << "Input an integer: ";
        cin >> num;

        if (num == 0)                   // checks if input is 0
            break;                      // breaks out of the loop

        sum = sum + num;                // accumulate sum of all inputs
    }

    cout << "The sum of the integers is " << sum << "\n";
}
```

break and *continue* Statements (continued…)

- *continue*
  - can only used inside loops.
  - When executed, it transfers control to the <*change of state*> statement of the *for* loop and, and to the <*expression*> part of the *while* or *do-while* loop skipping any statements that follow it.

Example (*continue*)

```
main()
{
    int ctr;                        // this is for the loop counter
    int sum;                        // this is for the accumulator

    sum = 0;                        // initialization

    for (ctr =1; ctr<=10; ctr++)
    {
        if (ctr % 2 == 0)           // checks if ctr is divisible by 2
            continue;               // jumps to next iteration of ctr

        sum = sum + ctr;            // accumulate sum of odd numbers
    }

    cout << "The sum of the integers is " << sum << "\n";
}
```

Practical Exercises #1

Question 1: Quarters in a year:

- {Jan-March = 1st}
- {April-Jun = 2nd }
- {July-Sept = 3rd }
- {Oct-Dec = 4th }

✓ Write a nested if – else statements to make selections between the quarters

✓ Write an alternative switch-case to make selections within the quarters

Practical Exercises #2

Question 2: Write a program to countdown using; (i.e 10-1)

1. while-loop
2. For loop
3. Do while loop

Reference Text

- Robert Lafore, Object oriented programming in C++. Fourth edition, Sams publishing, 2002.