

Lecture 7

Control Structures

Shibo Li

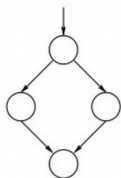
shiboli@cs.fsu.edu



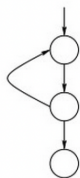
Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

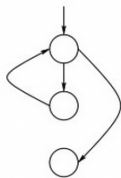
Control flow refers to the specification of the order in which the individual statements, instructions or function calls of an imperative program are executed or evaluated



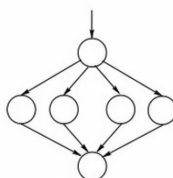
if-then-else



do until



while



case



for

Flow of control through any given function is implemented with three basic types of control structures:

- ▶ **Sequential:** Default mode. Statements are executed line by line.
- ▶ **Selection:** Used for decisions, branching – choosing between 2 or more alternative paths.
 - ▶ `if`
 - ▶ `if-else`
 - ▶ `switch`
 - ▶ other conditional states

- ▶ **Repetition:** Used for looping – repeating a piece of code multiple times in a row.
 - ▶ while
 - ▶ do-while
 - ▶ for
- ▶ The function construct, itself, forms another way to affect flow of control through a whole program. This will be discussed later in the course.

- ▶ **Selection** and **repetition** statements typically involve decision steps. These steps rely on conditions that are evaluated as **true** or **false**
- ▶ C++ has a boolean data type (called `bool`) that has values `true` and `false`. Improves readability.
- ▶ Most functions that answer a yes/no question (or a true/false situation) will return a boolean answer (or in the case of user-defined functions, they should be coded that way)

- ▶ Important: ANY C++ expression that evaluates to a value (i.e. any R-value) can be interpreted as a true/false condition. The rule is:
 - ▶ If an expression evaluates to 0, its truth value is false
 - ▶ If an expression evaluates to non-zero, its truth value is true

Relational Operators are use for comparison.

The comparison operators in C++ work much like the symbols we use in mathematics. Each of these operators returns a Boolean value: a true or a false.

```
x == y // x is equal to y
x != y // x is not equal to y
x <y  // x is less than y
x <= y // x is less than or equal to y
x >y  // x is greater than y
x >= y // x is greater than or equal to y
```

C++ has operators for combining expressions. Each of these operators returns a boolean value: a true or a false.

```

!x // the NOT operator (negation) true if x is false
x && y // the AND operator true if both x and y are
      true
x || y // the OR operator true if either x or y or
      both are true
    
```

These operators will be commonly used as test expressions in selection statements or repetition statements (loops).

<i>NOT</i>	
<i>x</i>	<i>x'</i>
0	1
1	0

<i>AND</i>		
<i>x</i>	<i>y</i>	<i>xy</i>
0	0	0
0	1	0
1	0	0
1	1	1

<i>OR</i>		
<i>x</i>	<i>y</i>	<i>x+y</i>
0	0	0
0	1	1
1	0	1
1	1	1

<i>XOR</i>		
<i>x</i>	<i>y</i>	<i>x⊕y</i>
0	0	0
0	1	1
1	0	1
1	1	0

```
(x >0 && y >0 && z >0) // all three of (x, y, z) are
    positive
(x <0 || y <0 || z <0) // at least one of the three
    variables is negative
( numStudents >= 20 && !(classAvg <70)) // there are
    at least 20 students and the class average is at
    least 70
( numStudents >= 20 && classAvg >= 70) // means the
    same thing as the previous expression
```

- ▶ The `&&` (AND) and `||` (OR) operators also have a feature known as short-circuit evaluation.
- ▶ In the Boolean AND expression (`X && Y`), if `X` is false, there is no need to evaluate `Y` (so the evaluation stops).

<code>x</code>	<code>y</code>	<code>x && y</code>
False	False	False
False	True	False
True	False	False
True	True	True

Table: Truth table for AND operator

► Example

```
(d != 0 && n / d > 0)
```

- Notice that the short circuit is crucial in this one. If d is 0, then evaluating (n / d) would result in division by 0 (illegal). But the "short-circuit" prevents it in this case. If d is 0, the first operand ($d \neq 0$) is false. So the whole $\&\&$ is false.

- ▶ Similarly, for the Boolean OR operation ($X \ || \ Y$), if the first part is true, the whole thing is true, so there is no need to continue the evaluation. The computer only evaluates as much of the expression as it needs. This can allow the programmer to write faster executing code.

x	y	$x \ \ y$
False	False	False
False	True	True
True	False	True
True	True	True

Table: Truth table for OR operator

- ▶ The most common selection statement is the if/else statement. Basic syntax:

```
if (expression)
{
    statement(s)
}
else
{
    statement(s)
}
```

- ▶ The else clause is optional.
- ▶ If there is **only one statement** in the if/else clause, the {} is also optional

The expression part can be any expression that evaluates a value (an R-value), and it must be enclosed in parentheses ().


- ▶ The best use is to make the expression a Boolean expression, which is an operation that evaluates to true or false
- ▶ For other expressions (like $x + y$), for instance):
 - ▶ an expression that evaluates to 0 is considered false
 - ▶ an expression that evaluates to anything else (non-zero) is considered true

- ▶ The statement parts are the “bodies” of the if-clause and the else-clause. The statement after the if or else clause must be either:
 - ▶ an empty statement
 - ▶ a statement
 - ▶ a block
- ▶ Appropriate indentation of the bodies of the if-clause and else-clause is a very good idea (for human readability), but irrelevant to the compiler



```
if (grade >= 68) {  
    cout <<"Passing";  
}
```

If grade is below 68, we just move on.



```
int x = 4;
if (x == 0) {
    cout <<"Nothing here";
} else {
    cout <<"There is a value";
}
```

```
if (y != 4)
{
    cout <<"Wrong number";
    y = y * 2;
    counter++;
}
else
{
    cout << "That is it!";
    success = true;
}
```

Multiple statements are to be executed as a result of the condition being true or false. In this case, notice the compound statement to delineate the bodies of the if and else clauses.

- ▶ What is output when `int val = 8`

```
if (val < 5)
    cout <<"True";
else
    cout <<"False";
    cout <<"Too bad!";
```

- ▶ What is output when `int val = 4`

```
if (val < 5)
    cout <<"True";
else
    cout <<"False";
    cout <<"Too bad!";
```

- ▶ Be careful with ifs and elses. If you don't use `else if`, you may think that you've included more under an if condition than you really have.
- ▶ Indentation is only for people! It improves readability, but means nothing to the compiler.

What is the output?

```
int a = 1+2;
int b = 3;

if (a==b){
    cout << "a is euqal to b" << endl;
}
else
{
    cout << "a is not equal to b" << endl;
}
```

What is the output?

```
double a = 0.1 + 0.2;
double b = 0.3;

if (a==b)
{
    cout << "a is euqal to b" << endl;
}
else
{
    cout << "a is not equal to b" << endl;
}
```

It is not safe to directly compare if two double vars are equal or not.

```
double a = 0.1 + 0.2;
double b = 0.3;

cout << a;
// 0.30000000000000004441
cout << b;
// 0.29999999999999998890
```

```
double a = 0.1 + 0.2;
double b = 0.3;
double tol = 1e-9; // comparison tolerance

// soft comparison
if ( fabs(a-b) < tol )
    cout << "a is equal to b";
else
    cout << "a is not equal to b"
```

```
int a = 2147483647; // a = 231 - 1
int b = a + 1

if (a<b)
    cout << "a<b"
else
    cout << "a>=b"
```

```
int a = 2147483647; // a = 231 - 1
int b = a + 1

if (a<b)
    cout << "a<b"
else
    cout << "a>=b"

// answer: a >= b
// a = 2147483647
// b = -2147483648
```

What's wrong with these if-statements? Which ones are syntax errors and which ones are logic errors?

```
if (x == 1 || 2 || 3)
    cout <<"x is in the range 1-3";
```

```
if (x >5) && (y <10)
    cout <<"Yahoo!";
```

```
if (response != 'Y' || response != 'N')
    cout <<"You must type Y or N (for yes or no)";
```

A switch statement is often convenient for occasions in which there are multiple cases to choose from. The syntax format is:

```
switch (expression)
{
    case constant:
        statements
    case constant:
        statements

    ...(as many case labels as needed)

    default: // optional label
        statements
}
```

- ▶ The switch statement evaluates the expression, and then compares it to the values in the case labels. If it finds a match, execution of code jumps to that case label.
- ▶ The values in case labels must be constants, and may only be integer types, which means that you
 - ▶ This means only integer types, type char, or enumerations (not yet discussed)
 - ▶ This also means the case label must be a literal or a variable declared to be const
 - ▶ Note: You may not have case labels with regular variables, strings, floating point literals, operations, or function calls

- ▶ If you want to execute code only in the case that you jump to, end the case with a `break` statement, otherwise execution of code will "fall through" to the next case

There is a special operator known as the conditional operator that can be used to create short expressions that work like if/else statements.

```
test expr ? true expr : false expr
```

- ▶ The test expression is evaluated for true/false value. This is like the test expression of an if-statement.
- ▶ If the expression is true, the operator returns the true expression value.
- ▶ If the test expression is false, the operator returns the false expression value.
- ▶ Note that this operator takes three operands. It is a ternary operator in the C++ language

```
cout <<(x >y) ? "x is greater than y" : "x is less
    than or equal to y");
// Note that this expression gives the same result
    as the following
if (x >y)
    cout <<"x is greater than y";
else
    cout <<"x is less than or equal to y");
```

```
(x < 0 ? value = 10 : value = 20);  
  
// this gives the same result as:  
value = (x < 0 ? 10 : 20);  
// and also gives the same result as:  
if (x < 0)  
    value = 10;  
else  
    value = 20;
```
