Lecture 2 Notes: Flow of Control

1 Motivation

Normally, a program executes statements from first to last. The first statement is executed, then the second, then the third, and so on, until the program reaches its end and terminates. A computer program likely wouldn't be very useful if it ran the same sequence of statements every time it was run. It would be nice to be able to change which statements ran and when, depending on the circumstances. For example, if a program checks a file for the number of times a certain word appears, it should be able to give the correct count no matter what file and word are given to it. Or, a computer game should move the player's character around when the player wants. We need to be able to alter the order in which a program's statements are executed, the *control flow*.

2 Control Structures

Control structures are portions of program code that contain statements within them and, depending on the circumstances, execute these statements in a certain way. There are typically two kinds: *conditionals* and *loops.*

2.1 Conditionals

In order for a program to change its behavior depending on the input, there must a way to test that input. Conditionals allow the program to check the values of variables and to execute (or not execute) certain statements. C++ has *if* and *switch-case* conditional structures.

2.1.1 Operators

Conditionals use two kinds of special operators: *relational* and *logical.* These are used to determine whether some condition is true or false.

The relational operators are used to test a relation between two expressions:

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

They work the same as the arithmetic operators (e.g., a > b) but return a Boolean value of either `true` or `false`, indicating whether the relation tested for holds. (An expression that returns this kind of value is called a Boolean expression.) For example, if the variables `x` and y have been set to 6 and 2, respectively, then `x > y` returns `true`. Similarly, `x < 5` returns `false`.

The logical operators are often used to combine relational expressions into more complicated Boolean expressions:

Operator	Meaning
&&	and
||	or
!	not

The operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

The `!` operator is a unary operator, taking only one argument and negating its value:

a	!a
true	false
false	true

Examples using logical operators (assume x = 6 and y = 2):

```
!(x > 2) → false
(x > y) && (y > 0) → true
(x < y) && (y > 0) → false
(x < y) || (y > 0) → true
```

Of course, Boolean variables can be used directly in these expressions, since they hold `true` and `false` values. In fact, any kind of value can be used in a Boolean expression due to a quirk C++ has: `false` is represented by a value of `0` and anything that is not `0` is `true`. So, `"Hello, world!"` is true, `2` is true, and any `int` variable holding a non-zero value is true. This means `!x` returns `false` and `x && y` returns `true`!

2.1.2 if, if-else and *else if*

The *if* conditional has the form:

```
if(condition)
{
	statement1
	statement2
	…
}
```

The condition is some expression whose value is being tested. If the condition resolves to a value of `true`, then the statements are executed before the program continues on. Otherwise, the statements are ignored. If there is only one statement, the curly braces may be omitted, giving the form:

```
if(condition)
	statement
```

The *if-else* form is used to decide between two sequences of statements referred to as *blocks*:

```
if(condition)
{
	statementA1
	statementA2
	…
}
else
{
	statementB1
	statementB2
	…
}
```

If the condition is met, the block corresponding to the `if` is executed. Otherwise, the block corresponding to the `else` is executed. Because the condition is either satisfied or not, one of the blocks in an if-else *must* execute. If there is only one statement for any of the blocks, the curly braces for that block may be omitted:

```
if(condition)
	statementA1
else
	statementB1
```

The *else if* is used to decide between two or more blocks based on *multiple* conditions:

```
if(condition1)
{
	statementA1
	statementA2
	…
}
else if(condition2)
{
	statementB1
	statementB2
	…
}
```

If `condition1` is met, the block corresponding to the `if` is executed. If not, then *only if* `condition2` is met is the block corresponding to the `else if` executed. There may be more than one `else if`, each with its own condition. Once a block whose condition was met is executed, any `else if`s after it are ignored. Therefore, in an *if-else-if* structure, either one or no block is executed.

An `else` may be added to the end of an if-else-if. If none of the previous conditions are met, the `else` block is executed. In this structure, one of the blocks *must* execute, as in a normal if-else.

Here is an example using these control structures:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int x = 6;
6     int y = 2;
7
8     if(x > y)
9           cout << "x is greater than y\n";
10    else if(y > x)
11          cout << "y is greater than x\n";
12    else
13          cout << "x and y are equal\n";
14
15    return 0;
16 }
```

The output of this program is `x is greater than y`. If we replace lines 5 and 6 with

```
int x = 2;
int y = 6;
```

then the output is `y is greater than x`. If we replace the lines with

```
int x = 2;
int y = 2;
```

then the output is `x and y are equal`.

2.1.3 switch-case

The *switch-case* is another conditional structure that may or may not execute certain statements. However, the switch-case has peculiar syntax and behavior:

```
switch(expression)
{
      case constant1:
            statementA1
            statementA2
            ...
            break;
      case constant2:
            statementB1
            statementB2
            ...
            break;
      ...
      default:
            statementZ1
            statementZ2
            ...
}
```

The `switch` evaluates `expression` and, if `expression` is equal to `constant1`, then the statements beneath `case constant 1:` are executed until a `break` is encountered. If `expression` is not equal to `constant1`, then it is compared to `constant2`. If these are equal, then the statements beneath `case constant 2:` are executed until a `break` is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath `default:` are executed.

Due to the peculiar behavior of `switch-case`s, curly braces are not necessary for cases where

there is more than one statement (but they *are* necessary to enclose the entire `switch-case`). `switch-case`s generally have `if-else` equivalents but can often be a cleaner way of expressing the same behavior.

Here is an example using `switch-case`:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int x = 6;
6
7     switch(x) {
8           case 1:
9                 cout << “x is 1\n”;
10                break;
11          case 2:
12          case 3:
13                cout << "x is 2 or 3";
14                break;
15          default:
16                cout << "x is not 1, 2, or 3";
17    }
18
19    return 0;
20 }
```

This program will print `x is not 1, 2, or 3`. If we replace line 5 with `int x = 2;` then the program will print `x is 2 or 3`.

2.2 Loops

Conditionals execute certain statements if certain conditions are met; loops execute certain statements *while* certain conditions are met. C++ has three kinds of loops: *while, do-while,* and *for.*

2.2.1 *while* and *do-while*

The *while* loop has a form similar to the if conditional:

```
while(condition)
{
      statement1
      statement2
      …
}
```

As long as condition holds, the block of statements will be repeatedly executed. If there is only one statement, the curly braces may be omitted. Here is an example:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int x = 0;
6
7     while(x < 10)
8           x = x + 1;
```

```
9
10    cout << "x is " << x << "\n";
11
12    return 0;
13 }
```

This program will print `x is 10`.

The *do-while* loop is a variation that guarantees the block of statements will be executed *at least once:*

```
do
{
      statement1
      statement2
      …
}
while(condition);
```

The block of statements is executed and then, if the condition holds, the program returns to the top of the block. Curly braces are *always* required. Also note the semicolon after the `while` condition.

2.2.2 for

The *for* loop works like the while loop but with some change in syntax:

```
for(initialization; condition; incrementation)
{
      statement1
      statement2
      …
}
```

The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. Curly braces may be omitted if there is only one statement. Here is an example:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5
6     for(int x = 0; x < 10; x = x + 1)
7           cout << x << "\n";
8
9     return 0;
10 }
```

This program will print out the values `0` through `9`, each on its own line.

If the counter variable is already defined, there is no need to define a new one in the initialization portion of the for loop. Therefore, it is valid to have the following:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5
6   int x = 0;
7   for(; x < 10; x = x + 1) 8
            cout << x << "\n";
9
10 return 0;
11 }
```

Note that the first semicolon inside the for loop's parentheses is still required.

A for loop can be expressed as a while loop and vice-versa. Recalling that a for loop has the form

```
for(initialization; condition; incrementation)
{
      statement1
      statement2
      …
}
```

we can write an equivalent while loop as

```
initialization
while(condition)
{
      statement1
      statement2
      …
      incrementation
}
```

Using our example above,

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5
6     for(int x = 0; x < 10; x = x + 1)
7           cout << x << "\n";
8
9     return 0;
10 }
```

is converted to

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5
6     int x = 0;
7     while(x < 10) {
8           cout << x << "\n";
9           x = x + 1;
10    }
11
12    return 0;
13 }
```

The incrementation step can technically be anywhere inside the statement block, but it is good practice to place it as the last step, particularly if the previous statements use the current value of the counter variable.

2.3 Nested Control Structures

It is possible to place ifs inside of ifs and loops inside of loops by simply placing these structures inside the statement blocks. This allows for more complicated program behavior.

Here is an example using nesting if conditionals:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int x = 6;
6     int y = 0;
7
8     if(x > y) {
9           cout << "x is greater than y\n";
10          if(x == 6)
11                cout << "x is equal to 6\n";
12          else
13                cout << "x is not equalt to 6\n";
14    } else
15          cout << "x is not greater than y\n";
16
17    return 0;
18 }
```

This program will print `x is greater than y` on one line and then `x is equal to 6` on the next line.

Here is an example using nested loops:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     for(int x = 0; x < 4; x = x + 1) {
6           for(int y = 0; y < 4; y = y + 1)
7                 cout << y;
8           cout << "\n";
9     }
10
11    return 0;
12 }
```

This program will print four lines of `0123`.