Tutorial 05 – Advanced Selection & Iteration

## Introduction

The previous tutorial explored how Logic is used to control program execution. The basics of Logic are fairly simple – just whether something is true or false – but it’s possible to build up very complex logic that allows for very sophisticated programming.

The programming concepts that use Logic to control the **sequence** of program instructions are called **selection** and **iteration**. A few C/C++ constructs were introduced that implement these concepts: if() and while(). As we’ll see in this tutorial, there are a number of additional **selection** and **iteration** constructs that provide you with more tools for different situations. These constructs and some examples of how to use them effectively provide you with what you need to write good C/C++ functions.

## Advanced Selection

The basics of selection uses the if() and if..else() statements, which are very effective. These can also be chained together in numerous ways to form more complex logical constructs.

## **if..else..if()** Statements

A common form of the if..else() statement is the if..else..if() combination.

if(*expression1* is true)

{

// do something

}

else if(*expression2* is true)

{

// do something different

}

... // add more else if() as needed

This can be understood as "**if** **expression1** is true do something, **else** **if** **expression2** is true do something different". You aren't limited to just a single else..if(), you can add as many as needed: … **else** **if** **expression3** do something different, **else** **if** **expression4** do something different, and so on…

The if..else..if()syntax is often used to select between more than two options for any given comparison. To demonstrate this imagine extending a previous example with player game scores. Instead of just a single check against the high score, suppose you want to make several comparisons. Using just if() statements you might do the following:

if (player\_score > high\_score)

{

cout << "You've got the new high score!" << endl;

}

if (player\_score < high\_score)

{

cout << "You didn't get the high score" << endl;

}

if (player\_score == high\_score)

{

cout << "You matched the high score" << endl;

}

This is again inefficient and prone to side effects, so instead you could use an if..else..if() combination like so:

if (player\_score > high\_score)

{

cout << "You've got the new high score!" << endl;

}

else if (player\_score < high\_score)

{

cout << "You didn't get the high score" << endl;

}

else if (player\_score == high\_score)

{

cout << "You matched the high score" << endl;

}

In this case you might also notice that the final if() is redundant - if player\_score isn't less than or greater than **high\_score** then it **must** be less equal. Therefore the above code could be rewritten:

if (player\_score > high\_score)

{

cout << "You've got the new high score!" << endl;

}

else if (player\_score < high\_score)

{

cout << "You didn't get the high score" << endl;

}

else // no need for the last comparison (player\_score == high\_score)

{

cout << "You matched the high score" << endl;

}

Here is the flow diagram that demonstrates the if..else..if()statement for this code:

**score > high\_score**

**TRUE**

**Display "New high score"**

**FALSE**

**score < high\_score**

**TRUE**

**Display "Not high score"**

**FALSE**

**Display "matched high score”**

The general form of if..else..if()is:

if(*expression 1* is true)

{

// do something 1

}

else if(*expression 2* is true)

{

// do something 2

}

.

.

.

else if(*expression n* is true)

{

// do something n

}

else // optional

{

// if none of the previous expressions are true, do this

}

#### **Unreachable Statements**

Take care when creating long chains of if..else..if() statements as it is easy to get into a situation where the combined logic of the expressions means that later else..if() statements can never be tested, e.g.:

if (integer1 > 2)

{

cout << "Integer1 is greater than 2" << endl;

}

else if(integer1 == 1)

{

cout << " Integer1 equals 1" << endl;

}

else if(integer1 == 2)

{

cout << " Integer1 equals 2" << endl;

}

else if(integer1 == 3) // can never get here because of first if()

{

cout << " Integer1 equals 3" << endl;

}

This is a fairly contrived example, but it shows how easy it can be to end up with **unreachable code**. Make sure you check your logic carefully in these situations.

It is actually unusual for good code to use long chains of if-else-ifs . There are other constructs that are usually as efficient, but more clear and easier to use effectively.

## Nested if() Statements

A program often requires branching logic that is based on a series of choices, with each decision depending on the results of previous ones. A simple example of this would be code that determined if a number was positive, and if so whether it is even or odd. This could be coded like so:

if (integer1 > 0)

{

cout << "Integer1 is positive" << endl;

if((integer1 % 2) == 1) // modulus 2 of any odd number will be 1

{

cout << "Integer1 is odd" << endl;

}

else

{

cout << "Integer1 is even" << endl;

}

}

else

{

cout << "Integer1 is zero or negative" << endl;

}

When there are if() statements contained within other if()statements they are called **nested** if() **statements**. You can have as many nested if() statements as needed to represent a given logical algorithm, but be careful when they are complex and/or deeply nested as it's easy to get the logic wrong.

Also keep in mind that there’s a logical **AND** happening here: a statement block is executed only if all of the nested conditions leading up to it are true. In this case it is if integer1 is odd, then integer1 is odd **AND** integer1 is positive.

Here is a flow chart representing the above code; note its **'branching'** structure that is typical of this type of code structure:

**TRUE**

**FALSE**

**Integer1**

**> 0?**

**Display "POSITIVE"**

**Integer1**

**is ODD?**

**FALSE**

**TRUE**

**Display "ODD"**

**Display "EVEN"**

## **switch** Statements

Another important selection operation you will find in C/C++ is called the switch **statement**. This is a specialised form of selection that allows you to conveniently test for several different values of an **integer** or **char** variable. Let's say you wanted to write a program that did different operations on two program variables depending on a number selected by the user, for instance if the user enters 1, they are added, if 2 they are subtracted, and 3 they are multiplied, otherwise an error message is output. This could be achieved using if..else..if() statements:

int integer1 = 7;

int integer2 = 6;

int result = 0;

int choice;

cout << "Enter 1, 2 or 3" << endl;

cin >> choice;

if(choice == 1)

{

result = integer1 + integer2;

cout << "Choice 1 : integer1 + integer2 = " << result << endl;

}

else if(choice == 2)

{

result = integer1 - integer2;

cout << "Choice 2 : integer1 - integer2 = " << result << endl;

}

else if(choice == 3)

{

result = integer1 \* integer2;

cout << "Choice 2 : integer1 \* integer2 = " << result << endl;

}

else

{

cout << "Please enter 1, 2 or 3 next time" << endl;

}

To do this using a **switch** statement results in somewhat more compact and easy to read (and potentially more optimal) code :

switch (choice)

{

case 1:

result = integer1 + integer2;

cout << "Choice 1 : integer1 + integer2 = " << result << endl;

break;

case 2:

result = integer1 - integer2;

cout << "Choice 2 : integer1 - integer2 = " << result << endl;

break;

case 3:

result = integer1 \* integer2;

cout << "Choice 2 : integer1 \* integer2 = " << result << endl;

break;

default:

cout << "Please enter 1, 2 or 3 next time");

break;

}

This code introduces a few new keywords. First, the switch statement itself takes the variable to be tested in parentheses, which in this example is choice. The value of choice is checked against each of the case *n*: statements, and if a match is found the code **after** that case statement is executed.

The break statements show where the code should stop being executed for any particular case statement. When a break is encountered, execution jumps out of the switch to the statement following the switch’s closing brace '}'.

Finally the default keyword is the code that is executed if there is **no match**, just like the final else statement in the if..else..if()version. The default keyword and statement block are optional, if they are not included then there is no default action if none of the case statements match.

Here is the flow diagram that demonstrates the switch statement for this code:

**switch(choice)**

**TRUE**

**result =**

**integer1 + integer2**

**case 1:**

**FALSE**

**TRUE**

**result =**

**integer1 - integer2**

**case 2:**

**FALSE**

**TRUE**

**case 3:**

**result =**

**integer1 \* integer2**

**FALSE**

**"enter 1, 2 or 3 next time"**

**default:**

The general form of a switch statement is:

switch (*integer\_expression*)

{

case *integer\_constant\_1*:

// statement block 1;

break;

case *integer\_constant\_2*:

// statement block 2;

break;

.

.

.

case *integer\_constant\_n*:

// statement block n;

break;

default: // optional

// default statement block

break;

}

If the break statement is omitted after any given case statement block, the code continues execution through into the next one, this is called **fall through**. This is demonstrated here in some code that checks for four particular numbers:

switch (choice)

{

case 1: // note missing break statements

case 7:

case 13:

case 99:

cout << "choice is 1, 7, 13 or 99" << endl;

break;

default:

cout << " choice is not 1, 7, 13 nor 99" << endl;

break;

}

This is occasionally a useful feature, and differentiates switch from if..else..if(), but is often a cause of logical errors if a break statement is inadvertently missed where it is needed. Finally, don't forget that the value used for the selection must be an integer, a common mistake is to try to use text strings in the comparison.

### Best Practices

- Try to use **constants** or **enumerations** when possible for the case labels. It's much cleaner and easier to read than "magic numbers".

- Always have a “default” case. Even if the default does nothing, it’s a good place to catch unexpected behaviour. Add a message that says “oops – unexpected value” and it will often catch problems for you.

## The Conditional Operator

The last form of selection operation that you may come across is the **conditional operator**, which is shown as a ? symbol. This operator behaves in the same way as an if() statement, but in a compact form that acts like an expression. Its format is:

(expression) ? result1 : result2

If **expression** is **true** the conditional operator will resolve to **result1**, otherwise it will resolve to **result2**. The fact that it is itself an expression means that it can be used directly with other expressions and in assignment statements, for instance:

int integer1 = 25, integer2 = 50, largest\_integer;

largest\_integer = (integer1 > integer2) ? integer1 **:** integer2;

This simply checks if integer1 is bigger than integer2 and if so returns integer1 otherwise it returns integer2. This is shorthand for the following code:

int largest\_integer;

int integer1 = 25, integer2 = 50;

if (integer1 > integer2)

{

largest\_integer = integer1;

}

else

{

largest\_integer = integer2;

}

This form of selection can be somewhat hard to understand for those new to C/C++, but don't worry if you don't. Any program can be implemented without this, it is merely a shortcut. However you will see this a lot in code written by other programmers, so it is worth eventually getting an understanding of how the conditional operator works.

## Advanced Iteration

As you've seen, more advanced **selection** statements allow a program to use more complex logic choose **which** sections of code to execute.

There are also more sophisticated forms iteration. The while **loop** allows a program to ***repeat*** a section of code while a condition is true. It’s very useful and probably the easiest to work with. But there are a few other loop types that are very useful and more appropriate for some situations.

## **do-while()** Loops

Basic while() loops have a sibling loop that is also really useful too. If the initial expression in while() loop is **false**, the loop body **will not be executed at all**. Therefore it can be said that a while() loop will execute **zero or more times**. For example in the following code you start off with no money, so you can't do any shopping:

double my\_money = 0;

while (my\_money > 0)

{

// buy some more stuff

}

Sometimes you need to execute a loop **at least once** in order for it to do something useful. For this case there is another form of loop called the do..while() loop. The format for this loop is:

do

{

// repeat the code in this statement block (executed at least once)

}

while(*expression* is true);

Note that the while() and conditional expression come ***after*** the statement block. That’s because the expression is evaluated after the block is executed one time. So unlike a regular while() loop, a do..while() loop will execute **one or more times**. Note the do..while() loop is **ended using a semicolon**. Take care to understand the difference when you start using the two separate types of loops as the naming similarity can cause confusion.

When would want to use a do..while() loop? Think about most (if not all) games you’ve played. When the game starts up the game always draws something before you can quit out right? A simple example of a main game loop looks like this:

bool quit\_game = false;

// do startup and initialisation stuff

do

{

// draw game screen

// get user input – if user wants to quit, set quit\_game to **true**

// process input and game logic

} while (false == quit\_game);

The body of the loop (the statement block between the do and the while) is repeated until the expression becomes **false** – which in this case is when the user wants to quit the game. This is a high-level example but a really common pattern.

Here is the flow diagram of the above do..while() loop. It's basically the same as a while() loop with the order changed a bit.

**quit\_game = false**

**Draw\_Game\_Screen()**

**Get\_User\_Input()**

**quit\_game = Process\_Input\_And\_Logic()**

**quit\_game == true?**

**TRUE**

**FALSE**

## **for()** Loops

The for() loop is generally used to repeat an set of operations a specific number of times. It uses a **control variable** to determine how many times to run through the statements within the loop. The control variable is incrementing or decrementing by a fixed amount with each iteration. The control variable in for loops is often called the **loop index**.

Though in essence it is functionally equivalent to a while() loop, the unique layout of the for() loop combines the control variable initialisation, the logical expression to be tested and the control variable update together into a single place. This format often makes it easier to see how the loop is being controlled, and allows you to easily see the loop processing as distinctly separate from the code repeating in the loop body. The general form of the for() loop is:

for(*initialisation ; expression ; update*)

{

// repeat the code in this statement block if *expression* istrue

}

The most important things to note is that the **three** statements inside a for() loop **must** be separated by semicolons, and they **must** be in the order *initialisation, expression, update*. The compiler has no way of detecting if you have any of these in the wrong place, and the loop will not behave correctly in this case. Each of the declarations will now be discussed with reference to the following example:

char alpha\_letter;

for (alpha\_letter = 'a'; alpha\_letter <= 'z'; alpha\_letter++)

{

cout << alpha\_letter << endl;

}

*initialisation:*  alpha\_letter = 'a'

This occurs just once immediately before the loop starts, and is where you initialise your loop control variable. In this case it is assigning the letter **'a'** to alpha\_letter.

*expression:* alpha\_letter <= 'z'

As with a while()loop this expression is tested **before** each iteration, and only if it evaluates to **true** will the loop body be executed. Again, in the same way as a while() loop, if the expression is **false** the **first time** it is evaluated the loop body will not be executed at all. In the above example the test is if the control variable alpha\_letter is less than or equal to the letter'z'.

*update:* alpha\_letter++

The update in a for() loop occurs **at the end** of each iteration. The update declaration is where you change (increment or decrement) the loop control variable. This is where the for() loop differs in a minor way from a while() loop, as in the while() loop the control variable could be updated **at any point** in the loop body, whereas in the for() loop you normally want do this only in the update. In the example the control variable alpha\_letter is incremented by one.

The previous example the for() loop will behave in exactly the same way as a while(). In fact if you look at them side by side you can see how the while() loop statements have been moved into the for()loop structure:

|  |  |
| --- | --- |
| char alpha\_letter = 'a';  while(alpha\_letter <= 'z')  {  cout << alpha\_letter;  alpha\_letter++;  } | char alpha\_letter;  for(alpha\_letter = 'a'; alpha\_letter <= 'z'; alpha\_letter++)  {  cout << alpha\_letter);  } |

It's probably not surprising to see that the flow diagram for a for() loop is almost identical to that of a while() loop. The only difference is that the **update** operation for the control variable **must** be the last operation in the loop body.

alpha\_letter = 'a'

alpha\_letter <= 'z'

**FALSE**

**TRUE**

cout << alpha\_letter;

alpha\_letter++

...

#### More about **for()** Loops

for() loops can seem very limited and inflexible, but they are actually very flexible because the rules are not rigid. All of the loop types are in fact interchangeable, depending on how the surrounding code is writtenWith this flexibility comes risk as well – the basic elements of the loops listed above show the **safest** ways to use them. You need to be careful the more you use the flexibility.

#### Control variable declaration

The main statement of a for() loop can in fact also be used to declare the control variable, so the above code could be written:

for (char alpha\_letter = 'a'; alpha\_letter <= 'z'; alpha\_letter++)

{

cout << alpha\_letter << endl;

}

So now the for() loop is now declaring **and** initialising alpha\_letter. When a variable is declared in this way in a for loop it is treated as a variable **local** to the for() loop statement block, i.e. not visible outside the loop. You will see this type of declaration a lot in C++ code, as it is often used for one-off counters that are only used in the loop.

#### Control variable updates

The control variable update is an automatic part of the for() loop: whatever statement is in the third position is executed automatically at the end of each iteration. But that statement isn't limited to incrementing/decrementing the control variable by 1. These statements could be used in the update:

alpha\_letter += 2 // increment by 2

alpha\_letter -= 5 // decrement by 5

alpha\_letter += increment // increment by the value in the variable "increment"

; // do nothing!

#### Exiting loops early - the break Statement

You may come across a condition where you need to exit a loop immediately – when you don't want to execute the remaining statements in the block or bother with the update and checking the loop condition. In this situation the break statement is exactly what you need.

for (char alpha\_letter = 'a'; alpha\_letter <= 'z'; alpha\_letter++)

{

cout << alpha\_letter << endl;

if (alpha\_letter == 'q')

break;

}

This code will cause the loop to exit immediately if the alpha\_letter variable is 'q'.

#### Ending the current iteration - the continue statement

Similarly, you might sometime want to skip the remaining statements in loop to continue with the next iteration. The continue statement jumps to the end of the loop and executes the control update.

for (char alpha\_letter = 'a'; alpha\_letter <= 'z'; alpha\_letter++)

{

if (alpha\_letter == 'q')

continue;

cout << alpha\_letter << endl;

}

This code will cause the loop to skip over the cout statement if the alpha\_letter variable is 'q'.

Note that break and continue work for all loops types!

#### Avoiding Unintended Infinite Loops

Always be aware if there isn't some kind of variable or condition that changes within the loop, then the loop expression may **never become false** and you will end up with an **infinite loop**. An infinite loop can be useful in certain more advanced code, but in most situations infinite loops are not what you want.

### Best Practices

Each loop type is better for certain cases, and makes the code more readable and maintainable when used for those cases. for loops in particular are best for iteration through a list of some kind... It's easy to write code that uses for loops in an interesting but odd way which can be difficult to understand later. Stick to while() and do-while() loops unless you have a list of elements.

We'll get to the *best* situations for using for loops in a few weeks.