Programming Fundamentals  
Tutorial 04 - Advanced Selection and Logical Operators

## Introduction

In this tutorial you will look at a wide variety of ways of doing selection using various forms of if() statements, **conditional operators** and switch statements. Flow diagrams representing each of the different forms of selection are considered alongside example code demonstrating each form. **Logical operators** are introduced that allow more complex expressions that compare values to be evaluated.

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

Last time you were introduced to the if() statement that allowed you to make simple choices based on the **true** or **false** result of a logical expression, usually in the form of a comparison between some values. This allowed you to perform a distinct set of operations if that expression resolved to **true**, otherwise those operations were not executed. Sometimes you may want to also perform some different operations if the expression resolves to **false**, and the way to do this is to use if..else() statements. Say, for instance, you wanted to compare two integers. Here's how it would be done with if() statements alone:

int integer1;

int integer2;

scanf(“%d”, &integer1);

scanf(“%d”, &integer2);

if(integer1 == integer2)

{

printf("Numbers are equal\n");

}

if(integer1 != integer2)

{

printf"Numbers are different\n");

}

Notice that the second if uses the opposite test, i.e. **is-not-equal** instead of **is-equal**. Instead of doing two tests like this it is clearer and more efficient to use the if..else()statement, which works as follows:

if(*expression* is true)

{

// do something

}

else

{

// do something different

}

This can be understood as "**if** the expression is true do something, **else** do something different". Note that for any given evaluation of the expression the operations in **only one** of the statement blocks will be executed.

The integer comparison code can now be written:

if(integer1 == integer2)

{

// executes if **true**

printf("Numbers are equal\n");

}

else

{

// executes if **false**

printf("Numbers are different");

}

Here is the flow diagram that demonstrates the if..else()statement with the above example:

**FALSE**

**TRUE**

**Integer1 = = Integer2?**

**Display "NOT EQUAL"**

**Display "EQUAL"**

With this new syntax you could now go back and rewrite the example code from the last tutorial that output the value of a boolean as text in a more efficient way:

bool my\_bool = true;

if(my\_bool == true)

{

printf("true\n");

}

else

{

printf("false\n");

}

#### Avoiding side effects

There's another good reason for using the if..else() format in situations where different sections of code are executed on the differing outcomes of a comparison and that is to **avoid side effects**. Consider the following code that keeps track of the highest number input by the user:

int highest\_number=10;

int input\_number;

scanf(“%d”,&input\_number);

if(input\_number > highest\_number)

{

printf("highest number.\n");

highest\_number = input\_number;

}

if(input\_number <= highest\_number)

{

printf("Number is not the highest.\n");

}

Because this code considers both cases separately, if a higher number is input the first if() statement will be true, so the " **You have input the highest number**" text will be output and the highest\_number variable will be updated. However, this means that when the second if()statement is evaluated it too will be true, as highest\_number has already been updated to equal the input number, so "**Number is not the highest**" will **also** be output which is obviously incorrect. This is called a **side-effect** and obviously this should be avoided. You could of course get around this by reorganising the code, but this is difficult to manage and error prone so a better way would be:

if(input\_number > highest\_number)

{

printf("highest number.\n");

highest\_number = input\_number;

}

else

{

printf("Number is not the highest.\n");

}

Only one of the statements can be executed for any given test, so no side effect will occur.

## **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.

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 to also output if integer1 is greater than or less than integer2. Using just if() statements you might do the following:

int integer1;

int integer2;

scanf(“%d”, &integer1);

scanf(“%d”, &integer2);

if(integer1 == integer2)

{

printf("Integer1 is equal to integer2\n");

}

if(integer1 > integer2)

{

printf("Integer1 is greater than integer2\n");

}

if(integer1 < integer2)

{

printf("Integer1 is less than integert\n");

}

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

int integer1;

int integer2;

scanf(“%d”, &integer1);

scanf(“%d”, &integer2);

if(integer1 == integer2)

{

printf("Integer1 is equal to integer2\n");

}

else if(integer1 > integer2)

{

printf(" Integer1 is greater than integer2\n");

}

else if(integer1 < integer2)

{

printf(" Integer1 is less than integer2\n");

}

In this case you might also notice that the final if() is redundant - if integer1 isn't equal to or greater than integer2 then it **must** be less then. Therefore the above code could be rewritten:

if(integer1 == integer2)

{

printf("Integer1 is equal to integer2\n");

}

else if(integer1 > integer2)

{

printf(" Integer1 is greater than integer2\n");

}

else // no need for -> if(integer1 < integer2)

{

printf(" Integer1 is less than integer2\n”;

}

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

**TRUE**

**Integer1 = = Integer2?**

**Display "EQUAL"**

**FALSE**

**TRUE**

**Integer1 > Integer2?**

**Display "GREATER THAN"**

**FALSE**

**Display "Less THAN"**

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)

{

printf("Integer1 is greater than 2\n");

}

else if(integer1 == 1)

{

printf("Integer1 equals 1\n”);

}

else if(integer1 == 2)

{

printf("Integer1 equals 2\n");

}

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

{

printf("Integer1 equals 3\n");

}

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.

## 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)

{

printf("Integer1 is positive\n");

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

{

printf("Integer1 is odd\n");

}

else

{

printf("Integer1 is even\n”);

}

}

else

{

printf("Integer1 is zero or negative\n");

}

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.

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"**

## Logical Operators

So far you have used relational operators to do simple comparisons between two items. **Logical operators** allow you to evaluate more complex expressions that can involve a combination of comparisons in a single expression. For instance, let's say you are interested in testing if an integer variable's value is between 3 and 9, i.e. 4, 5, 6, 7 or 8. The code to do this could be written as follows using nested if() statements.

if(integer1 > 3)

{

if(integer1 < 9)

{

printf("Integer1 is greater than 3 and less than 9\n");

}

}

However, this is not necessary if you use the logical **AND** operator, &&:

if( (integer1 > 3) && (integer1 < 9) )

{

printf("Integer1 is greater than 3 AND less than 9\n");

}

The format of the **AND** operator && is:

(expression1) && (expression2)

The && operator evaluates to **true** only if **both** expressions evaluate to **true**. If either or both are **false** then it evaluates to **false**. This can be summarised in the following table:

|  |  |  |
| --- | --- | --- |
| **Logical AND (&&)** | | |
| **expression1** | **expression2** | **expression1 && expression2** |
| **true** | **true** | **true** |
| **true** | **false** | **false** |
| **false** | **true** | **false** |
| **false** | **false** | **false** |

Another logical operator you will often come across is **OR**, represented by || (The || characters are **not** lowercase 'L' or numeric '1' but the **vertical line** character **'|'.** On a UK keyboard it is usually above the '\' character. The format of this operator is:

(expression1) || (expression2)

The || operator evaluates to **true** if **either** expression is **true**. This can be summarised as follows:

|  |  |  |
| --- | --- | --- |
| **Logical OR (||)** | | |
| **expression1** | **expression2** | **expression1 || expression2** |
| **true** | **true** | **true** |
| **true** | **false** | **true** |
| **false** | **true** | **true** |
| **false** | **false** | **false** |

Here's a short snippet of code to demonstrate; the if()statement is being used to check if integer1 is 3 or 7:

if( (integer1 == 3) || (integer1 == 7) )

{

printf("Integer1 is either 3 OR 7\n");

}

The last logical operator is **NOT**, represented by the! symbol. This is used with a single expression in the format:

!(expression)

**NOT** simply inverts the result so applying **NOT** to a **true** expression returns **false**, and vice versa. The table summarising its operation is:

|  |  |
| --- | --- |
| **Logical NOT (!)** | |
| **expression** | **!(expression)** | |
| **true** | **false** | |
| **false** | **true** | |

The following code demonstrates its use, checking if integer1 is **not** 3 **or** 7:

if( !((integer1 == 3) || (integer1 == 7)) )

{

printf("Integer1 is NOT equal to 3 nor equal to 7\n");

}

Notice the use of brackets to ensure the logical operations are done in the right order. The **NOT** operator has a higher precedence than the other logical operators, so if brackets were not used:

if( !(integer1 == 3) || (integer1 == 7) ) // missing brackets

{

printf("Integer1 is NOT equal to 3 nor equal to 7\n"); // wrong - will be output if integer1 = 7

}

the code would not work correctly when 7 is input, as !(integer1 == 3) would be evaluated first, which would resolve to **false**, then this **false** would be **OR'd** with (integer1 == 7) which would resolve to **true**.

It is highly advisable to always use brackets in logical expressions in the same way as any other expression, even if it's possible to write it correctly without.

#### Avoid Confusion with Bitwise Operators

C/C++ also has **bitwise operators** that perform operations such as **AND** and **OR** on each bit of a variable. For instance, if variable X is set to 5 (binary 0101) and variable Y is set to 11 (binary 1011) then X **AND** Y is 1 (binary 0001) and X **OR** Y is 15 (binary 1111). The operators for bitwise **AND** and **OR** are & and |respectively.

Obviously these are very similar to the logical operators, so take care not to accidentally use the wrong operator. Incorrectly using a bitwise operator in place of a conditional operator can cause some quite hard to notice logical errors in your code.

## The Conditional Operator

One 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 return **result1**, if it is not it will return **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;

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.

## **switch** Statements

The last type of 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;

printf(”Enter 1, 2 or 3\n");

scanf(“%d”, &choice);

if(choice == 1)

{

result = integer1 + integer2;

printf("Choice 1 : integer1 + integer2 = %d\n", result);

}

else if(choice == 2)

{

result = integer1 - integer2;

printf("Choice 2 : integer1 - integer2 = %d\n", result);

}

else if(choice == 3)

{

result = integer1 \* integer2;

printf("Choice 2 : integer1 \* integer2 = %d\n", result);

}

else

{

printf("Please enter 1, 2 or 3 next time\n");

}

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;

printf("Choice 1 : integer1 + integer2 = %d\n", result);

break;

case 2:

result = integer1 - integer2;

printf("Choice 2 : integer1 - integer2 = %d\n", result);

break;

case 3:

result = integer1 \* integer2;

printf("Choice 2 : integer1 \* integer2 = %d\n", result);

break;

default:

printf("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 brackets, in this example choice. Then 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. Next the break statements show where the code should stop being executed for any particular case statement, when a break is encountered execution jumps to the next statement after the switch statement'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:**

It can be a good idea to incorporate enumerations, which are integer values, with switch statements to make the choices even clearer, for instance:

enum enum\_colours { RED, GREEN, BLUE };

enum\_colours my\_colour = GREEN;

switch (my\_colour)

{

case RED:

printf("my\_colour is RED" );

break;

case GREEN:

printf("my\_colour is GREEN" );

break;

case BLUE:

printf("my\_colour is BLUE");

break;

}

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:

printf("choice is 1, 7, 13 or 99\n");

break;

default:

printf(" choice is not 1, 7, 13 nor 99\n");

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.

## Exercises

Name exercises **[your name] Tutorial xx Exercise yy**, with appropriate xx and yy.

#### Exercise 01

[Don't worry about months and days for this exercise, unless you want to]

1. Design a program that inputs the current year and a birth year of someone
2. Calculate the age of that person. Display age appropriate messages for less than, equal to and over 18 years old.
3. Implement the design in code, using if..else()
4. Test by putting a breakpoint on the if..else()statement, and running the program several times to test each of the outcomes. Step through the code each time to see where the code is being executed.

#### Exercise 02

1. Go back and look at your **Tutorial 03 Exercise 02,** and redraw the flow diagram using the else structure.
2. Implement your new design, and test that it works as expected using the debugger.

#### Exercise 03

Design and write a program that inputs five integers from the user, tests each of them to see if they are odd or even, and keeps a count of how many are odd and how many are even. The program should then output all five numbers saying which are odd and even, and a count displaying how many are odd and even.

#### Exercise 04

Design, code and test a program that prompts the user to enter an exam mark and prints out the appropriate grade or an error message if the mark is invalid:

|  |  |
| --- | --- |
| **Mark as %** | **Grade awarded** |
| **0 to 39** | fail |
| **40 to 49** | 3rd |
| **50 to 59** | 2:2 |
| **60 to 69** | 2:1 |
| **70 to 100** | 1st |
| **any other** | invalid mark entered. |

#### Exercise 05

Design an algorithm that fulfils the following, using a single flow chart:

1. If it is July then I will fly to Egypt. If there is a sandstorm I will stay in the hotel and go for a swim, otherwise I will go for a camel ride, then visit the pyramids.
2. If it is December I will go to the Alps. If there is snow, I will go skiing, otherwise I will eat fondue.
3. At any other time I have to go to work.

This exercise may sound familiar. Go back to your Tutorial 02 exercises to see how you can improve on your original design, if at all. You will need to add basic user input at various points, e.g. input 1 for July, 2 for December, 3 for other, and appropriate outputs.

Implement your final design. What type of selection is most appropriate here?

#### Exercise 06

Design and write a program that inputs two integers number1 and number2. Using only single if() statements (no **else**) and logical operators report back to the user if:

1. number1 is between 10 and 20 inclusive.
2. number1 or number2 is equal to 1234.
3. number1 is greater than 0 and number2 is between 12 and 32.
4. number1 is greater than number2 or number1 negative.
5. number1 is NOT between 10 and 20 inclusive. Do this 2 different ways.
6. number1 is equal to 6 or 9, and number2 is equal to 2 or 3.
7. (number1 is equal to 6 or 9, and number2 is equal to 2 or 3) or number1 is greater than number2

#### Exercise 07

An integer variable contains a value of 1 that represents Monday, 3 for Tuesday, 5 for Wednesday, 6 for Thursday, 8 for Friday, 2 for Saturday and 7 for Sunday.

1. Design and write a program using if() statements to input a number into this variable and output "weekday" and "weekend" as appropriate.
2. Design and write the same program using switch statements.
3. Explain in your logbook which you find is the best solution, and why. Can you think of a better solution?