This document is part of the HTML publication "**An Introduction to the Imperative Part of C++**"

The original version was produced by **Rob Miller** at **Imperial College London**, September 1996.

Version 1.1 (modified by **David Clark** at **Imperial College London**, September 1997)

Version 1.2 (modified by **Bob White** at **Imperial College London**, September 1998)

Version 1.3, 1.4, 2.0, ..., 2.15 (modified by **William Knottenbelt** at **Imperial College London**, September 1999-September 2016)

5. Branch and Loop Statements

5.1 Boolean Values, Expressions and Functions

In this lecture we will look more closely at branch and loop statements such as "for" and "while" loops and "if ... else" statements. All these constructs involve the evaluation of one or more logical (or "Boolean") expressions, and so we begin by looking at different ways to write such expressions.

As we have seen, in reality C++ represents "True" as the integer 1, and "False" as 0. However, expressions such as

```
condition1 == 1
```

or

```
condition2 == 0
```

aren't particularly clear - it would be better to be able to follow our intuition and write

```
condition1 == True
```

and

```
condition2 == False
```

Furthermore, it is desirable to have a separate type for variables such as "`condition1`", rather than having to declare them as of type "`int`". We can achieve all of this with a *named enumeration*:

```
enum Logical {False, True}
```

which is equivalent to

```
enum Logical {False = 0, True = 1}
```

This line acts a kind of *type definition* for a new data type "`Logical`", so that lower down the program we can add variable declarations such as:

```
Logical condition1, condition2;
```

Indeed, we can now use the identifier "`Logical`" in exactly the same way as we use the identifiers "`int`", "`char`", etc. In particular, we can write functions which return a value of type "`Logical`". The following example program takes a candidate's age and test score, and reports whether the candidate has passed the test. It uses the following criteria: candidates between 0 and 14 years old have a pass mark of 50%, 15 and 16 year olds have a pass mark of 55%, over 16's have a pass mark of 60%:

```
#include <iostream>
using namespace std;

enum Logical {False, True};

Logical acceptable(int age, int score);

/* START OF MAIN PROGRAM */
int main()
{
        int candidate_age, candidate_score;

        cout << "Enter the candidate's age: ";
        cin >> candidate_age;
        cout << "Enter the candidate's score: ";
        cin >> candidate_score;

        if (acceptable(candidate_age, candidate_score))
                cout << "This candidate passed the test.\n";
        else
                cout << "This candidate failed the test.\n";

        return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO EVALUATE IF TEST SCORE IS ACCEPTABLE */
Logical acceptable(int age, int score)
{
        if (age <= 14 && score >= 50)
                return True;
        if (age <= 16 && score >= 55)
                return True;
        if (score >= 60)
                return True;
        return False;
}
/*END OF FUNCTION */
```

Program 5.1.1

Note that since "`True`" and "`False`" are constants, it makes sense to declare them outside the scope of the main program, so that the type "`Logical`" can be used by every function in the file.

An alternative way to write the above function "`acceptable(...)`" would be:

```
/* FUNCTION TO EVALUATE IF TEST SCORE IS ACCEPTABLE */
Logical acceptable(int age, int score)
{
        Logical passed_test = False;

        if (age <= 14 && score >= 50)
                passed_test = True;
        else if (age <= 16 && score >= 55)
                passed_test = True;
        else if (score >= 60)
                passed_test = True;

        return passed_test;
}
/*END OF FUNCTION */
```

Defining our own data types (even if for the moment they're just sub-types of "`int`") brings us another step closer to object-oriented programming, in which complex types of data structure (or *classes* of *objects*) can be defined, each with their associated libraries of operations.

Note: The Identifiers "`true`" and "`false`" in C++

Note that C++ implicitly includes the named enumeration

```
enum bool {false, true};
```

So you can't (re)define the all-lower-case constant identifiers "`true`" and "`false`" for yourself. In addition, you can use the type `bool` in the same way as we used `Logical` in our example.

(BACK TO COURSE CONTENTS)

5.2 "For", "While" and "Do ... While" Loops

We have already been introduced to "for" loops in Lecture 2 and to "while" loops in Lecture 4. Notice that any "for" loop can be re-written as a "while" loop. For example, Program 2.2.2 from Lecture 2, which was:

```
#include <iostream>
using namespace std;

int main()
{
        int number;
        char character;

        for (number = 32 ; number <= 126 ; number = number + 1) {

                character = number;
                cout << "The character '" << character;
                cout << "' is represented as the number ";
                cout << number << " in the computer.\n";
        }

        return 0;
}
```

Program 2.2.2

can be written equivalently as

```
#include <iostream>
using namespace std;

int main()
{
        int number;
        char character;

        number = 32;
        while (number <= 126)
        {
                character = number;
                cout << "The character '" << character;
                cout << "' is represented as the number ";
                cout << number << " in the computer.\n";
                number++;
        }

        return 0;
}
```

Program 5.2.1

Moreover, any "while" loop can be trivially re-written as a "for" loop - we could for example replace the line

```
while (number <= 126)
```

with the line

```
for ( ; number <= 126 ; )
```

in the program above.

There is a third kind of "loop" statement in C++ called a *"do ... while" loop*. This differs from "for" and "while" loops in that the statement(s) inside the {} braces are always executed once, before the repetition condition is even checked. "Do ... while" loops are useful, for example, to ensure that the program user's keyboard input is of the correct format:

```
...
...
do
{
        cout << "Enter the candidate's score: ";
        cin >> candidate_score;
        if (candidate_score > 100 || candidate_score < 0)
                cout << "Score must be between 0 and 100.\n";
}
while (candidate_score > 100 || candidate_score < 0);
...
...
```

Program Fragment 5.2.2a

This avoids the need to repeat the input prompt and statement, which would be necessary in the equivalent "while" loop:

```
...
...
```

```
cout << "Enter the candidate's score: ";
cin >> candidate_score;
while (candidate_score > 100 || candidate_score < 0)
{
        cout << "Score must be between 0 and 100.\n";
        cout << "Enter the candidate's score: ";
        cin >> candidate_score;
}
...
...
```

Program Fragment 5.2.2b

(BACK TO COURSE CONTENTS)

5.3 Multiple Selection and Switch Statements

We have already seen (Lecture 1) how "if" statements can be strung together to form a "multiway branch". Here's a simplified version of the previous example:

```
...
...
if (total_test_score >=0 && total_test_score < 50)
        cout << "You are a failure - you must study much harder.\n";
else if (total_test_score < 60)
        cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
        cout << "You have done quite well.\n";
else if (total_test_score <= 100)
        cout << "Your score is excellent - well done.\n";
else
        cout << "Incorrect score - must be between 0 and 100.\n";
...
...
```

Because multiple selection can sometimes be difficult to follow, C++ provides an alternative method of handling this concept, called the *switch* statement. "Switch" statements can be used when several options depend on the value of a single variable or expression. In the example above, the message printed depends on the value of "`total_test_score`". This can be any number between 0 and 100, but we can make things easier to handle by introducing an extra integer variable "`score_out_of_ten`", and adding the assignment:

```
score_out_of_ten = total_test_score / 10;
```

The programming task is now as follows: (i) if "`score_out_of_ten`" has value 0, 1, 2, 3 or 4, print "You are a failure - you must study much harder", (ii) if "`score_out_of_ten`" has value 5, print "You have just scraped through the test", (iii) if "`score_out_of_ten`" has value 6 or 7, print "You have done quite well", and finally (iv) if "`score_out_of_ten`" has value 8, 9 or 10, print "Your score is excellent - well done". Here's how this is achieved with a "switch" statement:

```
...
...
score_out_of_ten = total_test_score / 10;

switch (score_out_of_ten)
{
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:     cout << "You are a failure - you ";
```

```
                        cout << "must study much harder.\n";
                        break;

            case 5:     cout << "You have just scraped through the test.\n";
                        break;

            case 6:
            case 7:     cout << "You have done quite well.\n";
                        break;

            case 8:
            case 9:
            case 10:    cout << "Your score is excellent - well done.\n";
                        break;

            default:    cout << "Incorrect score - must be between ";
                        cout << "0 and 100.\n";
    }
    ...
    ...
```

Program 5.3.1

In general, the syntax of a "switch" statement is (approximately):

```
switch (selector)
{
    case label1:    <statements 1>
                    break;
    ...
    ...
    ...

    case labelN:    <statements N>
                    break;

    default:        <statements>
}
```

There are several things to note about such "switch" statements:

- The statements which are executed are exactly those between the first label which matches the value of selector and the first "`break`" after this matching label.
- The "`break`" statements are optional, but they help in program efficiency and clarity and should ideally always be used to end each case. When a "`break`" is encountered within a case's statement, control is transfered immediately to the first program statement following the entire "switch" statement. Otherwise, execution continues.
- The selector can have a value of any ordinal type (e.g. "`char`" or "`int`" but not "`float`").
- The "`default`" is optional, but is a good safety measure.

(BACK TO COURSE CONTENTS)

5.4 Blocks and Scoping

We have already seen how *compound statements* in C++ are delimited by "`{}`" braces. These braces have a special effect on variable declarations. A compound statement that contains one

or more variable declarations is called a *block*, and the variables declared within the block have the block as their *scope*. In other words, the variables are "created" each time the program enters the block, and "destroyed" upon exit. If the same identifier has been used both for a variable inside and a variable outside the block, the variables are unrelated. While in the block, the program will assume by default that the identifier refers to the inner variable - it only looks outside the block for the variable if it can't find a variable declaration inside. Hence the program

```
#include <iostream>
using namespace std;

int integer1 = 1;
int integer2 = 2;
int integer3 = 3;

int main()
{
        int integer1 = -1;
        int integer2 = -2;
        {
                int integer1 = 10;
                cout << "integer1 == " << integer1 << "\n";
                cout << "integer2 == " << integer2 << "\n";
                cout << "integer3 == " << integer3 << "\n";
        }
        cout << "integer1 == " << integer1 << "\n";
        cout << "integer2 == " << integer2 << "\n";
        cout << "integer3 == " << integer3 << "\n";

        return 0;
}
```

Program 5.4.1

produces the output

```
integer1 == 10
integer2 == -2
integer3 == 3
integer1 == -1
integer2 == -2
integer3 == 3
```

The use of variables local to a block can sometimes be justified because it saves on memory, or because it releases an identifier for re-use in another part of the program. The following program prints a series of "times tables" for integers from 1 to 10:

```
#include <iostream>
using namespace std;

int main()
{
        int number;

        for (number = 1 ; number <= 10 ; number++)
        {
                int multiplier;

                for (multiplier = 1 ; multiplier <= 10 ; multiplier++)
                {
                        cout << number << " x " << multiplier << " = ";
                        cout << number * multiplier << "\n";
                }
```

```
                cout << "\n";
        }
        ...
        ...
```

Program 5.4.2

However, we can achieve the same effect, and end up with a clearer program, by using a function:

```
#include <iostream>
using namespace std;

void print_times_table(int value, int lower, int upper);

int main()
{
        int number;

        for (number = 1 ; number <= 10 ; number++)
        {
                print_times_table(number,1,10);
                cout << "\n";
        }
        ...
        ...
}

void print_times_table(int value, int lower, int upper)
{
        int multiplier;

        for (multiplier = lower ; multiplier <= upper ; multiplier++)
        {
                cout << value << " x " << multiplier << " = ";
                cout << value * multiplier << "\n";
        }
}
```

Program 5.4.3

or eliminate all variable declarations from "`main()`" using two functions:

```
#include <iostream>

void print_tables(int smallest, int largest);

void print_times_table(int value, int lower, int upper);

int main()
{
        print_tables(1,10);
        ...
        ...
}

void print_tables(int smallest, int largest)
{
        int number;

        for (number = smallest ; number <= largest ; number++)
        {
                print_times_table(number,1,10);
                cout << "\n";
```

```
        }
}

void print_times_table(int value, int lower, int upper)
{
        int multiplier;

        for (multiplier = lower ; multiplier <= upper ; multiplier++)
        {
                cout << value << " x " << multiplier << " = ";
                cout << value * multiplier << "\n";
        }
}
```

Program 5.4.4

(BACK TO COURSE CONTENTS)

5.5 A Remark about Nested Loop Statements

The above "times table" programs illustrate how nested loop statements can be made more readable by the use of functional abstraction. By making the body of the loop into a function call, its design can be separated from the design of the rest of the program, and problems with scoping of variables and overloading of variable names can be avoided.

5.6 Summary

In this lecture we have seen how a Boolean data type can be created using a named enumeration, and how functions can be written which return "`True`" or "`False`". We have seen how these functions can be used as conditions in branch and loop statements. We have discussed the relationship between "if", "while" and "do ... while" loops, and shown how nested "if" statements can in some cases be re-written as "switch" statements. We have also seen how local variables function within blocks delimited by "`{}`" braces. The material here is also covered in more detail in Savitch, Chapter 3.

Exercises
