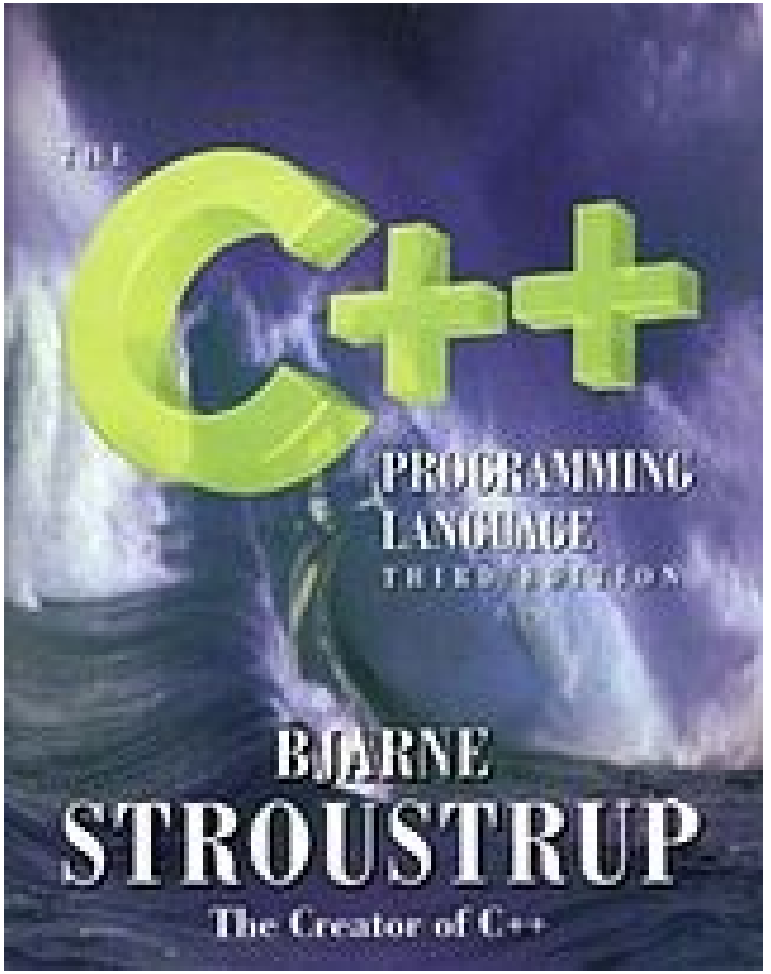
C++ Program Flow Control

Mark Slater

UNIVERSITY OF BIRMINGHAM

Overview

1. Program Flow and Scope
2. Conditionals
3. Loops
4. Using Functions



1. Program Flow and Scope

Program Flow

- It would be difficult to do much with the language if a program was just executed from top to bottom and you couldn't control what parts of the code were executed
- There are a number of ways provided to gain this control over the program:
  - Conditionals
  - Loops
  - Functions (covered in detail next week)

Program Flow – Scope (1)

- Before we go into the main ways of controlling program flow, it's important to understand the idea of scope. This refers to 'blocks of code' that are separated from each other by braces. You have already encountered one such code block in the 'main' function
- Variables declared in one code block will not be visible in an 'outer' block but will be present in an 'inner' block. When the end of a code block is reached, any local variables declared and objects created in that block are destroyed – this is termed going 'out of scope'

Program Flow – Scope (2)

Variables a and b are initialised in the outer code block

We initialise another variable called 'a' in the inner code block (this doesn't affect the previous one!) in addition to a 'c' variable

NOTE: This is a very bad idea!!

```
#include <iostream>

int main()
{   // Start of outer block
    int a{43};
    int b{21};

    {   // Start of inner block
        int a{12};
        int c{88};

        std::cout << c << std::endl;
    }   // End of inner block

    std::cout << a << std::endl;

    c = a * b;
}   // End of outer block
```

The value of 'c' is printed (88 in this case)

With the closing brace, the new variables, 'a' and 'c', go out of scope and are deleted. Hence, when 'a' is printed, it's the original value of 43.

This line will cause the compilation to fail as, though 'c' was declared in the inner block, it wasn't in the outer block and so is not present here

2. Conditionals

Program Flow - Conditionals

- Conditionals allows different code blocks to be executed based on the outcome of a simple test. It uses a number of additional operators, including:
  - Comparison: ==
  - Greater/Less than: > / <
  - Greater/Less than or equal to: >= / <=
  - Not equal to: !=
- The general syntax for this is shown here:
  - The 'if' keyword is used
  - The condition must evaluate to true or false
  - Don't confuse assignment ('=') with comparison ('==')

```
if (a == b)
{
    // Do something...
}
else
{
    // Do something else instead
}
```

Program Flow – Switch Statement

- There are several times in programming where you need to 'chain' many 'if' statements together, e.g. to do different things depending on the setting of a flag:
- These can become very hard to read

  and are also difficult to control

```
if (flag == 0)
{
    // Do something for this value
}
else if (flag == 1)
{
    // Do something else for this value
}
else
{
    // Do something for all other values
}
```

```
switch (flag)
{
    case 0:
        // Do something for this value
        break;

    case 1:
        // Do something else for this value
        break;

    default:
        // Do something for all other values
        break;
}
```

- In these cases, use the 'switch' statement:
  - Only works on integer/char types
  - Use the 'break' statement to leave a 'case' block
  - Note: If you don't use the break statement, program flow will continue to the next 'case' block

3. Loops

Program Flow – Loops (1)

- Loops are very useful for re-using code – a very important practise in all code development, not just C++. Loops allow you to repeat a code block a set number of times or until a condition is met. The first type of loop we will look at is the 'for' loop which has a syntax:

for (<initialisation>; <condition>; <loop_process>) { <code_block> }

```
for (int i{0}; i < 10; ++i)
{
    // Do something 10 times
}
```

- Some useful points to note are:
  - The initialisation step is performed at the start of the loop
  - The loop continues until the condition evaluates to 'false'
  - After every loop cycle, the process code is run
  - Be careful about the position of the semi-colons!
  - You can use 'break' to terminate a loop and 'continue' to skip to the next iteration
  - You can edit any loop variables within the loop but they only exist within the loop scope

Program Flow – Loops (2)

- The other type of loop we will look at is the 'while' loop. This is somewhat simpler than the for loop as it just loops until a condition evaluates to 'false'. The syntax is:

while (<condition>) { <code_block> }

```
int i{0};
while (i < 10)
{
    // Do something 10 times
    ++i;
}
```

Some useful points to note are:

- You must declare any loop variables before the 'while' statement
- The evaluation of the condition is done at the beginning of each loop
- The loop will continue until the condition is false, so be careful of infinite loops!
- As before, use 'break' to get out of the loop and 'continue' to skip the iteration
- You can have the evaluation done at the end of the loop using:

```
int i{0};
do
{
    ++i;
}
while (i < 10)
```

4. Using Functions

Program Flow – Using Functions

- As mentioned before, its good coding practise to reuse as much code as possible. It can also be very useful to take advantage of code other people have written so you don't have to write it yourself.
- The main way of doing this in C++ is through the use of functions that can then be called in other code blocks. At this point, the program 'jumps' to this code, executes what's there, and returns.
- To call functions, you give it's name along with any extra information (arguments) it needs. If the function returns anything, this can then be assigned to a variable or used directly

```
#include <iostream>
#include <math.h>

int main()
{
    // Will output '3'
    std::cout << sqrt(9) << std::endl;
}
```

Often need to include a different 'header' file so the compiler knows what the function is

The sqrt function takes the number you give it and returns the square root of that number

Transliterate User Input (Ex 5)

- We have now covered enough material to start writing our cipher program
- As we will be dealing with classical ciphers, we have to impose the following restrictions:
  - Letters must be only one case (we have chosen upper-case)
  - Numbers must be changed to words
  - Any other non-alphanumeric characters need to be removed
- The next exercise is to take some input from the user, change it based on the above restrictions and then print it out

Transliterate User Input (Ex 5)

- To help you with writing the transliteration code, we'll break it down into steps and you should tackle each one in turn
- Check each individual step is working and outputting what you expect before moving on to the next

```
int main(){

  // Take each letter from user input and in each case:

  //    - Convert to upper case

  //    - Change numbers to words

  //    - Ignore any other (non-alpha) characters

  //    - In each case, add result to a string variable

  // print out the new string
}
```

Use the following code to loop over user input

```
char in_char{'x'};
while(std::cin >> in_char)
{
    // Loop until the user
    // presses Enter then Ctrl+D
}
```

Best to use a 'switch' statement on the input char

Use Google to find functions that check if a char is alpha and shift it to uppercase

- Note: In C++, you need to use single quotes (') to reference a single character rather than double quotes which reference a string