Lesson 3 - Program Flow
Basic Building Blocks for Controlling Program Flow

Summary

We want to allow the user to enter numbers and sum the factorials of such numbers. When the sum exceeds 2000 we want to stop and show the user the sum. We also want to send the user messages if the sum exceeds 3000.

New Concepts

If statements, While and For loops, Nested function calls in statements.

Program Flow

Line 12 begins our function for working out the factorial of an integer value. We declare a variable `facSum` to store the ongoing sum for working out the factorial. Notice how we not only declare the variable but we also give it an initial value. This is because it is used straight away. Variables that are not initialised cannot be used as they contain "junk" (i.e. the bits in memory are not set appropriately for our purposes).

On line 15 we have a `for` loop that counts down. It is used to calculate the factorial of `x`. Notice how we declare a variable `count` and test it each loop iteration to determine if it is greater than zero. In addition, we reduce it by one each time the loop is entered. The special operator `--` simply decreases count by one (`++` would increase count by one). We could have written `count = count - 1`. On line 18 we return the `facSum` to the calling code.

```
1  #include <iostream>
2
3  using namespace std;
4
5  // We don't really need this function (we
6  // could use a plus sign), but we retain it
7  // for demonstration purposes.
8  int add(int x, int y) {
9     return x + y;
10 }
11
12 int fac(int x) {
13    int facSum = 0;
14
15    for (int count = x; count > 0; count--)
16       facSum = facSum + count;
17
18    return facSum;
19 }
```

programflow.cpp

On line 21 we declare `total` to be the variable that maintains our overall running total and initialise it to 0. We need to use it straight away in the test phase of the `while` loop. We declare `x` on line 23 to be the variable that holds the input value of the user. A `while` loop is used when we don't know how

many times the loop will go on for (see line 27). The user is typing in numbers, but they may type in 50,000 for the first number meaning the loop would only go through one iteration. Alternatively, the user may type in the value 1 many times before 2000 is reached.

Notice on line 29 how we combine two function calls in this statement. A most convenient way to do it. The `fac` function is first applied to `x` and the value returned is passed as the first parameter to the `add` function.

On line 34 we have an `if` statement, which is the fundamental way of altering the flow in a program. We test to see if the total is over 3000, informing the user if they exceeded this amount by a lot or not. Notice how the `else` part of the statement requires code to be enclosed in { }. This is because more than one statement is present. In the `while` loop we needed them but not for the `for` loop (as the `for` loop only had one statement in it).

```
20 int main() {
21    int total = 0;
22
23    int x;
24
25    cout << "Enter numbers, will finish if sum of factorials exceeds 2000\n";
26
27    while (total <= 2000) {
28       cin >> x;
29       total = add(fac(x), total);
30    }
31
32    cout << "The total is " << total << "\n";
33
34    if (total > 3000)
35       cout << "you overshot by a lot!" << "\n";
36    else {
37       cout << "You just went over" << "\n";
38       cout << "Only over by " << total - 2000 << "\n";
39    }
40
41    cout << "enter a number to exit" << "\n";
42
43    cin >> x;
44
45    return 0;
46 }
```

programflow.cpp

Under the Hood

Modern processors are quite sophisticated when it comes to managing control flow in a computer program and numerous optimisations are available depending on the compiler and hardware being used. If statements, for example, can cause modern processors to perform *branch prediction* where the processor executes the body of either an `if` or `else` statement before the `if` condition has been evaluated. As most modern processors perform the fetching and executing of instructions in a 'parallelised' pipeline, branch prediction allows processors to process the next step of their pipelines without waiting for data to be loaded from memory (i.e. the data required for the evaluation of an `if` statement). If the processor guesses correctly, then substantial time savings can be made. If the processor guesses wrong, however, then the execution must be discarded.

Some compilers provided operations to allow the programmer to specify whether the evaluation of an `if` statement is likely to be true or false, thus providing hints to branch prediction. These are not currently available with the Microsoft Visual Studio compiler, however. Many feel that this places an unwelcome burden on the developer while the compiler is capable of making

these decisions better than a human! If you're interesting in learning more, however, check out techniques such as *predication* and *loop splitting*.

Exercises

1. Write a function that accepts an integer argument and returns a boolean value. The function should return `true` if the integer argument is greater than or equal to zero, and `false` otherwise. Call this function with a positive and negative value to test that it works, using `cout` to display the result.

2. Write another function that continuously asks for numbers (use `cin`), until the user provides a negative number. You'll need some kind of loop and you should make use of the function you created in Exercise 1.

3. Recursion is an alternative to writing loops where a function "calls itself". Write another version of the factorial function that uses Recursion instead of loops. What is one advantage and one disadvantage of using Recursion as an alternative to loops?