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)

3. Functions and Procedural Abstraction

3.1 The Need for Sub-programs

A natural way to solve large problems is to break them down into a series of sub-problems, which can be solved more-or-less independently and then combined to arrive at a complete solution. In programming, this methodology reflects itself in the use of *sub-programs*, and in C++ all sub-programs are called *functions* (corresponding to both "functions" and "procedures" in Pascal and some other programming languages).

We have already been using sub-programs. For example, in the program which generated a table of square roots, we used the following "for loop":

```
...
#include<cmath>
...
...
        for (number = 1 ; number <= 10 ; number = number + 1) {
                cout.width(20);
                cout << number << sqrt(number) << "\n";
        }
...
...
```

The function "sqrt(...)" is defined in a sub-program accessed via the library file cmath (old header style math.h). The sub-program takes "number", uses a particular algorithm to compute its square root, and then returns the computed value back to the program. We don't care what the algorithm is as long as it gives the correct result. It would be ridiculous to have to explicitly (and perhaps repeatedly) include this algorithm in the "main" program.

In this chapter we will discuss how the programmer can define his or her own functions. At first, we will put these functions in the same file as "main". Later we will see how to place different functions in different files.

(BACK TO COURSE CONTENTS)

3.2 User-defined Functions

Here's a trivial example of a program which includes a user defined function, in this case called "area(...)". The program computes the area of a rectangle of given length and width.

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

int area(int length, int width);            /* function declaration */

/* MAIN PROGRAM: */
int main()
{
        int this_length, this_width;

        cout << "Enter the length: ";             /* <--- line 9 */
        cin >> this_length;
        cout << "Enter the width: ";
        cin >> this_width;
        cout << "\n";                             /* <--- line 13 */

        cout << "The area of a " << this_length << "x" << this_width;
        cout << " rectangle is " << area(this_length, this_width);

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

/* FUNCTION TO CALCULATE AREA: */
int area(int length, int width)   /* start of function definition */
{
        int number;

        number = length * width;

        return number;
}                                 /* end of function definition */
/* END OF FUNCTION */
```

Program 3.2.1

Although this program is not written in the most succinct form possible, it serves to illustrate a number of features concerning functions:

- The structure of a function definition is like the structure of "`main()`", with its own list of variable declarations and program statements.
- A function can have a list of zero or more parameters inside its brackets, each of which has a separate type.
- A function has to be declared in a function declaration at the top of the program, just after any global constant declarations, and before it can be called by "`main()`" or in other function definitions.
- Function declarations are a bit like variable declarations - they specify which type the function will return.

A function may have more than one "return" statement, in which case the function definition will end execution as soon as the first "return" is reached. For example:

```
double absolute_value(double number)
{
        if (number >= 0)
                return number;
        else
                return 0 - number;
}
```

(BACK TO COURSE CONTENTS)

3.3 Value and Reference Parameters

The parameters in the functions above are all *value parameters*. When the function is called within the main program, it is passed the values currently contained in certain variables. For example, "`area(...)`" is passed the current values of the variables "`this_length`" and "`this_width`". The function "`area(...)`" then stores these values in its own private variables, and uses its own private copies in its subsequent computation.

Functions which use Value Parameters are Safe

The idea of value parameters makes the use of functions "safe", and leads to good programming style. It helps guarantee that a function will not have hidden *side effects*. Here is a simple example to show why this is important. Suppose we want a program which produces the following dialogue:

```
Enter a positive integer:
4
The factorial of 4 is 24, and the square root of 4 is 2.
```

It would make sense to use the predefined function "`sqrt(...)`" in our program, and write another function "`factorial(...)`" to compute the factorial n! = (1 x 2 x ... x n) of any given positive integer n. Here's the complete program:

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

int factorial(int number);

/* MAIN PROGRAM: */
int main()
{
        int whole_number;

        cout << "Enter a positive integer:\n";
        cin >> whole_number;
        cout << "The factorial of " << whole_number << " is ";
        cout << factorial(whole_number);
        cout << ", and the square root of " << whole_number << " is ";
        cout << sqrt(whole_number) << ".\n";

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

/* FUNCTION TO CALCULATE FACTORIAL: */
int factorial(int number)
{
        int product = 1;

        for ( ; number > 0 ; number--)
                product *= number;

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

Program 3.3.1

By the use of a value parameter, we have avoided the (correct but unwanted) output

```
Enter a positive integer:
4
The factorial of 4 is 24, and the square root of 0 is 0.
```

which would have resulted if the function "`factorial(...)`" had permanently changed the value of the variable "`whole_number`".

(BACK TO COURSE CONTENTS)

Reference Parameters

Under some circumstances, it is legitimate to require a function to modify the value of an actual parameter that it is passed. For example, going back to the program which inputs the dimensions of a rectangle and calculates the area, it would make good design sense to package up lines 9 to 13 of the main program into a "get-dimensions" sub-program (i.e. a C++ function). In this case, we require the function to alter the values of "`this_length`" and "`this_width`" (passed as parameters), according to the values input from the keyboard. We can achieve this as follows using reference parameters, whose types are post-fixed with an "&":

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

int area(int length, int width);

void get_dimensions(int& length, int& width);

/* MAIN PROGRAM: */
int main()
{
        int this_length, this_width;
        get_dimensions(this_length, this_width);
        cout << "The area of a " << this_length << "x" << this_width;
        cout << " rectangle is " << area(this_length, this_width);

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

/* FUNCTION TO INPUT RECTANGLE DIMENSIONS: */
void get_dimensions(int& length, int& width)
{
        cout << "Enter the length: ";
        cin >> length;
        cout << "Enter the width: ";
        cin >> width;
        cout << "\n";
}
/* END OF FUNCTION */

/* FUNCTION TO CALCULATE AREA: */
int area(int length, int width)
{
        return length * width;
}
/* END OF FUNCTION */
```

Program 3.3.2

Notice that, although the function "get_dimensions" permanently alters the values of the parameters "this_length" and "this_width", it does not return any other value (i.e. is not a "function" in the mathematical sense). This is signified in both the function declaration and the function heading by the reserved word "void".

(BACK TO COURSE CONTENTS)

3.4 Polymorphism and Overloading

C++ allows *polymorphism*, i.e. it allows more than one function to have the same name, provided all functions are either distinguishable by the typing or the number of their parameters. Using a function name more than once is sometimes referred to as *overloading* the function name. Here's an example:

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

int average(int first_number, int second_number, int third_number);

int average(int first_number, int second_number);

/* MAIN PROGRAM: */
int main()
{
        int number_A = 5, number_B = 3, number_C = 10;

        cout << "The integer average of " << number_A << " and ";
        cout << number_B << " is ";
        cout << average(number_A, number_B) << ".\n\n";

        cout << "The integer average of " << number_A << ", ";
        cout << number_B << " and " << number_C << " is ";
        cout << average(number_A, number_B, number_C) << ".\n";

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

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
int average(int first_number, int second_number, int third_number)
{
        return ((first_number + second_number + third_number) / 3);
}
```

```
        /* END OF FUNCTION */

        /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
        int average(int first_number, int second_number)
        {
                return ((first_number + second_number) / 2);
        }
        /* END OF FUNCTION */
```

Program 3.4.1

This program produces the output:

```
        The integer average of 5 and 3 is 4.

        The integer average of 5, 3 and 10 is 6.
```

(BACK TO COURSE CONTENTS)

3.5 Procedural Abstraction and Good Programming Style

One of the main purposes of using functions is to aid in the *top down* design of programs. During the design stage, as a problem is subdivided into tasks (and then into sub-tasks, sub-sub-tasks, etc.), the problem solver (programmer) should have to consider only what a function is to do and not be concerned about the details of the function. The function name and comments at the beginning of the function should be sufficient to inform the user as to what the function does. (Indeed, during the early stages of program development, experienced programmers often use simple "dummy" functions or *stubs*, which simply return an arbitrary value of the correct type, to test out the control flow of the main or higher level program component.)

Developing functions in this manner is referred to as *functional* or *procedural abstraction*. This process is aided by the use of value parameters and local variables declared within the body of a function. Functions written in this manner can be regarded as "black boxes". As users of the function, we neither know nor care why they work.

(BACK TO COURSE CONTENTS)

3.6 Splitting Programs into Different Files

As we have seen, C++ makes heavy use of predefined standard libraries of functions, such as "`sqrt(...)`". In fact, the C++ code for "`sqrt(...)`", as for most functions, is typically split into two files:

- The *header file* "`cmath`" contains the function declarations for "`sqrt(...)`" (and for many other mathematical functions). This is why in the example programs which call "`sqrt(...)`" we are able to write "`#include<cmath>`", instead of having to declare the function explicitly.
- The *implementation file* "`math.cpp`" contains the actual function definitions for "`sqrt(...)`" and other mathematical functions. (In practice, many C++ systems have one or a few big file(s) containing all the standard function definitions, perhaps called "`ANSI.cpp`" or similar.)

It is easy to extend this library structure to include files for user-defined functions, such as "`area(...)`", "`factorial(...)`" and "`average(...)`". As an example, Program 3.6.1 below is the same as Program 3.4.1, but split into a main program file, a header file for the two average functions, and a corresponding implementation file.

The code in the main program file is as follows:

```
        #include<iostream>
        #include"averages.h"

        using namespace std;

        int main()
        {
                int number_A = 5, number_B = 3, number_C = 10;

                cout << "The integer average of " << number_A << " and ";
                cout << number_B << " is ";
```

```
            cout << average(number_A, number_B) << ".\n\n";

            cout << "The integer average of " << number_A << ", ";
            cout << number_B << " and " << number_C << " is ";
            cout << average(number_A, number_B, number_C) << ".\n";

            return 0;
    }
```

Program 3.6.1

Notice that whereas "include" statements for standard libraries such as "iostream" delimit the file name with angle ("<>") brackets, the usual convention is to delimit user-defined library file names with double quotation marks - hence the line " #include"averages.h" " in the listing above.

The code in the header file "averages.h" is listed below. Notice the use of the file identifier "AVERAGES_H", and the reserved words "ifndef" ("if not defined"), "define", and "endif". "AVERAGES_H" is a (global) symbolic name for the file. The first two lines and last line of code ensure that the compiler (in fact, the *preprocessor*) only works through the code in between once, even if the line "#include"averages.h"" is included in more than one other file.

Constant and type definitions are also often included in header files. You will learn more about this in the object-oriented part of the course.

```
    #ifndef AVERAGES_H
    #define AVERAGES_H

    /* (constant and type definitions could go here) */

    /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
    int average(int first_number, int second_number, int third_number);

    /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
    int average(int first_number, int second_number);

    #endif
```

averages.h

Finally, the code in the implementation file "averages.cpp" is as follows:

```
    #include<iostream>
    #include"averages.h"

    using namespace std;

    /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
    int average(int first_number, int second_number, int third_number)
    {
            return ((first_number + second_number + third_number) / 3);
    }
    /* END OF FUNCTION */

    /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
    int average(int first_number, int second_number)
    {
            return ((first_number + second_number) / 2);
    }
    /* END OF FUNCTION */
```

averages.cpp

Note the modularity of this approach. We could change the details of the code in "averages.cpp" without making any changes to the code in "averages.h" or in the main program file.

(BACK TO COURSE CONTENTS)

3.7 Summary

In this lecture we have shown how C++ programs can include user-defined functions. We have seen how functions are passed parameters, which may either be value parameters or reference parameters. We have discussed how the inclusion of functions facilitates the use of procedural abstraction in the top-down design of programs. We have also seen how function definitions may be located in separate implementation files, and accessed via header files. Most of the material here is covered in more detail in Savitch, Chapters 4 and 5.

Exercises