10

10 Functions

The most complex of the programs presented in chapters 6…9 consisted of a single `main()` function with some assignment statements that initialized variables, a loop containing a `switch` selection statement, and a few final output statements. When your data processing task grows more complex than this, you must start breaking it down into separate functions.

Functions that compute values

We have already made use of functions from the maths library. Functions like `sin()` obviously model the mathematical concept of a function as something that maps (converts) an input value in some domain into an output value in a specified range (so sine maps values from the domain (0..2π) into the range (-1.0...1.0)).

We have also used functions from the iostream and related libraries. An output statement such as:

```
   cout << "Chisquared : " << chisq;
```

Functions with "side effects"

actually involves two function calls. The `<<` "takes from" operator is a disguised way of calling a function. The calls here are to a "PrintText" function and to a "PrintDouble" function. Unlike mathematical functions such as `sin()`, these functions don't calculate a value from a given input; instead they produce some desired "side effect" (like digits appearing on the screen). Things like `cout.setf(ios::showpoint)` again involve a function call executed to achieve a side effect; in this case the side effect is to change the way the numbers would be printed subsequently.

We will be using an increasing number of different functions from various standard libraries. But now we need more, we need to be able to define our own functions.

If you can point at a group of statements and say "these statements achieve X", then you have identified a potential function. For example, in the program in section 9.8, the following statements were used to "get an integer in a given range"

```
        int interest;

        cout << "Please specify attribute of interest"
                          "(1...5) : ";
        cin >> interest;

        if((interest < 1) ||(interest >5)) {
               cout << "Sorry, don't understand, wanted"
                             " value 1..5" << endl;
               exit(1);
        }
```

"Get an integer in a given range" – that is a specification of a "function". You have identified a `GetInteger()` operation. It is going to be a function that has to be given two items of data, the low limit and the high limit (1 and 5 in this specific case), and which returns as a result an integer in that range.

Even if, as in the program in section 9.8, this "function" is only needed once, it is worth composing it as a separate function. "Abstracting" the code out and making it a separate function makes the code more intelligible. The program now consists of two parts that can be read, analyzed, and understood in isolation:

```
    function to get integer in specified range
        prompt
        input
        check
        …
        return OK integer

    main program
        get checked integer data item using function
        other initialization
        …
```

The example in section 8.5.2 (Newton's method for roots of a polynomial) illustrates another situation where it would be appropriate to invent a special purpose function. The following expression had to appear at several points in the code where we required the value of the polynomial at a particular value of x:

```
    13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2) + 16.5*x + 30
```

It is easier to read and understand code like:

```
    fun1 = polyfunction(g1);
    fun2 = polyfunction(g2);
```

than code like:

```
    double x = g1;
    fun1 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
                      + 16.5*x + 30;
    x = g2;
```

```
fun2 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
                  + 16.5*x + 30;
```

If you are reading the code with the "calls" to `polyfunction()`, you can see that `fun1` and `fun2` are being initialized with values depending on the input data values `g1`, and `g2`. The *details* of how they are being initialized don't appear and so don't disrupt you chain of thought as you read the code. In addition, the use of a separate function to evaluate the polynomial would actually save some memory. Code using calls to a function would require less instructions than the code with repeated copies of the instructions to evaluate the expression.

A long time ago, function calls were expensive. In the computers of those days, the instruction sets were different and there were limitations on use of CPU registers; together these factors meant that both the processes of setting up a call to a function, and then of getting a result back required many instructions. So, in those days, functions had to be reasonably long to make the overheads of a function call worthwhile. But, that was a very long time ago (it was about the time when the "Beatles", four young men in funny suits, were starting to attract teenagers to the Cavern club in Liverpool).

The overhead for a function call is rarely going to be of any significance on a modern computer architecture. Because functions increase understandability of programs, they should be used extensively. In any case, as explained in 10.6, C++ has a special construct that allows you to "have your cake and eat it". You can define functions and so get the benefit of additional clarity in the code, but at the same time you can tell the compiler to avoid using a function call instruction sequence in the generated code. The code for the body of the function gets repeated at each place where the function would have been called.

10.1 FORM OF A FUNCTION DEFINITION

A function definition will have the following form:

```
return_type
   function_name(
       details of data that must be given to the function
        )
{
    statements performing the work of the function
    return    value of appropriate type  ;
}
```

For example:

```
int GetIntegerInRange(int low, int high)
{
	int res;

	do {
		cout << "Enter an integer in the range " << low
			<< " ... " << high << " :";
		cin >> res;
	} while (! ((res >= low) && (res <= high)));
	return res;
}
```

The line

```
int GetIntegerInRange(int low, int high)
```

defines the name of the function to be "`GetIntegerInRange`", the return type `int`, and specifies that the function requires two integer values that its code will refer to as `low` and `high`. The definition of the local variable `res`, and the `do … while` loop make up the body of the function which ends with the `return res` statement that returns an acceptable integer value in the required range. (The code is not quite the same as that in the example in section 9.8. That code gave up if the data value input was inappropriate; this version keeps nagging the user until an acceptable value is entered.)

Function names

The rules, given in section 6.6, that define names for variables also apply to function names. Function names must start with a letter or an underscore character (_) and can contain letters, digits, and underscore characters. Names starting with an underscore should be avoided; these names are usually meant to be reserved for the compiler which often has to invent names for extra variables (or even functions) that it defines. Names should be chosen to summarize the role of a function. Often such names will be built up from several words, like Get Integer In Range.

When you work for a company, you may find that there is a "house style"; this style will specify how capitalization of letters, or use of underscores between words, should be used when choosing such composite names. Different "house styles" might require: GetIntegerInRange, or Get_integer_in_range, or numerous variants. Even if you are not required to follow a specified "house style", you should try to be consistent in the way you compose names for functions.

Argument list

After the function name, you get a parenthesised list of "formal parameters" or argument declarations (the terminology "arguments", "formal parameters" etc comes from mathematics where similar terms are used when discussing mathematical functions). You should note that each argument, like the `low` and `high` arguments in the example, requires its own type specification. This differs from variable definitions, where several variables can be defined together (as in earlier examples where we have had things like `double Obs00, Obs01, Obs10, Obs11;`).

Functions with no arguments

Sometimes, a function won't require arguments. For example, most IDEs have some form of "time used" function (the name of this function will vary). This function, let's call it `TimeUsed()`, returns the number of milliseconds of CPU time that the program has already used. This function simply makes a request to the operating system (using some special "system call") and interprets the data returned to get a time represented as a long integer. Of course the function does not require any information from the user's program so it has no arguments. Such a function can be specified in two ways:

```
long TimeUsed()
```

i.e. a pair of parentheses with nothing in between (an empty argument list), or as

```
long TimeUsed(void)
```

i.e. a pair of parentheses containing the keyword `void`. ("Void" means empty.) The first form is probably more common. The second form, which is now required in standard C, is slightly better because it make it even more explicit that the function does not require any arguments.

Trap for the unwary Pascal programmer (or other careless coder)

A call to the `TimeUsed()` function (or any similar function with a void argument list e.g. `int TPSOKTSNW(void)`) just has the form:

```
TPSOKTSNK()
```

e.g.

```
if(TPSOKTSNW()) {
    // action
    …
    }
```

In Pascal, and some other languages, a function that takes no arguments is called by just writing its function name without any `()` parentheses. People with experience in such languages sometimes continue with their old style and write things like:

```
if(TPSOKTSNW) {
    // action
    …
    }
```

This is perfectly legal C/C++ code. Its meaning is a bit odd – it means check whether the function `TPSOKTNSW()` has an address. Now if the program has linked successfully, `TPSOKTNSW()` will have an address so the `if` condition is always true. (The function `TPSOKTNSW()` – it was "`The_President_Says_OK_To_Start_Nuclear_War()`", but that name exceeded the length limits set by the style guides for the project. Fortunately, the code was checked before being put into use.)

10.2 RESULT TYPES

Like mathematical functions that have a value, most functions in programs return a value. This value can be one of the standard data types (char, long, double, etc) or may be a structure type (Chapter 16) or pointer type (introduced in Part IV, Chapter 20).

Default return type of int

If you don't define a return type, it is assumed that it should be `int`. This is a left over from the early days of C programming. You should always remember to define a return type and not simply use this default (which may disappear from the C++ language sometime).

The compiler will check that you do return a value of the appropriate type. You will probably already have had at least one error report from the compiler on some occasion where you forgot to put a `return 0;` statement at the end of your `int main()` program.

A return statement will normally have an expression; some examples are:

```
return res;

return n % MAXSIZE;

return 3.5 + x*(6.1 + x*(9.7 + 19.2*x));
```

You will sometimes see code where the style convention is to have an extra pair of parentheses around the expression:

```
return(res);

return(n % MAXSIZE);

return(3.5 + x*(6.1 + x*(9.7 + 19.2*x)));
```

Don't worry about the extra parentheses; they are redundant (don't go imagining that it is a call to some strange function called `return()`).

Functions that don't return results

If you have a function that is used only to produce a side effect, like printing some results, you may not want to return a result value. In the 1970s and early 1980s when earlier versions of C were being used, there was no provision for functions that didn't return results. Functions executed to achieve side effects were expected to return an integer result code reporting their success or failure. You can still see this style if you get to use the alternate stdio input-output library. The main print function, `printf()`, returns an integer; the value of this integer specifies the number of data items successfully printed. If you look at C code, you will see that very few programmers ever check the success status code returned by `printf()`. (Most programmers feel that if you can't print any results there isn't anything you can do about it; so why bother checking).

void

Since the success/failure status codes were so rarely checked, the language designers gave up on the requirement that all "side effect" functions return an integer success code. C++ introduced the idea of functions with a `void` return type (i.e. the return value slot is empty). This feature was subsequently retrofitted into the older C language.

For example, suppose you wanted to tidy up the output part the example program from section 8.5.1 (the example on childrens' heights). The program would be easier to follow if there was a small separate function that printed the average and standard deviation. This function would be used purely to achieve the side effect of producing some printout and so would not have a return value.

It could defined as follows:

```
void PrintStats(int num, double ave, double std_dev)
{
    cout << "The sample size was " << num << endl;
    cout << "Average height " << ave;
    cout << "cm, with a standard deviation of " <<
                  std_dev << endl;
    return;
}
```

The `return;` statement at the end can be omitted in a `void` function.

10.3 FUNCTION DECLARATIONS

When you start building larger programs where the code must be organized in separate files, you will often need to "declare" functions. You will have defined your function in one file, but code in another file needs a call to that function. The second file will need to contain a declaration describing the function, its return type, and its argument values. This is necessary so that the compiler (and linker) can know that the function does exist and so that they can check that it is being used correctly (getting the right kinds of data to work with, returning a result that suits the code of the call).

A function declaration is like a function definition – just without the code! A declaration has the form:

```
return_type
   function_name(
       details of data that must be given to the function
        ) ;
```

The end of a *declaration* has to be marked (because otherwise the compiler, having read the argument list, will be expecting to find a `{` begin bracket and some code). So, function declarations end with a semicolon after the closing parenthesis of the argument list. (Similarly, it is a mistake to put a semicolon after the closing parenthesis and before the body in a function *definition*. When the compiler sees the semicolon, it "knows" that it just read a function declaration and so is expecting to find another declaration, or the definition of a global variable or constant. The compiler gets very confused if you then hit it with the `{ code }` parts of a function definition.)

Function "prototypes"

Examples of function declarations (also known as function "prototypes") are:

```
void PrintStats(int num, double ave, double std_dev);
int GetIntegerInRange(int low, int high);
```

In section 6.5.2, part of the math.h header file was listed; this contained several function declarations, e.g.:

```
double sin(double);
double sqrt(double);
double tan(double);
double log10(double);
```

Most declarations are in header files

Most function declarations appear in header files. But, sometimes, a function declaration appears near the start of a file with the definition appearing later in the same file.

Unlike languages such as Pascal (which requires the "main program" part to be the last thing in a file), C and C++ don't have any firm rules about the order in which functions get defined.

For example, if you were rewriting the heights example (program 8.5.1) to exploit functions you could organize your code like this:

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

double Average(double total, int number)
{
    return total/number;
}

double StandardDev(double sumsquares, double average, int num)
{
    return sqrt(
           (sumsquares - num*average*average) /
                  (num-1));
}

void PrintStats(int num, double ave, double std_dev)
{
    …
}

int main()
{
    int           children, boys, girls;
    double        cSum, bSum, gSum;
    double         cSumSq, bSumSq, gSumSq;

    …

    cout << "Overall results all children:" << endl;
    average = Average(cSum, children);
    standard_dev = StandardDev(cSumSq, average children);
    PrintStats(children, average, standard_dev);

    cout << endl;
    cout << "Results for girls only:" << endl;

    …

    return 0;
}
```

Alternatively, the code could be arranged like this:

Forward declarations

```
#include <iostream.h>
#include <math.h>
double Average(double total, int number);
double StandardDev(double sumsquares, double average, int num);
void PrintStats(int num, double ave, double std_dev);

int main()
{
    int            children, boys, girls;
    double         cSum, bSum, gSum;
    double         cSumSq, bSumSq, gSumSq;

    …

    cout << "Overall results all children:" << endl;
    average = Average(cSum, children);
    standard_dev = StandardDev(cSumSq, average children);
    PrintStats(children, average, standard_dev);

    cout << endl;

    cout << "Results for girls only:" << endl;
    …

    return 0;
}

void PrintStats(int num, double ave, double std_dev)
{
    …
}
```

```
double Average(double total, int number)
{
    return total/number;
}

double StandardDev(double sumsquares, double average, int num)
{
    return sqrt(
           (sumsquares - num*average*average) /
                  (num-1));
}
```

This version requires declarations for functions `Average()` etc at the start of the file. These declarations are "forward declarations" – they simply describe the functions that are defined later in the file.

These forward declarations are needed for the compiler to deal correctly with the function calls in the main program (`average = Average(cSum, children);` etc). If the compiler is to generate appropriate code, it must know that `Average()` is a function that returns a double and takes two arguments – a double and an integer. If you didn't put in the forward declaration, the compiler would give you an error message like "Function Average has no prototype."

Normally, you should use the first style when you are writing small programs that consist of `main()` and a number of auxiliary functions all defined in the one file. Your programs layout will be vaguely reminiscent of Pascal with `main()` at the bottom of the file. There are a few circumstances where the second arrangement is either necessary or at least more appropriate; examples will appear later.

There is another variation for the code organization:

Prototypes declared in the body of the code of caller

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

int main()
{
    double Average(double total, int number);
    double StandardDev(double sumsquares,
                  double average, int num);
    void PrintStats(int num, double ave, double std_dev);

    int           children, boys, girls;
    double        cSum, bSum, gSum;

    …
    average = Average(cSum, children);
    standard_dev = StandardDev(cSumSq, average children);
    PrintStats(children, average, standard_dev);
    …
    return 0;
}
```

```
void PrintStats(int num, double ave, double std_dev)
{
    …
}

double Average(double total, int number)
{
    return total/number;
}

double StandardDev(double sumsquares, double average, int num)
{
…
}
```

There is no advantage to this style. It introduces an unnecessary distinction between these functions and those functions, like `sqrt()`, whose prototypes have appeared in the header files that have been #included. Of course, as you would realize, a declaration of a function prototype in the body of another function does *not* result in that function being called at the point of declaration!

10.4 DEFAULT ARGUMENT VALUES

Some functions need to be given lots of information, and therefore they need lots of arguments. Sometimes, the arguments may almost always get to be given the same data values.

For example, if you are writing are real window-based application you will not be using the iostream output functions, instead you will be using some graphics functions. One might be `DrawString()`, a function with the following specification:

```
void DrawString(char Text[], int TextStyle,
            int TextSize, int Hoffset, int Voffset)
```

This function plots the message Text in the current window.
TextStyle defines the style, 0 Normal, 1 Italic, 2 Bold, 3 Bold Italic.
TextSize defines the size, allowed sizes are 6, 9, 10, 12, 14, 18, 24, 36, 48.
Hoffset and Voffset are the horizontal offsets for the start of the string relative to the current pen position

In most programs, the current pen position is set before a call to `DrawString()` so, usually, both `Hoffset` and `Voffset` are zero. Graphics programs typically have a default font size for text display, most often this is 12 point. Generally, text is displayed in the normal ("Roman") font with only limited use of italics and bold fonts. About the only thing that is different in every call to `DrawString()` is the message.

So, you tend to get code like:

```
…
DrawString("Enter your bet amount", 0, 12, 0, 0);
…
DrawString("Press Roll button to start game", 0, 12, 0, 0);
…
DrawString("Lock any columns that you don't want rolled again",
                   0, 12, 0, 0);
…
DrawString("Congratulations you won", 3, 24, 0, 0);
…
DrawString("Double up on your bet?", 3, 12, 0, 0);
…
```

C++ permits the definition of default argument values. If these are used, then the arguments don't have to be given in the calls.

If you have a declaration of `DrawString()` like the following:

```
void DrawString(char Text[], int TextStyle = 0,
           int TextSize = 12,
           int Hoffset = 0, int Voffset = 0);
```

This declaration would allow you to simplify those function calls:

Omitting trailing arguments that have default values

```
…
DrawString("Enter your bet amount");
…
DrawString("Press Roll button to start game");
…
DrawString("Lock any columns that you don't want rolled
again");
…
DrawString("Congratulations you won", 3, 24);
…
DrawString("Double up on your bet?", 3);
…
```

You are permitted to omit the last argument, or last pair of arguments, or last three arguments or whatever if their values are simply the normal default values. (The compiler just puts back any omitted values before it goes and generates code.) You are only allowed to omit trailing arguments, you aren't allowed to do something like the following:

Can't arbitrarily omit arguments

```
DrawString("Congratulations you won", , 24);
```

Default arguments can appear in function declarations, or in the definition of the function (but not both). Either way, details of these defaults must be known to the compiler before the code where a call is made. These requirements affect the layout of programs.

10.5 TYPE SAFE LINKAGE, OVERLOADING, AND NAME MANGLING

The C++ compiler and its associated linking loader don't use the function names that the programmer invented!

Before it starts into the real work of generating instructions, the C++ systematically renames all functions. The names are elaborated so that they include the name provided by the programmer and details of the arguments. The person who wrote the compiler can choose how this is done; most use the same general approach. In this approach, codes describing the types of arguments are appended to the function name invented by the programmer.

Using this kind of renaming scheme, the functions `TimeUsed`, `Average`, `StandardDev`, and `PrintStats`:

```
void TimeUsed(void);
double Average(double total, int number);
double StandardDev(double sumsquares,
                   double average, int num);
void PrintStats(int num, double ave, double std_dev);
```

Mangled names

would be assigned compiler generated (or "mangled") names like:

```
__TimeUsed__fv
__Average__fdfi
__StandardDev_f2dfi
__PrintStats__fif2d
```

You might well think this behaviour strange. Why should a compiler writer take it on themselves to rename your functions?

Type safe linkage

One of the reasons that this is done is this makes it easier for the compiler and linking loader to check consistency amongst separate parts of a program built up from several files. Suppose for instance you had a function defined in one file:

```
void Process(double delta, int numsteps)
{
    …
}
```

In another file you might have incorrectly declared this function as:

```
void Process(int numsteps, double delta);
```

and made calls like:

```
Process(11, 0.55);
```

If there were no checks, the program would be compiled and linked together, but it would (probably) fail at run-time and would certainly produce the wrong results because the function `Process()` would be getting the wrong data to work with.

The renaming process prevents such run-time errors. In the first file, the function gets renamed as:

```
__Process__fdfi
```

while in the second file it is:

```
__Process__fifd
```

When the linking loader tried to put these together it would find the mismatch and complain (the error message would probably say something like "Can't find __Process_fifd", this is one place where the programmer sometimes gets to see the "mangled" names).

Suppressing the name mangling process for C libraries

This renaming occasionally causes problems when you need to use a function that is defined in an old C language library. Suppose the C library has a function `void seed(int)`, normally if you refer to this in a piece of C++ code the compiler renames the function making it `__seed__fi`. Of course, the linking loader can't find this function in the C library (where the function is still called `seed`). If you need to call such a function, you have to warn the C++ compiler *not* to do name mangling. Such a warning would look something like:

```
extern "C" {
void seed(int);
}
```

It is unlikely that you will encounter such difficulties when working in your IDE because it will have modernised all the libraries to suit C++. But if you get to use C libraries from elsewhere, you may have to watch out for this renaming problem.

Overloaded names

Another affect of this renaming is that a C++ compiler would see the following as distinct functions:

```
short abs(short);
long abs(long);
double abs(double);
```

after all it sees these as declaring functions called:

```
__abs__fs
__abs__fl
__abs__fd
```

This has the advantage that the programmer can use the single function name `abs()` (take the absolute value of ...) to describe the same operation for a variety of different data types. The name `abs()` has acquired multiple (though related) meanings, but the compiler is smart enough to keep them distinct. Such a name with multiple meanings is said to be *overloaded*.

It isn't hard for the compiler to keep track of what is going on. If it sees code like:

```
double x;
…
double v = -3.9 + x*(6.5 +3.9*x);

if(abs(v) > 0.00001)
    …
```

The compiler reasons that since `v` is a double the version of `abs()` defined for doubles (i.e. its function `__abs__fd`) is the one that should be called.

This approach, which is more or less unique to C++, is much more attractive than the mechanisms used in other languages. In most languages, function names must be distinct. So, if a programmer needs an absolute value function (just checking, you do know that that simply means you force any ± sign to be + don't you?) for several different data types, then several separately named functions must be defined. Thus, in the libraries used by C programs, you have to have:

```
double fabs(double);
long labs(long);
int abs(int);
```

Note: math library does NOT have overloaded definitions of abs()

Note that because of a desire for compatibility with C, most versions of the maths library do *not* have the overloaded `abs()` functions. You still have to use the specialized `fabs()` for doubles and so forth. C++ will let you use `abs()` with doubles – but it truncates the values to integers so you lose the fraction parts of the numbers.

Overloading for alternative interface

Function `abs()` illustrates the use of "overloading" so that a common name is used to describe equivalent operations on different data. There is a second common use for overloading. You will most likely encounter this use in something like a graphics package. There you might well find a set of overloaded `DrawRectangle()` functions:

```
void DrawRectangle(int left, int top, int width, int height);
void DrawRectangle(Rectangle r);
void DrawRectangle(Point topleft, int width, int height);
void DrawRectangle(Point topleft, Point dimension);
```

There is one basic operation of drawing a rectangle. Sometimes, the data available are in the form of four integers – it is then convenient to use the first overloaded function. Sometimes you may have a "Rectangle" data structure (a simple structure, see Chapter 17, that has constituent data elements defining top-left and bottom-right corners); in this case you want the second variant. Sometimes your data may be in the form of "Point" data structures (these contain an 'x' integer value, and a 'y' integer value) and either the third of fourth form of the function would be convenient. Here, overloading is used to provide multiple interfaces to a common operation.

Should you use overloading and define multiple functions with the same base name?

Probably not. Treat this as a "read only feature" of C++. You need to be able to read code that uses this feature, but while you are a beginner you shouldn't write such code. Stick to the KISS principle (remember Keep It Simple Stupid). Fancy styles like overloaded functions are more likely to get you into trouble than help you in your initial efforts.

10.6 HOW FUNCTIONS WORK

Section 4.8, on Algol, introduced the idea of a "stack" and showed how this could be used both to organize allocation of memory for the variables of a function, and to provide a mechanism for communicating data between a calling function and the function that it calls. It is worth looking again at these details.

The following code fragment will be used as an example:

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

Function definition

```
int GetIntegerInRange(int low, int high)
{
    int res;
    do {
           cout << "Enter an integer in the range " << low
                  << " ... " << high << " :"
           cin >> res;
    } while (! ((res >= low) && (res <= high)));
⇒   return res;
}

int main()
{
    const int LOWLIMIT = 1;
    int highval;
    cout << "Please enter high limit for parameter : ";
    cin >> highval;
    …
    int val;
    …
```

Function call

```
⇒   val = GetIntegerInRange(LOWLIMIT, highval);
    …
    …
}
```

Imagine that the program is running, the value 17 has been entered for highval, the function call (marked ⇒) has been made, and the function has run almost to completion and is about to return a value 9 for res (it is at the point marked ⇒). The state of the "stack" at this point in the calculation would be as shown in Figure 10.1.

The operating system will allocate some specific fixed area in memory for a program to use for its stack. Usually, the program starts using up this space starting at the highest address and filling down.

Each routine has a "frame" in the stack. The compiler will have sorted this out and worked out the number of bytes that a routine requires. Figure 10.1 shows most of the stack frame for the main program, and the frame for the called `GetIntegerInRange` function.

Local, "automatic", variables are allocated space on the stack. So the frame for `main()` will include the space allocated for variables such as `highval`; these spaces are where these data values are stored.

The compiler generates "calling" code that first copies the values of arguments onto the stack, as shown in Figure 10.1 where the data values 1 and 17 appear at the top of the "stack frame" for `GetIntegerInRange()`. The next slot in the stack would be left blank; this is where the result of the function is to be placed before the function returns.

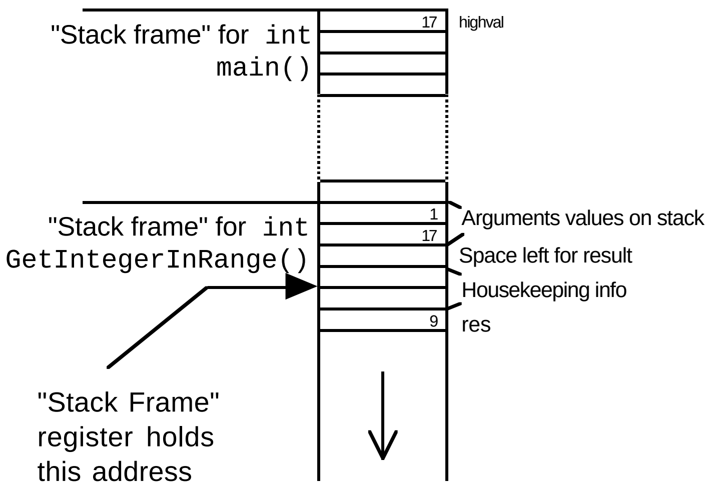

"Stack frame" for int main()
17
highval
"Stack frame" for int GetIntegerInRange()
1
17
Arguments values on stack
Space left for result
Housekeeping info
9
res
"Stack Frame" register holds this address

Figure 10.1 Stack during function call

Following the code that organizes the arguments, the compiler will generate a subroutine call instruction. When this gets executed at run-time, more "housekeeping information" gets saved on the stack. This housekeeping information will include the "return address" (the address of the instruction in the calling program that immediately follows the subroutine call). There may be other housekeeping information that gets saved as well.

The first instructions of a function will normally claim additional space on the stack for local variables. The function `GetIntegerInRange()` would have to claim sufficient space to store its `res` variable.

When a program is running, one of the cpu's address registers is used to hold the address of the current stack frame (the address will be that of some part of the housekeeping information). As well as changing the program counter, and saving the old value of the program counter on the stack, the subroutine call instruction will set this stack frame register (or "stack frame pointer sfp").

A function can use the address held in this stack frame pointer to find its local variables and arguments. The compiler determines the arrangement of stack frames and can generate address specifications that define the locations of variables by their offsets relative to the stack frame pointer. The `res` local variable of function `GetInteger-InRange()` might be located 8-bytes below the address in the stack frame pointer. The location for the return value might be 12-bytes above the stack frame pointer. The argument `high` might be 16-bytes above the stack frame pointer, and the argument `low` would then be 20-bytes above the stack frame pointer ("sfp"). The compiler can work all of this out and use such information when generating code for the function.

The instruction sequences generated for the function call and the function would be along the following lines:

Caller

```
// place arguments 1, and highval on stack
copy 1 onto stack
copy highval onto stack
subroutine call
copy result from stack into val
tidy up stack
```

Called function

```
  claim stack space for locals (i.e. res)
loop:
  code calling the output and input routines,
       result of input operation put
       in stack location for res
  load a CPU register from memory location 8 below sfp (res)
  load another CPU register from memory 20 above sfp (low)
  compare registers
  jump if less to loop // res is too small
  load other CPU from memory 16 above sfp (high)
  compare registers
  jump if greater to loop // res is too large
  // res in range
  store value from register into location 12 above sfp (slot
        for result)
    return from subroutine
```

As illustrated, a function has space in the stack for its local variables and its arguments (formal parameters). The spaces on the stack for the arguments are initialized by copying data from the actual parameters used in the call. But, they are quite separate. Apart from the fact that they get initialized with the values of the actual parameters, these argument variables are just like any other local variable. Argument values can be incremented, changed by assignment or whatever. Changes to such value parameters don't have any affect outside the function.

10.7 INLINE FUNCTIONS

A one-line function like:

```
double Average(double total, int number)
{
    return total/number;
}
```

really is quite reasonable. In most circumstances, the overheads involved in calling this function and getting a result returned will be negligible. The fact that the function is defined helps make code more intelligible.

But, sometimes it isn't appropriate to pay the overhead costs needed in calling a function even though the functional abstraction is desired because of the way this enhances intelligibility of code.

When might you want to avoid function call overheads?

Usually, this will be when you are doing something like coding the "inner loop" of some calculation where you may have a piece of code that you know will be executed hundreds of millions of times in a single run of a program, cutting out a few instructions might save the odd minute of cpu time in runs taking several hours. Another place where you might want to avoid the extra instructions needed for a function call would be in a piece of code that must be executed in the minimal time possible (e.g. you are writing the code of some real time system like the controller in your car which has to determine when to inflate the air-bags).

C++ has what it calls "inline" functions:

```
inline double Average(double total, int number)
{
    return total/number;
}
```

This presents the same abstraction as any other function definition. Here `Average()` is again defined as a function that takes a double and an integer as data and which returns their quotient as a double. Other functions can use this `Average()` function as a primitive operation and so avoid the details of the calculation appearing in the code of the caller.

The extra part is the `inline` keyword. This is a hint to the compiler. The programmer who defined `Average()` in this way is suggesting to the compiler that it should try to minimize the number of instructions used to calculate the value.

Consider a typical call to function `Average()`:

```
double girls_total, boys_total;
int num;
…
…
double ave = Average(girls_total + boys_total, num);
```

The instruction sequence normally generated for the call would be something like the following:

```
// Preparing arguments in calling routine
load double value girls_total into a floating point register
add double value boys_total to register
copy double result "onto stack"
copy integer value num "onto stack"
// make call, subroutine call instruction places more info
// in stack
subroutine call
// Calling code executed after return from routine
extract double result from stack and place
    in floating point register
clear up stack
```

The code for `Average()` would be something like:

```
copy double value total from place in stack
    to floating point register
copy integer value num into integer register
use "float" instruction to get floating point version of num in
    second floating point register
use floating point divide instruction to get quotient (left in
    first floating point register)
store quotient at appropriate point in stack
return from subroutine
```

The italicised instructions in those code sequences represent the "overhead" of a function call.

If you specify that the `Average()` be an `inline` function, the compiler will generate the following code:

```
load double value girls_total into a floating point register
```

```
add double value boys_total to register
load num into an integer register
use "float" instruction to get floating point version of num in
    second floating point register
use floating point divide instruction to get quotient
```

This is a rather extreme example in that the `inline` code does require significantly fewer instructions.

If things really are going to be time critical, you can request `inline` expansion for functions. You should only do this for simple functions. (Compilers interpretations of "simple" vary a little. Basically, you shouldn't declare a function `inline` if it has loops, or if it has complex selection constructs. Compilers are free to ignore your hints and compile the code using the normal subroutine call sequence; most compilers are polite enough to tell you that they ignored you `inline` request.)

Of course, if a compiler is to "expand" out the code for a function inline, then it must know what the code is! The definition of an `inline` function must appear in any file that uses it (either by being defined there, or by being defined in a file that gets #included). This definition must occur before the point where the function is first used.

10.8 A RECURSIVE FUNCTION

The idea of recursion was introduced in section 4.8. Recursion is a very powerful problem solving strategy. It is appropriate where a problem can be split into two subproblems, one of which is easy enough to be solved immediately and the second of which is simply the original problem but now with the data slightly simplified.

Recursive algorithms are common with problems that involve things like finding ones way through a maze. (Maze walk problem: moving from room where you are to exit room; solution, move to an adjacent room and solve maze walk for that room.) Surprisingly, many real programming problems can be transformed into such searches.

Section 4.8 illustrated recursion with a pseudo-code program using a recursive number printing function. How does recursion help this task? The problem of printing a number is broken into two parts, printing a digit (easy), and printing a (smaller) number (which is the original problem in simpler form). The recursive call must be done first (print the smaller number) then you can print the final digit. Here it is implemented in C++:

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

void PrintDigit(short d)
{
    switch(d) {
case 0 : cout << '0'; break;
case 1 : cout << '1'; break;
case 2 : cout << '2'; break;
case 3 : cout << '3'; break;
case 4 : cout << '4'; break;
case 5 : cout << '5'; break;
case 6 : cout << '6'; break;
case 7 : cout << '7'; break;
case 8 : cout << '8'; break;
case 9 : cout << '9'; break;
    }
}
```

Recursive function

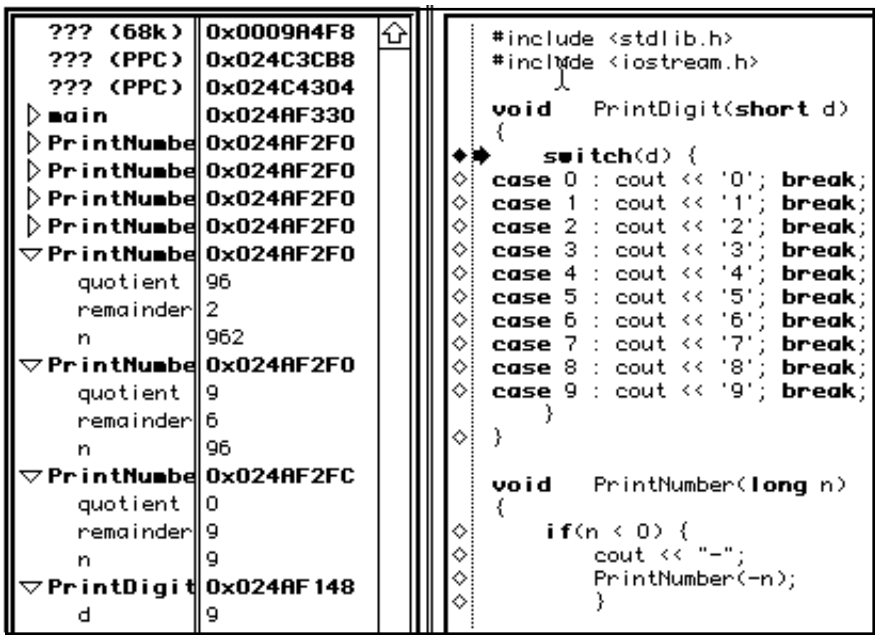
```
void        PrintNumber(long n)
{
    if(n < 0) {
            cout << "-";
            PrintNumber(-n);
            }
    else {
            long   quotient;
            short  remainder;
            quotient = n / 10;
            remainder = n % 10;
            if(quotient > 0)
                   PrintNumber(quotient);
            PrintDigit(remainder);
            }
}

int main()
{
  PrintNumber(9623589); cout << endl;
  PrintNumber(-33); cout << endl;
  PrintNumber(0); cout << endl;
  return 0;
}
```

Recursive call in function

Enter this program and compile with debugging options on. Before running the program, use the debugger environment in your IDE to set a breakpoint at the first line of the `PrintDigit()` function. Then run the program.

The program will stop and return control to the debugger just before it prints the first digit (the '9' for the number 9623589). Use the debugger controls to display the "stack backtrace" showing the sequence of function calls and the values of the arguments. Figure 10.2 illustrates the display obtained in the Symantec 8 environment.



```
??? (68k)    0x0009A4F8
??? (PPC)    0x024C3CB8
??? (PPC)    0x024C4304
▷ main       0x024AF330
▷ PrintNumbe 0x024AF2F0
▷ PrintNumbe 0x024AF2F0
▷ PrintNumbe 0x024AF2F0
▷ PrintNumbe 0x024AF2F0
▽ PrintNumbe 0x024AF2F0
    quotient 96
    remainder 2
    n        962
▽ PrintNumbe 0x024AF2F0
    quotient 9
    remainder 6
    n        96
▽ PrintNumbe 0x024AF2FC
    quotient 0
    remainder 9
    n        9
▽ PrintDigit 0x024AF148
    d        9
```

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

void   PrintDigit(short d)
{
    switch(d) {
case 0 : cout << '0'; break;
case 1 : cout << '1'; break;
case 2 : cout << '2'; break;
case 3 : cout << '3'; break;
case 4 : cout << '4'; break;
case 5 : cout << '5'; break;
case 6 : cout << '6'; break;
case 7 : cout << '7'; break;
case 8 : cout << '8'; break;
case 9 : cout << '9'; break;
    }
}

void   PrintNumber(long n)
{
    if(n < 0) {
        cout << "-";
        PrintNumber(-n);
        }
```

Figure 10.2 A "stack backtrace" for a recursive function.

The stack backtrace shows the sequence of calls: `PrintDigit()` was called from `PrintNumber()` which was called from `PrintNumber()` … which was called from `main()`. The display shows the contents of the stack frames (and, also, the return addresses) so you can see the local variables and arguments for each level of recursion.

You should find the integrated debugger a very powerful tool for getting to understand how functions are called; it is particularly useful when dealing with recursion.

10.9 EXAMPLES OF SIMPLE FUNCTIONS

10.9.1 GetIntegerInRange

A simple version of this function was introduced in 10.1:

```
int GetIntegerInRange(int low, int high)
{
    int res;
    do {
           cout << "Enter an integer in the range " << low
                  << " ... " << high << " :";
```

```
            cin >> res;
        } while (! ((res >= low) && (res <= high)));
        return res;
    }
```

Well, it is conceptually correct. But if you were really wanting to provide a robust function that could be used in any program you would have to do better.

What is wrong with the simple function?

First, what will happen if the calling code is mixed up with respect to the order of the arguments. ("*I need that GetIntegerInRange; I've got to give it the top and bottom limits – `count = GetIntegerInRange(100,5).`*") A user trying to run the program is going to find it hard to enter an acceptable value; the program will just keep nagging away for input and will keep rejecting everything that is entered.

There are other problems. Suppose that the function is called correctly (`GetInteger InRange(5,25)`) but the user mistypes a data entry, keying `y` instead of 7. In this case the `cin >> res` statement will fail. Error flags associated with `cin` will be set and the value 0 will be put in res. The function doesn't check the error flag. But it does check the value of `res`. Is `res` (0) between 5 and 25? No, so loop back, reprompt and try to read a value for `res`.

That is the point where it starts to be "fun". What data are waiting to be read in the input?

The next input data value is that letter `y` still there waiting to be read. The last read routine invoked by the `cin >>` operation didn't get rid of the `y`. The input routine simply saw a `y` that couldn't be part of a numeric input and left it. So the input operation fails again. Variable `res` is still 0. Consequently, the test on `res` being in range fails again. The function repeats the loop, prompting for an input value and attempting to read data.

Try running this version of the function with such input data (you'd best have first read your IDE manuals to find how to stop an errant program).

Fundamental law of programming

Remember Murphy's law: *If something can go wrong it will go wrong*. Murphy's law is the fundamental law of programming. (I don't know whether it is true, but I read somewhere that the infamous Mr. Murphy was the leader of one of IBM's first programming projects back in the early 1950s.)

You need to learn to program defensively. You need to learn to write code that is not quite as fragile as the simple `GetIntegerInRange()`.

Verify the data given to a function

One defensive tactic that you should always consider is verifying the data given to a function. If the data are wrong (e.g. `high < low`) then the function should not continue.

What should a function do under such circumstances?

In Part IV, we will explore the use of "exceptions". Exceptions are a mechanism that can sometimes be used to warn your caller of problems, leaving it to the caller to sort out a method of recovery. For the present, we will use the simpler approach of terminating the program with a warning message.

Using assert

A mechanism for checking data, and terminating a program when the data are bad, is so commonly needed that it has been packaged and provided in one of the standard libraries. Your IDE will have a library associated with the assert.h header file.

This library provides a "function" known as `assert()` (it may not be implemented as a true function, some systems use "macros" instead). Function `assert()` has to be given a "Boolean" value (i.e. an expression that evaluates to 0, False, or non-zero True). If the value given to `assert()` is False, the function will terminate the program after printing an error message with the general form "*Assertion failed in line … of function … in file …*".

We would do better starting the function `GetIntegerInRange()` as follows:

```
#include <stdlib.h>
#include <iostream.h>
#include <limits.h>
#include <assert.h>

long GetIntegerInRange(long low = LONG_MIN,
    long high = LONG_MAX)
{
    assert(low <= high);
```

The function now starts with the `assert()` check that `low` and `high` are appropriate. Note the #include <assert.h> needed to read in the declaration for `assert()`. (Another change: the function now has default values for low and high; the constants `LONG_MAX` etc are taken from limits.h.)

You should always consider using `assert()` to validate input to your functions. You can't always use it. For example, you might be writing a function that is supposed to search for a specific value in some table that contains data "arranged in ascending order"; checking that the entries in the table really were in ascending order just wouldn't be worth doing.

But, in this case and many similar cases, validating the arguments is easy. Dealing with input errors is a little more difficult.

Handling input errors

Some errors you can't deal with. If the input has gone "bad", there is nothing you can do. (This is going to be pretty rare; it will necessitate something like a hardware failure, e.g. unreadable file on a floppy disk, to make the input go bad.)

If all the data in a file has been read, and the function has been told to read some more, then once again there is nothing much that can be done.

So, for bad input, or end of file input, the function should simply print a warning message and terminate the program.

Clearing the failure flag

If an input operation has failed but the input stream, `cin`, is not bad and not at end of file, then you should try to tidy up and recover. The first thing that you must do is clear the fail flag on `cin`. If you don't do this, anything else you try to do to `cin` is going to be ignored. So, the first step toward recovery is:

```
cin.clear();
```

(The `clear()` operation is another of the special functions associated with streams like the various format setting `setf()` functions illustrated in 9.7.)

Next, you have to deal with any erroneous input still waiting to be read (like the letter y in the discussion above). These characters will be stored in some data area owned by the cin stream (the "input buffer"). They have to be removed.

We are assuming that input is interactive. (Not much point prompting a file for new data is there!) If the input is interactive, the user types some data and ends with a return. So we "know" that there is a return character coming.

We can get rid of any queued up characters in the input buffer if we tell the `cin` stream to "ignore" all characters up to the return (coded as '`\n`'). An input stream has an `ignore()` function, this takes an integer argument saying how many characters maximum to skip, and a second argument identifying the character that should mark the end of the sequence that is to be skipped. A call of the form:

```
cin.ignore(SHRT_MAX, '\n');
```

should jump over any characters (up to 30000) before the next return.

All of this checking and error handling has to be packaged in a loop that keeps nagging away at the user until some valid data value is entered. One possible coding for this loop is:

Prompting

Trying to read

Successful read? Return an OK value

Warn on out of range and loop again

Look for disasters

```
for(;;) {
    cout << "Enter a number in the range " << low
           << " to " << high << " : ";
    int val;
    cin >> val;

    if(cin.good())
           if((val >= low) && (val <= high)) return val;
           else {
                  cout << val << " is not in required range."
                         << endl;
                  continue;
           }

    if(cin.bad()) {
           cout << "Input has gone bad, can't read data; "
                         "quitting." << endl;
                  exit(1);
           }

    if(cin.eof()) {
           cout << "Have unexpected end of file; quitting."
                         << endl;
           exit(1);
           }
```

```
        cin.clear();
        cout << "Bad data in input stream, flushing buffer."
                      << endl;

        cin.ignore(SHRT_MAX, '\n');
        }
```

Must just be bad data in input, clear up

This is an example of a "forever" loop. The program is stuck in this loop until either an acceptable value is entered or something goes wrong causing the program to terminate. The loop control statement:

```
    for(;;)
```

has nothing in the termination test part (i.e. nothing terminates this loop).

After the read (`cin >> val`) , the first check is whether `cin` is "good". It will be good if it was able to read an integer value and store this value in variable `val`. Of course, the value entered might have been an integer, but not one in the acceptable range. So, that had better be checked. Note the structure of the nested ifs here. *If* `cin` is good *then if* the number is in range return *else* complain and loop.

Here the return statement is in the middle of the body of the loop. This style will offend many purists (who believe that you should only return from the end point of the function). However, it fits logically with the sequence of tests and the code in this form is actually less complex than most variants.

Note also the use of `continue`. This control statement was introduced in section 7.8 but this is the first example where it was used. If the user has entered an out of range value, we need to print a warning and then repeat the process of prompting for and reading data. So, after the warning we want simply to "continue" with the next iteration of the for loop.

The remaining sections of the loop deal with the various input problems already noted and the attempt to clean up if inappropriate data have been entered.

Although it won't survive all forms of misuse, this version of `GetIntegerInRange()` is a little more robust. If you are writing functions that you hope to use in many programs, or which you hope to share with coworkers, you should try to toughen them like this.

10.9.2 Newton's square root algorithm as a function

This second example is a lot simpler! It is merely the program for finding a square root (7.2.2) recoded to use a function:

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

```
double NewtRoot(double x)
{
    double r;
    const double SMALLFRACTION = 1.0E-8;
    assert(x > 0.0);

    r = x / 2.0;
    while(fabs(x - r*r) > SMALLFRACTION*x) {
           cout << r << endl;
           r = 0.5 *(x / r + r);
           }

    return r;
}

int main()
{
    double x;
    cout << "Enter number : ";
    cin >> x;

    cout << "Number was : " << x << ", root is "
                  << NewtRoot(x) << endl;
    return EXIT_SUCCESS;
}
```

Sanity check on argument value

Once again, it is worth putting in some checking code in the function. The `assert()` makes certain that we haven't been asked to find the root of a negative number.

10.10 THE RAND() FUNCTION

The stdlib library includes a function called `rand()`. The documentation for stdlib describes `rand()` as a "simple random number generator". It is supposed to return a random number in the range 0 to 32767.

What is your idea of a "random number" generator? Probably you will think of something like one of those devices used to pick winners in lotteries. A typical device for picking lottery winners has a large drum containing balls with the numbers 1 to 99 (or maybe some smaller range); the drum is rotated to mix up the balls and then some mechanical arrangement draws six or so balls. The order in which the balls are drawn doesn't matter; you win the lottery if you had submitted a lottery coupon on which you had chosen the same numbers as were later drawn. The numbers drawn from such a device are pretty random. You would have won this weeks lottery if you had picked the numbers 11, 14, 23, 31, 35, and 39; last week you wished that you had picked 4, 7, 17, 23, 24, and 42.

The `rand()` function don't seem to work quite that way. For example, the program:

```
int main()
{
    for(int i = 0;i<20;i++)
           cout << rand() << endl;
    return 0;
}
```

run on my Macintosh produces the numbers:

```
16838
 5758
10113
17515
31051
 5627
23010
    …
    …
25137
25566
```

Exactly the same numbers are produced if I run the program on a Sun workstation (though I do get a different sequence on a PowerPC).

"Seeding" the random number generator

There is an auxiliary function that comes with `rand()`. This is `srand()`. The function `srand()` is used to "seed the random number generator". Function `srand()` takes an integer as an argument. Calling `srand()` with the argument 5 before the loop generating the "random numbers":

```
    srand(5);
    for(int i = 0;i<20;i++)
           cout << rand() << endl;
```

and (on a Macintosh with Symantec 7.06) you will get the sequence

```
18655
 8457
10616
31877
    …
    …
  985
32068
24418
```

Try another value for the "seed" and you will get a different sequence.

For a given seed, the same numbers turn up every time.

Arbitrary numbers? Yes. Random numbers? No. It is the same sequence of arbitrary numbers every time (and since most versions of stdlib are identical, it is the same sequence on most computers). A lottery company wouldn't be wise to try to computerize their selection process by using `rand()` like this. Smart punters would soon learn which were the winning numbers because these would be the same every week.

The term "random number generator" is misleading. The rand function has a more correct description; "*rand() uses a multiplicative congruential random-number generator with period 2^32 that returns successive pseudo-random numbers in the range from 0 to (2^15)-1*".

What does that mathematical mumbo-jumbo mean? It means that these "pseudo-random numbers" are generated by an algorithmic (non-random) mathematical process but that as a set they share some mathematical properties with genuinely randomly picked numbers in the same range.

Some of these properties are:

- The counts for the frequency of each of the possible last digits, obtained by taking the number modulo 10 (i.e. `rand() % 10`), will be approximately equal.
- If you look at pairs of successive pseudo-random numbers, then the number times that the second is larger will be approximately equal to the number of times that the first is larger.
- You will get about the same number of even numbers as odd numbers.
- If you consider a group of three successive random numbers, there is a one in eight chance that all will be odd, a one in eight chance that all are even, a three in eight chance that you will get two odds and one even. These are the same (random) chances that you would get if you were tossing three coins and looking for three heads etc.

Although the generated numbers are not random, the statistical properties that they share with genuinely random numbers can be exploited.

If you've lost your dice and want a program to roll three dice for some game, then the following should be (almost) adequate:

```
int main()
{
    // Roll three dice for game
    char ch;
    …
    do {
           for(int i=0; i < 3; i++) {
                  int roll = rand() % 6;
                  // roll now in range 0..5
                  roll++;  // prefer 1 to 6
                  cout << roll << " ";
                  }
           cout << endl;
```

```
                cout << "Roll again? ";
                cin >> ch;
        } while ((ch == 'Y') || (ch == 'y'));
        return 0;
}
```

About one time in two hundred you will get three sixes; you have the same chance of getting three twos. So the program does approximate the rolling of three fair dice.

But it is a bit boring. Every time you play your game using the program you get the same dice rolls and so you get the same sequence of moves.

```
3 5 4
Roll again? y
2 2 6
Roll again? y
1 4 1
Roll again? y
1 2 6
Roll again? n
```

You can extend the program so that each time you run it you get a different sequence of arbitrary numbers. You simply have to arrange to seed the random number generator with some "random" value that will be different for each run of the program.

Finding a seed

Of course, normal usage requires everything to be the same every time you run the same program on the same data. So you can't have anything in your program that provides this random seed; but you can always ask the operating system (OS) about other things going on in the machine. Some of the values you can get through such queries are likely to be different on different runs of a program.

Most commonly, program's ask the OS for the current time of day, or for some related time measure like the number of seconds that the machine has been left switched on. These calls are OS dependent (i.e. you will have to search your IDE documentation for details of the name of the timing function and its library). Just for example, I'll assume that your system has a function `TickCounter()` that returns the number of seconds that the machine has been switched on. The value returned by this function call can be used to initialize the random number generator:

```
int main()
{
        // Roll three dice for game
        char ch;
        srand(TickCounter());
        do {
                for(int i=0; i < 3; i++) {
                        int roll = rand() % 6;
                …
                …
```

With this extension to "randomize" the number sequence, you have a program that will generate a different sequence of dice rolls every time you run it.

If "`TickCounter()`" is random enough to seed the generator, why not just use it to determine the roll of the dice:

```
do {
    for(int i=0; i < 3; i++) {
           int roll = TickCounter() % 6;
           roll++;  // prefer 1 to 6
```

Of course this doesn't work! The three calls to `TickCounter()` are likely to get exactly the same value, if not identical then the values will be in increasing order. A single call to `TickCounter()` has pretty much random result but inherently there is no randomness in successive calls; the time value returned is monotonically increasing.

When developing and testing a program, you will often chose to pass a constant value in the call to `srand()`. That way you get the same sequence of numbers every time. This can help debugging because if you find something going wrong in your program then, after adding trace statements, you can rerun the program on identical data. When the program has been developed, you arrange either to get the user to enter the seed or you get a seed value from a system call like `TickCounter()`.

10.11 EXAMPLES

10.11.1 π-Canon

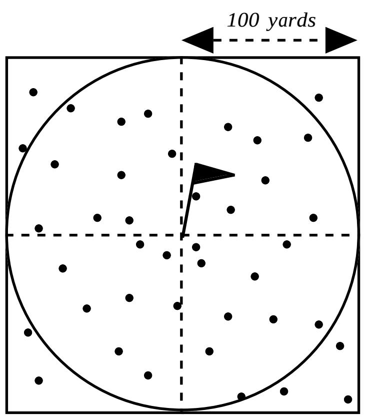
A mathematics professor once approached a colonel of his state's National Guard asking for the Guards assistance in a new approach to estimating the value of π.

The professor reckoned that if a cannon was fired at a distant target flag, the balls would fall in a random pattern around the target. After a large number of balls had been fired, he would be able to get an estimate of π by noting where the balls had fallen. He used the diagram shown in Figure 10.3, to explain his approach to estimating π:

The colonel refused. He reckoned that his gunners would put every shot within 50 paces of the target flag and that this would lead to the result π = 4 which everyone knew was wrong.

Write a computer program that will allow the maths. professor to simulate his experiment.

The program should let the professor specify the number of shots fired. It should simulate the experiment printing estimates of π at regular intervals along with a final estimate when all ammunition has been expended.



Area of square *4 x 100 square yards*
Area of circle *π x 100 square yards*

If the shot fall randomly in the target area, then the numbers falling within the circle and the square should be proportionate to their areas..

Hence can estimate π from these ratios:

```
Number in circle     Area of circle    π
---------------- =  --------------- = ---
Number in square     Area of square    4
```

Fot the shot pattern shown:

```
    33       π
    --- =   ---       so π ≈   3.219
    41       4
```

A better estimate would be obtained when additional shot had been fired.

Figure 10.3 Estimating π.

Design

The design process appropriate to problems of this complexity is known as "top-down functional decomposition". We know the overall function (the "top function" of the program), it is to run the simulation and calculate π. But obviously, that involves a lot of work. It is far too much work to be encoded in a single main function. So, we proceed by iterating through the problem repeatedly, at each iteration we try to refine our model and abstract out distinct subproblems that can be analyzed separately.

The problem obviously involves random numbers. For each "shot" we will need two random numbers, one to give the "x" position relative to the flag, the second to give the "y" position. We can arrange it so that all shots fall in the square by selecting the range for the random numbers (just have to tell the professor that we are ignoring bad shots that fall too far from the target). When we have the x, y coordinates of where the shot is supposed to have landed, we can calculate the distance from the flag. If this distance is less than 100.0, the shot adds to the count of those falling in the circle. We can get an estimate of π whenever we need by just doing the calculation using our counts of total shots fired and shots landing in the circle.

First iteration through design

Based on the problem description, we can identify a rough framework for the program:

```
//initialization phase
    get professor to say how much ammunition he has
    get some seed for the random number generator
loop until ammunition exhausted
    increment count of shots fired
```

```
        get x, y coordinates
        test position and, if appropriate,
               update count of shots in circle
        maybe print next estimate of π (not for every shot,
                      What is frequency of these reports?)
    print final estimate of π
```

Some "functions" appear immediately in this description, for example, "estimate π", "test position", "get coordinate". All of these tasks involve several calculation steps that clearly belong together, can be thought about in isolation, and should be therefore separate functions. Even something like "initialization phase" could probably be a function because it groups related operations.

Second iteration through design

We should now start to identify some of these functions:

```
    get ammunition
        prompts professor for inputs like amount of ammunition,
               and seed for random number generator
        use seed to seed the random number generator
        return amount of ammunition

    π-estimator function
        must be given total shot count and circle shot count
        calculates π from professor's formula
        returns π estimate

    get-coordinate function
        use random number generator, converts 0..32767 result
               into a value in range -100.0 to +100.0 inclusive
        returns calculated value

    print π-estimate
        produce some reasonably formatted output,
        probably should show π and the number of shots used to
               get this estimate, so both these values will
               be needed as arguments

    test in circle
        given an x and y coordinate pair test whether they
               represent a point within 100.0 of origin
        return 0 (false) 1 (true) result

    main function
        get ammunition
        loop until ammunition exhausted (How checked?)
               increment count of shots fired
                      (Where did this get initialized?)
               x = get-coordinate
               y = get-coordinateget x,y coordinates
               if test in circle (x, y)
                      increment circle count
```

```
              maybe print next estimate of π (not for every shot,
                     What is frequency of these reports?)
       call print π estimate
```

There are some queries. These represent issues that have to be resolved as we proceed into more detailed design.

Third iteration through design process

This problem is still fairly simple so we have already reached the point where we can start thinking about coding. At this point, we compose function prototypes for each of the functions. These identify the data values that must be given to the function and the type of any result:

Composing the function prototypes

```
   long       GetAmmunition(void);

   double     PiEstimate(long squareshot, long circleshot);

   double     GetCoordinate(void);

   void       PrintPIEstimate(double pi, long shotsused);

   int        TestInCircle(double x, double y);

   int        main();
```

Final stage design and implementation

Next, we have to consider the design of each individual function. The process here is identical to that in Part II where we designed the implementations of the little programs. We have to expand out our english text outline for the function into a sequence of variable definitions, assignment statements, loop statements, selection statements etc.

Most of the functions that we need here are straightforward:

```
   long GetAmmunition(void)
   {
       long   munition;
       long   seed;
       cout << "Please Professor, how many shots should we fire?";
       // Read shots, maybe we should use that GetIntegerInRange
       // to make certain that the prof enters a sensible value
       // Risk it, hope he can type a number
       cin >> munition;
       // Need seed for random number generator,
       // Get prof to enter it (that way he has chance of
       // getting same results twice).  Don't confuse him
       // with random number stuff, just get a number.
       cout << "Please Professor, which gunner should lay the
   gun?"
```

```
                    << endl;
    cout << "Please enter a gunner identifier number 1..1000
:";
    cin >> seed;
    srand(seed);
    return munition;
}

int TestInCircle(double x, double y)
{
    // Use the sqrt() function from math.h
    // rather than NewtRoot()!
    // Remember to #include <math.h>
    double distance = sqrt(x*x + y*y);
    return distance < 100.0;
}

double PiEstimate(long squareshot, long circleshot)
{
    return 4.0*double(circleshot)/squareshot;
}
```

The `GetCoordinate()` function is slightly trickier; we want a range -100.0 to +100.0. The following should suffice:

```
double GetCoordinate(void)
{
    int    n = rand();
    // Take n modulo 201, i.e. remainder on dividing by 201
    // That should be a number in range 0 to 200
    // deduct 100
    n = n % 201;
    return double(n-100);
}
```

The `main()` function is where we have to consider what data values are needed for the program as a whole. We appear to need the following data elements:

```
long counter for total shots
long counter for shots in circle
long counter for available munitions

double for π estimate
```

within the loop part of `main()` we also require:

```
double x coordinate
double y coordinate
```

We have to decide how often to generate intermediate printouts of π estimates. If professor is firing 1000 shots, he would probably like a report after every 100; if he can only afford to pay for 100 shots, he would probably like a report after every 10. So it seems we can base report frequency on total of shots to be fired. We will need a couple of variables to record this information; one to say frequency, the other to note how many shots fired since last report. So, `main()` will need two more variables.

```
long counter for report frequency
long counter for shots since last report
```

We can now complete a draft for `main()`:

```
int main()
{
    long   fired = 0, ontarget = 0, ammunition;
    double pi = 4.0; // Use the colonel's estimate!
    long   freq, count;

    ammunition = GetAmmunition();

    freq = ammunition / 10;
    count = 0;

    for(; ammunition > 0; ammunition--) {
           double x, y;
           fired++;
           x = GetCoordinate();
           y = GetCoordinate();
           if(TestInCircle(x, y))
                  ontarget++;

           count++;
           if(count == freq) {
                  // Time for another estimate
                  pi = PiEstimate(fired, ontarget);
                  PrintPIEstimate(pi, fired);
                  // reset count to start over
                  count = 0;
                  }
           }

    cout << "We've used all the ammunition." << endl;
    cout << "Our final result:" << endl;
    pi = PiEstimate(fired, ontarget);
    PrintPIEstimate(pi, fired);
    return 0;
}
```

Complete the implementation of the program (you need to provide `PrintPIEstimate()`) and run it on your system.

The results I got suggest that the prof. would have to apply for, and win, a major research grant before he could afford sufficient ammunition to get an accurate result:

```
Estimate for pi, based on results of 1000 shots, is 3.24800000
Estimate for pi, based on results of 10000 shots, is 3.12160000
```

10.11.2 Function plotter

Problem

When doing introductory engineering and science subjects, you often need plots of functions, functions such as

```
f(x) = a*cos(b*x + c)*exp(-d*x)
```

This function represents an exponentially decaying cosine wave. Such functions turn up in things like monitoring a radio signal that starts with a pulse of energy and then decays away. The parameters a, b, c, and d determine the particular characteristics of the system under study.

It is sometimes useful to be able to get rough plots that just indicate the forms of such functions. These plots can be produced using standard iostream functions to print characters. The width of the screen (or of a printer page) can be used to represent the function range; successive lines of output display the function at successive increasing values of x. Figure 10.4 illustrates such a plot.

Specification

Write a program that will plot function values using character based output following the style illustrated in Figure 10.4. The program is to plot values of a fixed function f(x) (this function can only be changed by editing and recompilation). The program is to take as input a parameter R that defines the range of the function, the page (screen) width is to be used to represent the range -R to +R. Values of the function are to be plotted for x values, starting at x = 0.0, up to some maximum value X. The maximum, and the increments for the x-axis, are to be entered by the user when the program runs.

Design

"Plotting a function" – the "top function" of this program is again simple and serves as the starting point for breaking the problem down.

```
------------------------------------+-----------------------------------
                                    |                              *
                                    |                      *
                                    |             *
                                    |    *
                                *   |
                        *           |
                   *                |
               *                    |
              *                     |
               *                    |
                 *                  |
                     *              |
                         *          |
                              *     |
                                   *|
                                    |   *
                                    |      *
                                    |         *
                                    |          *
                                    |          *
                                    |         *
                                    |       *
                                    |     *
                                    |  *
                                    |*
                                  * |
                                *   |
                               *    |
                              *     |
                             *      |
                              *     |
                              *     |
                               *    |
                                 *  |
```

Figure 10.4 Character based plotting of a function.

First iteration through design

A rough framework for the program would be:

```
//initialization phase
    range data, maximum x value, and x increment
    initialize current x to 0.0
//
    Plot axis
// main loop
loop until x exceeds maximum
    work out function value
    plot current value (and x-axis marker)
    increment x
```

The obvious functions are: "work out function value", "plot current value", and "plot axis".

Second iteration through design

These functions can start to be elaborated:

```
fun
    Given current x value as data, works out value of
expression
    returns this value as a double
    (expression will probably contain parameters like a, b,
...;
    but these can be defined as constants as they aren't to be
    changed by user input)

plotaxis
    prints a horizontal rule (made from '-' characters) with a
           centre marker representing the zero point of the
           range
    probably should also print labels, -R and +R, above the
rule
           so the value for the range should be given as an
           argument
    (? width of screen ?  will need this defined as a constant
           that could be adjusted later to suit different
           screen/page sizes)

plot function
    given a value for f(x), and details of the range,
           this function must first work out where on line
           a mark should appear
    a line should then be output made up from spaces, a |
           character to mark the axis, and a * in the
           appropriate position
```

Functions `main()`, `plotaxis()`, and `fun()` look fairly simple. You wouldn't need to iterate further through the design process for these; just have to specify the prototypes and then start on coding the function bodies.

Function prototypes

```
int main();
void PlotAxis(double range);
double fun(double x);
void PlotFunctionValue(double value, double range);
```

However, the description of `PlotFunctionValue()` seems a bit complex. So, it is too soon to start thinking about coding. The various steps in `PlotFunctionValue()` must be made more explicit.

Third iteration through design process

The width of the screen is meant to represent the range -R to R with R the user specified range. A value v for f(x) has to be scaled; the following formula defines a suitable scaling:

```
position relative to centre = v/R*half_width
```

Example, screen width 60 places, range represented -2.0 to +2.0, value to be plotted 0.8:

```
position relative to centre      = 0.8/2.0*30
                                 = 12 places right of centre
```

Example, screen width 60 places, range represented -2.0 to +2.0, value to be plotted -1.5:

```
position relative to centre      = -1.5/2.0*30
                                 = -22.5
                                 i.e. 22.5 places left of centre
```

A C++ statement defining the actual character position on the line is:

```
int where = floor((value/range*HALF) + 0.5) + HALF;
```

The constant `HALF` represents half the screen width. Its value will depend on the environment used.

Since we now know how to find where to plot the '*' character we need only work out how to do the correct character output.

There are a number of special cases that may occur:

- The value of `where` is outside of the range `1..WIDTH` (where `WIDTH` is the screen width). This will happen if the user specified range is insufficient. If the value is out of range, we should simply print the '|' vertical bar character marking the x-axis itself.

- `where's` value is `HALF`, i.e. the function value was 0.0. We don't print the usual '|' vertical bar axis marker, we simply plot the '*' in the middle of the page.

- `where` is less than `HALF`. We need to output the right number of spaces, print the '*', output more spaces until half way across the page, print the axis marker.

- `where` is greater than `HALF`. We need to output spaces until half way across page, print the axis marker, then output additional spaces until at point where can print the '*' symbol.

There are smarter ways to deal with the printout, but breaking it down into these possibilities is about the simplest.

The problem has now been simplified to a point where coding can start.

Implementation

There isn't much to the coding of this example and the complete code is given below.

Conditional compilation directives

The code is used to provide a first illustration of the idea of "conditional compilation". You can include in the text of your program various "directives" that instruct the compiler to take special actions, such as including or omitting a particular piece of code.

Conditional compilation for debugging and customisation

There are two common reasons for having conditionally compiled code. The first is that you want extra "tracing" code in your program while you are developing and debugging it. In this case, you will have conditionally compiled sections that have extra output statements ("trace" output); examples will be shown later. This code provides a rather simple instance of the second situation where conditional compilation is required; we have a program that we want run on different platforms – Symantec, Borland-compiled DOS application, and Borland-compiled EasyWin application. These different environments require slightly different code; but rather than have three versions of the code, we can have a single file with conditionally compiled customizations.

The differences are very small. The default font in an EasyWin window is rather large so, when measured in terms of characters, the window is smaller. If you compile and run the program as a DOS-standard program it runs, and exits back to the Borland IDE environment so quickly that you can't see the graph it has drawn! The DOS version needs an extra statement telling it to slow down.

The directives for conditional compilation, like most other compiler directives, occur on lines starting with a # character. (The only other directives we've used so far has been the #include directive that loads a header file, and the #define directive that defines a constant.) Here we use the #if, #elif ("else if"), #else, and #endif directives and use #define in a slightly different way. These directives are highlighted in bold in the code shown below; the italicised statements are either included or omitted according to the directives.

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

#define SYMANTEC

#if defined(SYMANTEC)
const int WIDTH = 71;
#elif defined(EASYWIN)
const int WIDTH = 61;
#else
const int WIDTH = 81;
#endif

#if defined(DOS)
#include <dos.h>
#endif
```

The line #define SYMANTEC "defines" SYMANTEC as a compile time symbol (it doesn't get a value, it just exists); this line is specifying that this version of the code is to be compiled for the Symantec environment. The line would be changed to #define EASYWIN or #define DOS if we were compiling for one of the other platforms.

The #if defined() … #elif defined() … #else … #endif statements select the WIDTH value to be used depending on the environment. The other #if … #endif adds an extra header file if we are compiling for DOS. The DOS version uses the standard sleep() function to delay return from the program. Function sleep() is defined in most environments; it takes an integer argument specifying a delay period for a program. Although it is widely available, it is declared in different header files and is included in different libraries on different systems.

```
const int HALF = 1 + (WIDTH/2);

const double a = 1.0;
const double b = 1.0;
const double c = 0.5;
const double d = 0.2;
```

The definition of constant HALF illustrates compile time arithmetic; the compiler will have a value for WIDTH and so can work out the value for HALF. The other constants define the parameters that affect the function that is being plotted.

```
void PlotAxis(double range)
{
    cout << -range;
    for(int i=0;i<WIDTH-10;i++)
           cout << ' ';
    cout << range << endl;
    for(i=1;i<HALF;i++)
           cout << '-';
    cout << '+';
    for(i=HALF+1;i<=WIDTH;i++)
           cout << '-';
    cout << endl;
}
```

The code for PlotAxis() is simple; it prints the range values, using the loop printing spaces to position the second output near the right of the window. Note that variable i is defined in the first for statement; as previously explained, its scope extends to the end of the block where it is defined. So, the other two loops can also use i as their control variables. (Each of the cases is coded as a separate compound statement enclosing a for loop. Each for loop defines int i. Each of these 'i's has scope that is limited to just its own compound statement.)

```
void PlotFunctionValue(double value, double range)
{
```

Determine position on line

```
    int where = floor(value/range*HALF) + HALF;
```

Special case 1, out of range

```
    if((where < 1) || (where > WIDTH)) {
           for(int i=1; i<HALF;i++)
                  cout<< ' ';
           cout << '|' << endl;
           return;
           }
```

Special case 2, on axis

```
    if(where == HALF) {
           for(int i=1; i<HALF;i++)
                  cout<< ' ';
           cout << '*' << endl;
           return;
           }
```

Special case 3, left of axis

```
    if(where < HALF) {
           for(int i=1; i<where;i++)
                  cout<< ' ';
           cout << '*';
           for(i=where+1;i<HALF;i++)
                  cout << ' ';
           cout << '|' << endl;
           return;
           }
```

Last case, right of axis

```
    for(int i=1; i<HALF;i++)
           cout<< ' ';
    cout << '|';
    for(i=HALF+1;i<where;i++)
           cout << ' ';
    cout << '*';
    cout << endl;

    return;
}
```

Your instructor may dislike the coding style used in this version of `PlotFunctionValue()`; multiple returns are often considered poor style. The usual recommended style would be as follows:

```
void PlotFunctionValue(double value, double range)
{
    int where = floor(value/range*HALF) + HALF;

    if((where < 1) || (where > WIDTH)) {
           for(int i=0; i<HALF;i++)
                  cout<< ' ';
           cout << '|' << endl;
           }
    else
```

```
        if(where == HALF) {
                …
                }
        else
        if(where < HALF) {
                …
                }
        else {
                …
                }
        return;
    }
```

Personally, I find long sequences of `if … else if … else if … else` constructs less easy to follow. When reading these you have to remember too much. I prefer the first style which regards each case as a filter that deals with some subset of the data and finishes. You need to be familiar with both styles; use the coding style that your instructor prefers.

```
    double fun(double x)
    {
        double val;
        val = a*cos(b*x+c)*exp(-d*x);
        return val;
    }

    int main()
    {
        double r;
        double xinc;
        double xmax;

        cout << "Range on 'y-axis' is to be -R to R" << endl;
        cout << "Please enter value for R : ";
        cin >> r;

        cout << "Range on 'x-axis' is 0 to X" << endl;
        cout << "Please enter value for X : ";
        cin >> xmax;

        cout << "Please enter increment for X-axis : ";
        cin >> xinc;

        PlotAxis(r);

        double x = 0.0;
        while(x<=xmax) {
                double f = fun(x);
                PlotFunctionValue(f, r);
                x += xinc;
        }

#if defined(DOS)
    // delay return from DOS to Borland IDE for
    // a minute so user can admire plotted function
    sleep(60);
#endif

    return 0;

}
```

There are no checks on input values, no assertions. Maybe there should be. But there isn't that much that can go seriously wrong here. The only problem case would be the user specifying a range of zero; that would cause an error when the function value is scaled. You can't put any limit on the range value; it doesn't matter if the range is given as negative (that flips the horizontal axis but does no real harm). A negative limit for `xmax` does no harm, nothing gets plotted but no harm done.

EXERCISES

1 Write a program that will roll the dice for a "Dungeons and Dragons" (DD) game.

DD games typically use several dice for each roll, anything from 1 to 5. There are many different DD dice 1…4, 1…6, 1…8, 1…10, 1…12, 1…20. All the dice used for a particular roll will be of the same type. So, on one roll you may need three six sided dice; on the next you might need one four sided dice.

The program is to ask how many dice are to be rolled and the dice type. It should print details of the points on the individual dice as well as a total score.

2 Write a "Chinese fortune cookie" program.

The program is to loop printing a cookie message (a joke, a proverb, cliche, adage). After each message, the program is to ask the user for a Y(es) or N(o) response that will determine whether an additional cookie is required.

Use functions to select and print a random cookie message, and to get the Y/N response.

You can start with the standard fortunes – "A new love will come into your life.", "Stick with your wife.", "Too much MSG, stomach pains tomorrow.", "This cookie was poisoned.". If you need others, buy a packet of cookies and plagiarise their ideas.