**Recursion**

The existence of functions makes possible a programming technique called *recursion*.

Recursion involves a function calling itself. This sounds rather improbable, and indeed a function

calling itself is often a bug. However, when used correctly this technique can be surprisingly

powerful.

Recursion is much easier to understand with an example than with lengthy explanations, so

let’s apply it to a program we’ve seen before: the FACTOR program of Chapter 3, “Loops and

Decisions.” That program used a for loop to calculate the factorial of a number. (See that

example for an explanation of factorials.) Our new program, FACTOR2, uses recursion instead of

a loop.

//factor2.cpp

//calculates factorials using recursion

#include <iostream>

using namespace std;

unsigned long factfunc(unsigned long); //declaration

int main()

{

int n; //number entered by user

unsigned long fact; //factorial

cout << “Enter an integer: “;

cin >> n;

fact = factfunc(n);

cout << “Factorial of “ << n << “ is “ << fact << endl;

return 0;

}

//-------------------------------------------------------------

// factfunc()

// calls itself to calculate factorials

unsigned long factfunc(unsigned long n)

{

if(n > 1)

return n \* factfunc(n-1); //self call

else

return 1;

}

The output of this program is the same as the FACTOR program in Chapter 3.

The main() part of FACTOR2 looks reasonable: it calls a function, factfunc(), with an argument

that is a number entered by the user. This function then returns the factorial of that number

to main().

The function factfunc() is another story. What’s it doing? If n is greater than 1, the function

calls itself. Notice that when it does this it uses an argument one less than the argument it was

called with. Suppose it was called from main() with an argument of 5. It will call a second

version of itself with an argument of 4. Then this function will call a third version with an

argument of 3, and so on.

Notice that each version of the function stores its own value of n while it’s busy calling another

version of itself.

After factfunc() calls itself four times, the fifth version of the function is called with an argument

of 1. It discovers this with the if statement, and instead of calling itself, as previous versions

have, it returns 1 to the fourth version. The fourth version has stored a value of 2, so it

multiplies the stored 2 by the returned 1, and returns 2 to the third version. The third version

has stored 3, so it multiplies 3 by the returned 2, and returns 6 to the second version. The

second version has stored 4, so it multiplies this by the returned 6 and returns 24 to the first

version. The first version has stored 5, so it multiplies this by the returned 24 and returns 120

to main().

Thus in this example we have five function calls followed by five function returns. Here’s a

summary of this process:

*Argument or*

*Version Action Return Value*

1 Call 5

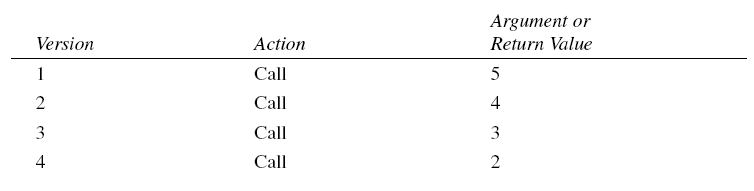
2 Call 4

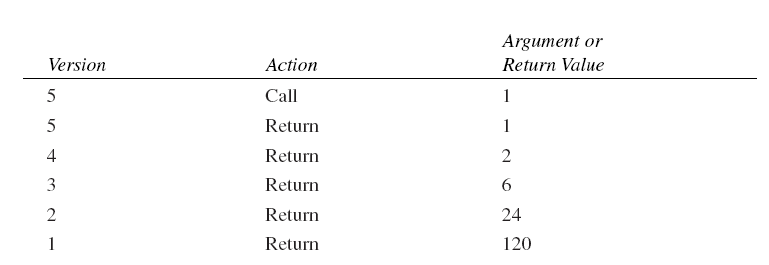
3 Call 3

4 Call 2

Chapter 5

194





Every recursive function must be provided with a way to end the recursion. Otherwise it will

call itself forever and crash the program. The if statement in factfunc() plays this role, terminating

the recursion when n is 1.

Is it true that many versions of a recursive function are stored in memory while it’s calling

itself? Not really. Each version’s variables are stored, but there’s only one copy of the function’s

code. Even so, a deeply-nested recursion can create a great many stored variables, which

can pose a problem to the system if it doesn’t have enough space for them.

**Inline Functions**

We mentioned that functions save memory space because all the calls to the function cause the

same code to be executed; the function body need not be duplicated in memory. When the

compiler sees a function call, it normally generates a jump to the function. At the end of the

function it jumps back to the instruction following the call, as shown in Figure 5.1 earlier in

this chapter.

While this sequence of events may save memory space, it takes some extra time. There must

be an instruction for the jump to the function (actually the assembly-language instruction

CALL or something similar), instructions for saving registers, instructions for pushing arguments

onto the stack in the calling program and removing them from the stack in the function

(if there are arguments), instructions for restoring registers, and an instruction to return to the

calling program. The return value (if any) must also be dealt with. All these instructions slow

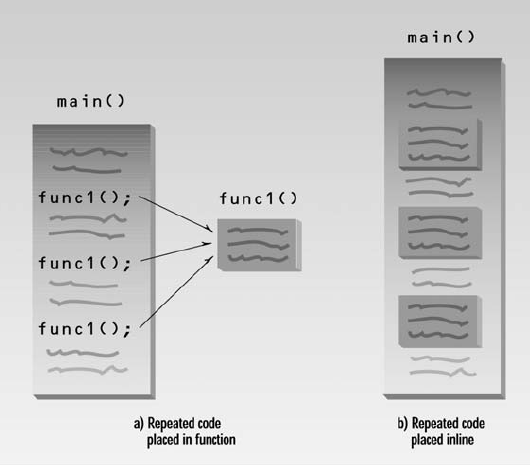
down the program.

To save execution time in short functions, you may elect to put the code in the function body

directly inline with the code in the calling program. That is, each time there’s a function call in

the source file, the actual code from the function is inserted, instead of a jump to the function.

The difference between a function and inline code is shown in Figure 5.9.



Long sections of repeated code are generally better off as normal functions: The savings in

memory space is worth the comparatively small sacrifice in execution speed. But making a

short section of code into an ordinary function may result in little savings in memory space,

while imposing just as much time penalty as a larger function. In fact, if a function is very

short, the instructions necessary to call it may take up as much space as the instructions within

the function body, so that there is not only a time penalty but a space penalty as well.

In such cases you could simply repeat the necessary code in your program, inserting the same

group of statements wherever it was needed. The trouble with repeatedly inserting the same

code is that you lose the benefits of program organization and clarity that come with using

functions. The program may run faster and take less space, but the listing is longer and more

complex.

The solution to this quandary is the *inline function*. This kind of function is written like a normal

function in the source file but compiles into inline code instead of into a function. The

source file remains well organized and easy to read, since the function is shown as a separate

entity. However, when the program is compiled, the function body is actually inserted into the

program wherever a function call occurs.

Chapter 5

196

Functions that are very short, say one or two statements, are candidates to be inlined. Here’s

INLINE, a variation on the CONVERT2 program. It inlines the lbstokg() function.

// inliner.cpp

// demonstrates inline functions

#include <iostream>

using namespace std;

// lbstokg()

// converts pounds to kilograms

inline float lbstokg(float pounds)

{

return 0.453592 \* pounds;

}

//--------------------------------------------------------------

int main()

{

float lbs;

cout << “\nEnter your weight in pounds: “;

cin >> lbs;

cout << “Your weight in kilograms is “ << lbstokg(lbs)

<< endl;

return 0;

}

It’s easy to make a function inline: All you need is the keyword inline in the function definition:

inline float lbstokg(float pounds)

You should be aware that the inline keyword is actually just a *request* to the compiler.

Sometimes the compiler will ignore the request and compile the function as a normal function.

It might decide the function is too long to be inline, for instance.

(C programmers should note that inline functions largely take the place of #define macros in

C. They serve the same purpose but provide better type checking and do not need special care

with parentheses, as macros do.)