C++ Without Fear: Functions

By Brian Overland
May 17, 2011

Recursive Functions

So far, I've only shown the use of **main** calling other functions defined in the program, but in fact, any function can call any function. But can a function call itself?

Yes. And as you'll see, it's less crazy than it sounds. The technique of a function calling itself is called *recursion*. The obvious problem is the same one for infinite loops: If a function calls itself, when does it ever stop? The problem is easily solved, however, by putting in some mechanism for stopping.

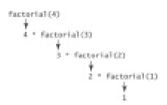
Remember the factorial function from Exercise 4.1.1 (page 90)? We can rewrite this as a recursive function:

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n – 1);  // RECURSION!
}
```

For any number greater than 1, the factorial function issues a call to itself but with a lower number. Eventually, the function factorial(1) is called, and the cycle stops.

There is a literal *stack* of calls made to the function, each with a different argument for n, and now they start returning. The *stack* is a special area of memory maintained by the computer: It is a last-in-first-out (LIFO) mechanism that keeps track of information for all pending function calls. This includes arguments and local variables, if any.

You can picture how to call a factorial(4) this way.



Click to view larger image

Many functions that use a **for** statement can be rewritten so they use recursion instead. But does it always make sense to use that approach?

No. The example here is not an ideal one, because it causes the program to store all the values 1 through n on the stack, rather than totaling them up directly in a loop. This approach is not efficient. The next section makes a better use of recursion.

Example 4.3. Prime Factorization

The prime-number examples we've looked at so far are fine, but they have a limitation. They tell you, for example, that a number such as 12,001 is not prime, but they don't tell anything more. Wouldn't it be more useful to know what numbers divide into 12,001?

It'd be more useful to generate the *prime factorization* for any requested number. This would show us exactly what prime numbers divide into that number. For example, if the number 36 was input, we'd get this:

```
2, 2, 3, 3
```

If 99 was input, we'd get this:

```
3, 3, 11
```

And if a prime number was input, the result would be the number itself. For example, if 17 was input, the output would be 17.

We have almost all the programming code to do this already. Only a few changes need to be made to the prime-number code. To get prime-factorization, first get the lowest divisor, and then factor the remaining quotient. To get all the divisors for a number n, do this:

```
For all whole numbers from 2 to the square root of n,
If n is evenly divisible by the loop variable (i),
Print i followed by a comma, and
Rerun the function on n / i, and
Exit the current function
If no divisors found, print n itself
```

This logic is a recursive solution, which we can implement in C++ by having the function get_divisors call itself.

prime3.cpp

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

void get_divisors(int n);

int main() {
    int n = 0;

    cout << "Enter a number and press ENTER: ";
    cin >> n;
    get_divisors(n);
    cout << endl;
    system("PAUSE");
    return 0;
}

// Get divisors function
//  This function prints all the divisors of n,
//  by finding the lowest divisor, i, and then
//  rerunning itself on n/i, the remaining quotient.

void get_divisors(int n) {
     int i;
     double sqrt_of_n = sqrt(n);
     for (i = 2; i <= sqrt_of_n; i++)
         if (n % i == 0) {   // If i divides n evenly,
            cout << i << ", ";    //   Print i,
            get_divisors(n / i);  //   Factor n/i,
            return;               //   and exit.
         }

     // If no divisor is found, then n is prime;
     //  Print n and make no further calls.

     cout << n;
}
```

How It Works

As always, the program begins by declaring functions—in this case, there is one function other than **main**. The new function is get_divisors.

Also, the beginning of the program includes iostream and cmath, because the program uses **cout**, **cin**, and **sqrt**. You don't need to declare **sqrt** directly, by the way, because this is done for you in cmath.

```
#include <iostream>
#include <cmath>

void get_divisors(int n);
```

The **main** function just gets a number from the user and calls get_divisors.

```
int main() {
    int n = 0;

    cout << "Enter a number and press ENTER: ";
    cin >> n;

    cout << endl;
    system("PAUSE");
    return 0;
}
```

The get_divisors function is the interesting part of this program. It has a **void** return value, meaning that it doesn't pass back a value. But it still uses the **return** statement to exit early.

```
void get_divisors(int n) {
     int i;
     double sqrt_of_n = sqrt(n);

     for (i = 2; i <= sqrt_of_n; i++)
         if (n % i == 0) {  // If i divides n evenly,
            cout << i << ", ";    //   Print i,
            get_divisors(n / i);  //   Factor n/i,
            return;               //   and exit.
         }

    // If no divisor is found, then n is prime;
    //  Print n and make no further calls.

     cout << n;
}
```

The heart of this function is a loop that tests numbers from 2 to the square root of n (which has been calculated and placed in the variable sqrt_of_n).

```
for (i = 2; i <= sqrt_of_n; i++)
    if (n % i == 0) {  // If i divides n evenly,
       cout << i << ", ";    //   Print i,
       get_divisors(n / i);  //   Factor n/i,
       return;               //   and exit.
    }
```

If the expression n % i == 0 is true, that means the loop variable i divides evenly into n. In that case, the function does several things: It prints out the loop variable, which is a divisor; calls itself recursively; and exits.

The function calls itself with the value n/i. Because the factor i is already accounted for, the function needs to get the prime-number divisors for *the remaining factors* of n, and these are contained in n/i.

If no divisors are found, that means the number being tested is prime. The correct response is to print this number and stop.

```
cout << n;
```

For example, suppose that 30 is input. The function tests to see what the lowest divisor of 30 is. The function prints the number 2 and then reruns itself on the remaining quotient, 15 (because 30 divided by 2 is 15).

During the next call, the function finds the lowest divisor of 15. This is 3, so it prints 3 and then reruns itself on the remaining quotient, 5 (because 15 divided by 3 is 5).

Here's a visual summary. Each call to get_divisors gets the lowest divisor and then makes another call unless the number being tested is prime.

Click to view larger image

Interlude: Interlude for Math Junkies

A little reflection shows why the lowest divisor is always a prime number. Suppose we test a positive whole number and that A is the lowest divisor *but is not a prime*. Since A is not prime, it must have at least one divisor of its own, B, that is not equal to either 1 or A.

But if B divides evenly into A and A is a divisor of the target number, then B must also be a divisor of the target number. Furthermore, B is less than A. Therefore, the hypothesis that the lowest divisor is not prime results in a contradiction.

This is easy to see by example. Any number divisible by 4 (a nonprime) is also divisible by 2 (a prime). The prime factors will always be found first, as long as you keep looking for the lowest divisor.

Exercises

Exercise 4.3.1. Rewrite the **main** function for Example 4.3 so that it prints the prompt message "Enter a number (0 = exit) and press ENTER." The program should call get_divisors to show the prime factorization and then prompt the user again, until he or she enters 0. (Hint: If you need to, look at the code for Example 4.2, on page 90.)

Exercise 4.3.2. Write a program that calculates triangle numbers by using a recursive function. A triangle number is the sum of all whole numbers from 1 to N, in which N is the number specified. For example, triangle(5) = 5 + 4 + 3 + 2 + 1.

Exercise 4.3.3. Modify Example 4.3 so that it uses a *nonrecursive* solution. You will end up having to write more code. (Hint: To make the job easier, write two functions: get_all_divisors and get_lowest_divisor. The

main function should call get_all_divisors, which in turn has a loop: get_all_divisors calls get_lowest_divisor repeatedly, each time replacing n with n/i, where i is the divisor that was found. If n itself is returned, then the number is prime, and the loop should stop.)

Example 4.4. Euclid's Algorithm for GCF

In the early grades of school, we're asked to figure out greatest common factors (GCFs). For example, the greatest common factor of 15 and 25 is 5. Your teacher probably lectured you about GCF until you didn't want to hear about it anymore.

Wouldn't it be nice to have a computer figure this out for you? We'll focus just on GCF, because as I'll show in Chapter 11, if you can figure out the CGF of two numbers, you can easily compute the lowest common multiple (LCM).

The technique was worked out almost 2,500 years ago by a Greek mathematician named Euclid, and it's one of the most famous in mathematics.

To get CGF: For whole two numbers A and B:

If B equals 0,
The answer is A.

Else
The answer is GCF(B, A%B)

You may remember remainder division (%) from earlier chapters. A%B means this:

Divide A by B and produce the remainder.

For example, 5%2 equals 1, and 4%2 equals 0. A result of 0 means that B divides A evenly.

If B does not equal 0, the algorithm replaces the arguments A, B with the arguments B, A%B and calls itself recursively. This solution works for two reasons:

- The terminal case (B equals 0) is valid. The answer is A.
- The general case is valid: GCF(A, B) equals CGF(B, A%B), so the function calls itself with new arguments B and A%B.

The terminal case, in which B equals 0, is valid assuming A is nonzero. You can see that A divides evenly into both itself and 0, but nothing larger can divide into A. (Note that 0 can be divided evenly by any whole number except itself.) For example, 997 is the greatest common factor for the pair (997, 0). Nothing larger divides evenly into both.

The general case is valid if the following is true:

The greatest common factor of the pair (B, A%B) is also the greatest common factor of the pair (A, B).

It turns out this *is* true, and because it is, the GCF problem is passed along from the pair (A, B) to the pair (B, A%B). This is the general idea of recursion: Pass the problem along to a simpler case involving smaller numbers.

It can be shown that the pair (B, A%B) involves numbers less than or equal to the pair (A, B). Therefore, during each recursive call, the algorithm uses successively smaller numbers until B is zero.

I save the rest of the proof for an interlude at the end of this section. Here is a complete program for computing greatest common factors:

gcf.cpp

```
#include <cstdlib>
#include <iostream>
using namespace std;
int gcf(int a, int b);

int main()
{
    int a = 0, b = 0; // Inputs to GCF.

    cout << "Enter a: ";
    cin >> a;
    cout << "Enter b: ";
    cin >> b;
    cout << "GCF = " << gcf(a, b) << endl;

    system("PAUSE");
    return 0;
}

int gcf(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcf(b, a%b);
}
```

How It Works

All that **main** does in this case is to prompt for two input variables a and b, call the greatest-common-factor function (gcf), and print results:

```
cout << "GCF = " << gcf(a, b) << endl;
```

As for the gcf function, it implements the algorithm discussed earlier:

```
int gcf(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcf(b, a%b);
}
```

The algorithm keeps assigning the old value of B to A and the value A%B to B. The new arguments are equal or less to the old. They get smaller until B equals 0.

For example, if we start with A = 300 and B = 500, the first recursive call switches their order. (This always happens if B is larger.) From that point onward, each call to gcf involves smaller arguments until the terminal case is reached:

Value of A	**Value of B**	**Value of A%B (Divide and Get Remainder)**
300	500	300
500	300	200
300	200	100
200	100	0
100	0	Terminal case: answer is 100

When B is 0, the gcf function no longer computes A%B but instead produces the answer.

If the initial value of A is larger than B, the algorithm produces an answer even sooner. For example, suppose A = 35 and B = 25.

Value of A	**Value of B**	**Value of A%B (Divide and Get Remainder)**
35	25	10
25	10	5
10	5	0

5	0	Terminal case: answer is 5

Interlude: Who Was Euclid?

Who was this Euclid guy? Wasn't he the Greek who wrote about geometry? (Something like "The shortest distance between two points is a straight line"?)

Indeed he was. Euclid's *Elements* is one of the most famous books in Western civilization. For almost 2,500 years it was used as a standard textbook in schools. In this work he demonstrated for the first time a *tour de force* of deductive logic, proving all that was then known about geometry. In fact, he invented the whole *idea* of proof. It is a great work that has had profound influence on mathematicians and philosophers ever since.

It was Euclid who (according to legend) said to King Ptolemy of Alexandria, "Sire, there is no royal road to geometry." In other words, you gotta work for it.

Although its focus is on geometry, Euclid's book has results in number theory as well. The algorithm here is the most famous of these results. Euclid expressed the problem geometrically, finding the biggest length commensurable with two sides of a rectangle. He conceived the problem in terms of rectangles, but we can use any two integers.

Exercises

Exercise 4.4.1. Revise the progra ere is a sample output:

```
GCF(500, 300) =>
GCF(300, 200) =>
GCF(200, 100) =>
GCF(100, 0) =>
100
```

Exercise 4.4.2. For experts: Revise the gcf function so that it uses an iterative (loop-based) approach. Each cycle through the loop should stop if B is zero; otherwise, it should set new values for A and B and then continue. You'll need a temporary variable—temp—to hold the old value of B for a couple of lines: temp=b, b=a%b, and a=temp.

Interlude: Interlude for Math Junkies: Rest of the Proof

Earlier, I worked out some of a proof of Euclid's algorithm. What remains is to show that the greatest common factor of the pair (B, A%B) is also the greatest common factor of the pair (A, B). This is true if we can show the following:

- If a number is a factor of both A and B, it is also a factor of A%B.
- If a number is a factor of both B and A%B, it is also a factor of A.

If these are true, then all the common factors of one pair are common factors of the other pair. In other words, the set of Common Factors (A, B) is identical to the set of common factors (B, A%B). Since the two sets are identical, they have the *greatest member*—therefore, they share the greatest common factor.

Consider the remainder-division operator (%). It implies the following, where m is a whole number:

```
A = mB + A%B
```

A%B is equal or less than A, so the general tendency of the algorithm is to get progressively smaller numbers. Assume that n, a whole number, is a factor of both A and B (meaning it divides both evenly). In that case:

```
A = cn
B = dn
```

where c and d are whole numbers. Therefore:

```
cn = m(dn) + A%B
A%B = cn – mdn = n(c – md)
```

This demonstrates that if n is a factor of both A and B, it is also a factor of A%B. By similar reasoning, we can show that if n is a factor of both B and A%B, it is also a factor of A.

Because the common factors for the pair (A, B) are identical to the common factors for the pair (B, A%B), it follows that they share the greatest common factor. Therefore, GCF(A, B) equals GCF(B, A%B). QED.

Example 4.5. Beautiful Recursion: Tower of Hanoi

Strictly speaking, the earlier examples don't require recursion. With some effort, they can be revised as iterative (loop-based) functions. But there is a problem that illustrates recursion beautifully, solving a problem that would be very difficult to solve otherwise.

This is the Tower of Hanoi puzzle: You have three stacks of rings. Each ring is smaller than the one it sits on. The challenge is to move all the rings from the first stack to the third, subject to these constraints:

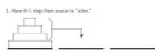
- You can move only one ring at a time.
- You can place a ring only on top of a larger ring, never a smaller.

It sounds easy, until you try it! Consider a stack four rings high: You start by moving the top ring from the first stack, but where do you move it, and what do you do after that?

To solve the problem, assume we already know how to move a group of N–1 rings. Then, to move N rings from a source stack to a destination stack, do the following:

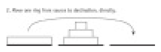
1. Move N–1 rings from the source stack to the (currently) unused, or "other," stack.
2. Move a single ring from the source stack to the destination stack.
3. Move N–1 rings from the "other" stack to the destination stack.

This is easier to envision graphically. First, the algorithm moves N–1 rings from the source stack to the "other" stack ("other" being the stack that is neither source nor destination for the current move). In this case, N is 4 and N–1 is 3, but these numbers will vary.



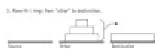
Click to view larger image

After this recursive move, at least one ring is left at the top of the source stack. This top ring is then moved: This is a simple action, moving one ring from source to destination.



Click to view larger image

Finally, we perform another recursive move, moving N–1 rings from "other" (the stack that is currently neither source nor destination) to the destination.



Click to view larger image

What permits us to move N–1 rings in steps 1 and 3, when the constraints tell us that we can move only one?

Remember the basic idea of recursion. Assume the problem *has already been solved* for the case N–1, although this may require many steps. All we have to do is tell the program how to solve the Nth case in terms of the N–1 case. The program magically does the rest.

It’s important, also, to solve the terminal case, N = 1. But that’s trivial. Where one ring is involved, we simply move the ring as desired.

Click to view larger image

The following program shows the C++ code that implements this algorithm:

tower.cpp

```
#include <cstdlib>
#include <iostream>

using namespace std;
void move_rings(int n, int src, int dest, int other);

int main()
{
  int n = 3;  // Stack is 3 rings high

  move_rings(n, 1, 3, 2); // Move stack 1 to stack 3
  system("PAUSE");
  return 0;
}

void move_rings(int n, int src, int dest, int other) {
  if (n == 1) {
    cout << "Move from " << src << " to " << dest
         << endl;
  } else {
    move_rings(n - 1, src, other, dest);
    cout << "Move from " << src << " to " << dest
         << endl;
    move_rings(n - 1, other, dest, src);
  }
}
```

How It Works

The program is brief considering what it does. In this example, I’ve set the stack size to just three rings, although it can be any positive integer:

```
int n = 3;  // Stack is 3 rings high
```

The call to the move_rings function says that three rings should be moved from stack 1 to stack 3; these are determined by the second and third arguments, respectively. The “other” stack, stack 2, will be used in intermediate steps.

```
     move_rings(n, 1, 3, 2); // Move stack 1 to stack
3
```

This small example—moving only three rings—produces the following output. You can verify the correctness of this solution by using three different coins, all of different sizes.

```
Move from 1 to 3
Move from 1 to 2
Move from 3 to 2
Move from 1 to 3
Move from 2 to 1
Move from 2 to 3
Move from 1 to 3
```

Try setting n to 4, and you'll get a list of moves more than twice as long.

The core of the move_ring function is the following code, which implements the general solution described earlier. Remember, this recursive approach assumes the N–1 case has already been solved. The function therefore passes along most of the problem to the N–1 case.

```
move_rings(n - 1, src, other, dest);
cout << "Move from " << src << " to " << dest
     << endl;
move_rings(n - 1, other, dest, src);
```

Notice how the functional role of the three stacks is continually switched between *source* (where to move a group of rings from), *destination* (where the group is going), and *other* (the intermediate stack, which is not used now but will be at the next level).

Exercises

Exercise 4.5.1. Revise the program so that the user can enter any positive integer value for n. Ideally, you should test the input to see whether it is greater than 0.

Exericse 4.5.2. Instead of printing the "Move" message directly on the screen, have the move_ring function call yet another function, which you give the name exec_move. The exec_move function should take a source and destination stack number as its two arguments. Because this is a separate function, you can use as many lines of code as you need to print a message. You can print a more informative message:

```
Move the top ring from stack 1 to stack 3.
```

Example 4.6. Random-Number Generator

OK, we've had enough fun with recursion. It's time to move on to another, highly practical example. This one generates random numbers—a function at the heart of many game programs.

The test program here simulates any number of dice rolls. It does this by calling a function, rand_0toN1, which takes an argument, n, and randomly returns a number from 0 to n – 1. For example, if the user inputs the number 6, this program simulates dice rolls:

```
3 4 6 2 5 3 1 1 6
```

Here is the program code:

dice.cpp

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

int rand_0toN1(int n);
int main() {
    int n, i;
    int r;

    srand(time(NULL)); // Set seed for random numbers.

    cout << "Enter number of dice to roll: ";
    cin >> n;

    for (i = 1; i <= n; i++) {
        r = rand_0toN1(6) + 1; // Get a number 1 to 6
        cout << r << " ";      // Print it
    }
    system("PAUSE");
    return 0;
}

// Random 0-to-N1 Function.
// Generate a random integer from 0 to N–1, with each
//  integer an equal probability.
//
int rand_0toN1(int n) {
    return rand() % n;
}
```

How It Works

The beginning of the program has to include a number of files to support the functions needed for random-number generation:

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

Make sure you include the last three here—cmath, cstdlib, and ctime—whenever you use random-number generation.

Random-number generation is a difficult problem in computing, because computers follow deterministic rules—which, by definition, are nonrandom. The solution is to generate what's called a *pseudorandom* sequence by taking a number and performing a series of complex transformations on it.

To do this, the program needs a number as random as possible to start off the sequence. So, we're back where we started, aren't we?

Well, fortunately no. You can take the system time and use it as a *seed*: That is the first number in the sequence.

```
srand(time(NULL));
```

NULL is a predefined value that means a data address set to nothing. You don't need to worry about it for now. The effect in this case is simply to get the current time.

C++0x

The C++0x specification provides the **nullptr** keyword, which should be used in preference to NULL if you have a C++0x-compliant compiler.

A program that uses random numbers should call **srand** first. System time changes too quickly for a human to guess its exact value, and even a tiny difference in this number causes big changes in the resulting sequence. This is a practical application of what chaos theorists call the Butterfly Effect.

The rest of **main** prompts for a number and then prints the quantity of random numbers requested. A **for** loop makes repeated calls to rand_0toN1, a function that returns a random number from 0 to n – 1:

```
for (i = 1; i <= n; i++) {
    r = rand_0toN1(6) + 1;  // Get num from 1 to 6
    cout << r << " ";       // Print it out
}
```

Here is the function definition for the rand_0toN1 function:

```
int rand_0toN1(int n) {
    return rand() % n;
}
```

This is one of the simplest functions we've seen yet! Calling **rand** produces a number anywhere in the range of the **int** type, which, on 32-bit systems, can be anywhere in the range of roughly plus or minus two billion. But we want much smaller numbers.

The solution is to use your old friend, the remainder-division operator (%), to divide by n and return the remainder. No matter how large the amount being divided, the result must be a number from 0 to n–1, which is exactly what the function is being asked to provide.

In this case, the function is called with the argument 6, so it returns a value from 0 to 5. Adding 1 to the number gives a random value in the range 1 to 6, which is what we want.

Exercises

Exercise 4.4.1. Write a random-number generator that returns a number from 1 to N (rather than 0 to N–1), where N is the integer argument passed to it.

Exercise 4.4.2. Write a random-number generator that returns a random floating-point number between 0.0 and 1.0. (Hint: Call **rand**, cast the result r to type **double** by using `static_cast<double>(r)`, and then divide by the highest value in the **int** range, **RAND_MAX**.) Make sure you declare the function with the **double** return type.

Share This Save To Your Account < Back **Page 4** of 6 Next >

Discussions

Comments for this thread are now closed. ×

0 Comments InformIT 1 Login

Recommend Share Sort by Oldest

This discussion has been closed.

ALSO ON INFORMIT WHAT'S THIS?

Windows 10--All That & A Bag of Chips

1 comment • a month ago

MarkCPhinn — Windows 10 Professional Is a Sick, Useless Dog Upgraded from Windows 7 …

How to Use Regular Expressions TODAY in Your Windows …

1 comment • a month ago

henryb — The '\d' character class matches 1 digit if the .NET Reg Ex is PCRE compliant.. the statement …

Network Switching Methods: Store-and-Forward Versus …

1 comment • 4 months ago

Ahmed Zabara — Cut-through : evolution ... Now that's new .

Top 10 Architectural, Organizational and Process …

2 comments • 3 months ago

Mike Fisher — Thanks Infiyaz. We agree the structure of the development team + environment …

Subscribe Add Disqus to your site Privacy DISQUS