Functions in C++

Eric Roberts
CS 106B
January 7, 2015

The Syntax of a Function Definition

• The general form of a function definition looks essentially the same as it does in Java:

```c++
type name (parameter list) {
    statements in the function body
}
```

where `type` indicates what type the method returns, `name` is the name of the method, and `parameter list` is a list of variable declarations used to hold the values of each argument.

• All functions need to be declared before they are called by specifying a prototype consisting of the header line followed by a semicolon.

Computing Factorials

• The factorial of a number \( n \) (which is usually written as \( n! \) in mathematics) is defined to be the product of the integers from 1 up to \( n \). Thus, 5! is equal to 120, which is \(1 \times 2 \times 3 \times 4 \times 5\).

```c++
int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

• The following function definition uses a for loop to compute the factorial function:

```c++
int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

C++ Enhancements to Functions

• Functions can be overloaded, which means that you can define several different functions with the same name as long as the correct version can be determined by looking at the number and types of the arguments. The pattern of arguments required for a particular function is called its signature.

• Functions can specify optional parameters by including an initializer after the variable name. For example, the function prototype

```c++
void setMargin(int margin = 72);
```

Indicates that `setMargin` takes an optional argument that defaults to 72.

• C++ supports call by reference, which allows functions to share data with their callers.

Call by Reference

• C++ indicates call by reference by adding an ampersand (\&) before the parameter name. A single function often has both value parameters and reference parameters, as illustrated by the `solveQuadratic` function from Figure 2-3 on page 76, which has the following prototype:

```c++
void solveQuadratic(double a, double b, double c, double & x1, double & x2);
```

• Call by reference has two primary purposes:
  - It creates a sharing relationship that makes it possible to pass information in both directions through the parameter list.
  - It increases efficiency by eliminating the need to copy an argument. This consideration becomes more important when the argument is a large object.
Call by Reference Example

- The following function swaps the values of two integers:
  ```cpp
  void swap(int & x, int & y) {
    int tmp = x;
    x = y;
    y = tmp;
  }
  ```
- The arguments to `swap` must be assignable objects, which for
  the moment means variables.
- If you left out the `&` characters in the parameter declarations,
  calling this function would have no effect on the calling
  arguments because the function would exchange local copies.

Libraries and Interfaces

- Modern programming depends on the use of libraries. When
  you create a typical application, you write only a tiny fraction
  of the code.
- Libraries can be viewed from two perspectives. Code that
  uses a library is called a client. The code for the library itself
  is called the implementation.
- The point at which the client and the implementation meet is
called the interface, which serves as both a barrier and a
communication channel:

```
client implementation
```
Exercise: Finding Perfect Numbers

- Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of \( n \) is any divisor less than \( n \) itself). They called such numbers perfect numbers. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

- Our first exercise today is to design and implement a C++ program that finds all the perfect numbers between two limits entered by the user, as follows:

```
Enter lower limit: 6
28
496
8128
Enter upper limit: 1
10000
```

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function, which can be defined in either of the following ways:

\[
\begin{align*}
5! & = 5 \times (5-1) \times (5-2) \times (5-3) \times 2 \times 1 \\
0! & = 1 \\
\text{otherwise} & = n \times (n-1)! \\
\end{align*}
\]

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

```
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (first is a simple case) {
    Compute and return the simple solution without using recursion.
} else {
    Divide the problem into one or more subproblems that have the same form. Solve each of the problems by calling this method recursively. Return the solution from the results of the various subproblems.
}
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify simple cases that can be solved without recursion.
  2. Find a recursive decomposition that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

Exercise: Greatest Common Divisor

One of the oldest known algorithms that is worthy of the title is Euclid’s algorithm for computing the greatest common divisor (GCD) of two integers, \( x \) and \( y \). Euclid’s algorithm is usually implemented iteratively using code that looks like this:

```
int gcd(int x, int y) {
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid’s insight that the greatest common divisor of \( x \) and \( y \) is also the greatest common divisor of \( y \) and the remainder of \( x \) divided by \( y \).