Functions

Chris Wild & Steven Zeil

May 25, 2013

Contents

The basic C++ language is relatively primitive. Its basic data types (integers, reals and characters) and operations on them (addition, subtraction, etc). are limited.

To build complex programs one needs to combine these primitive elements into larger building blocks. Then we compose the entire program from a combination of these building blocks.

In C++, *functions* are building blocks composed of groups of related statements.

Later in this course, we will look at *modules*, which are building blocks made from a combination of functions, variables, and data types. Eventually, we will see that the C++ *class* is a good way to implement most modules.

It's even possible to go further. (Think back to our discussion in the orientation about scaling up to larger and larger projects.) We can combine classes into larger building blocks called *namespaces*, but we won't get there in this course.

1 Functions

A *function* is a packaging of C++ statements into a unit which called as needed from elsewhere in the program.

- Every C++ program has at least one function - it is called `main`.
- The same function can be called many times in the same or even different programs. Thus functions are reusable pieces of program.
- The job that a function does usually *abstracts* some part of the problem and thus becomes a convenient building block for programming complex systems.
- Functions extend the things that a C++ program can do. Many commonly used functions are placed in libraries and accessed through their header files (e.g."`#include <iostream>`" accesses the common and useful functions for doing I/O).
- The identification of "good" functions (and classes) is a critical part of the art of programming. Good functions lead to programs that are easier to write and maintain. Bad functions lead to programs which are difficult to understand and thus prone to failures.
- Functions are program modules which are called by other parts of the program. Functions and their calling programs communicate information in various ways.

  - *Parameters* are data objects which are exchanged between the calling program and the function.
    * *Input parameters* are data objects which are passed from the calling program to the function.
    * *Output parameters* are data objects which are passed from the function to the calling program.
    * It is possible for a single parameter to be passed both ways, in which case it is simultaneously an input and an output parameter.
  - The *return value* is a single data object which is returned as the value of the function.

- *Global variables* are variables which can be accessed by both the function and the calling program. Since the setting of global variables and their use can be difficult to determine when examining the function call, their use is tricky and is general discouraged as bad programming practice. Such data should be passed via parameters instead.
  - In some cases, this will result in an uncomfortably large number of parameters. Later we will look at *structured data* which can be used to group data items together and at *classes*, which can also pass data to *member functions* via *data members*.

- A function has the following parts

  name: any valid c++ identifier.

  return value type: any data type or class name. *void* is used to denote that this function does not return a value.

  parameter list: an ordered list giving the types of the parameters.

  body: a compound statement, inside `{ }`, containing the program statements that implement the function.

- A function *prototype* consists of the return-type function-name and parameter-list.
  - We write a function's prototype into our code to *declare* the function. This gives the compiler enough information to set up calls to that function without knowing all the details of the function (the statements that make up its body).

    For example,

```
void foo(); // declaration of foo
int bar(int i); // declaration of bar
```


These declarations tell the compiler (and you!) that these functions exist, what names you can use to invoke them, how many parameters they take and what the data types of those parameters are, and what kind of data these functions will return.

That's pretty much all that you need to know in order to write a legal call to these functions and for the compiler to compile that call. Of course, it doesn't tell you what those functions actually *do*, but that's a separate issue.

  - This is very important because it allows functions to be defined separately and put into libraries for common use.
  - *Header* files consist mainly of function prototypes (including member functions of classes).

- A function *signature* consists of the function-name and parameter-list.

  A function signature is used by the compiler for handling function overload.

- A function *definition* consists of all the elements of a function: name, return type, parameter list, and body.

  For example,

```
void foo();  // declaration of foo

// definition of foo:
void foo()
{
  cout << "Listen!" << endl;
  int i = bar(42);
}
```

  - Because a function definition contains all the information expected in a declaration, all definitions are also declarations.

2 Example

```
// define a function that squares an integer number – a simple example to illustrate the concepts
// function "square" takes on input parameter called "x" and returns the square of x
```


```
int square(int x)
{
 return x*x;
}
```

- the function prototype of this function is

```
int square(int); // naming the input paramter is optional
```

- the function signature is

```
//  square(int)
```

3 Tips

- A function prototype must end in a semi-colon (';'). It's absence is a common programming error
- A function definition must NOT contain a semi-colon (';') after the parameter-list. Otherwise the compiler will confuse it with a function prototype. Instead a function definition is followed by a compound statement containing the function body. It is actually the function body which defines the function.
- Every function prototype should have a corresponding function definition elsewhere in the program, possibly in a separate program file or library.
- The signatures and return types MUST agree between a prototype and definition. If not, the compiler and loader will not be able to find the function definition and will report undefined function.
- If you are placing functions that call one another into the same file, you have a choice of whether to declare and define the low level (called) or the high level (calling) functions first. It is matter of preference really.
  - However, you *must* declare a function before calling it from another.

  For example, given two functions `foo` and `bar`, such that `foo` calls `bar`, this is legal:

```
void bar() // declaration and definition of bar.
{
   cout << "I think I hear someone calling me." << endl;
}

void foo() // declaration and definition of foo.
{
   cout << "Listen!" << endl;
   bar();   // Notice that bar() has already been declared
}
```

and so is this

```
void foo(); // declaration of foo
void bar(); // declaration of bar

void bar() // definition of bar.
{
   cout << "I think I hear someone calling me." << endl;
}

void foo() // definition of foo.
{
   cout << "Listen!" << endl;
   bar();   // Notice that bar() has already been declared
}
```

and this

```
void foo(); // declaration of foo
void bar(); // declaration of bar

void foo() // definition of foo.
{
   cout << "Listen!" << endl;
```


```
    bar();    // Notice that bar() has already been declared
}

void bar() // definition of bar.
{
    cout << "I think I hear someone calling me." << endl;
}
```

But *this* is not legal:

```
void foo() // declaration and definition of foo.
{
    cout << "Listen!" << endl;
    bar();    // Oops! bar() has not yet been declared.
}

void bar() // declaration and definition of bar.
{
    cout << "I think I hear someone calling me." << endl;
}
```

4 Overloading

- Overloading a function means using the same function name for two or more different function definitions.
- A common example is the "+" operator (which is an infix function), which can be used to add integers or floating points numbers

  Another common example would be the I/O functions "<<" and ">>", which can handle a variety of data types.

  Of course, those are operators. Later we'll learn that they really are just funny-looking functions, but we can do overloading with ordinary functions as well:

```
void bar()
```


```
{
   cout << "I think I hear someone calling me." << endl;
}

int bar(int i) // bar is now overloaded
{
   return i * i;
}

void foo()
{
   cout << "Listen!" << endl;
   bar();
   int k = bar(12);
   cout << k << ": that's just gross!" << endl;
}
```

- So how does the compiler tell the difference? By the number and type of the parameters to the function. As long as the number or type of the parameters are different, there is no problem.
- Basically if you can tell from the function prototype (without the parameter names), they are different