Functions

Functions allow to structure programs in segments of code to perform individual tasks.

函数允许在代码段中构造程序来执行单独的任务。

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

在c++中，函数是一组有名称的语句，可以从程序的某个位置调用。定义函数最常见的语法是:

`type name ( parameter1, parameter2, ...) { statements }`
Where:

- `type` is the type of the value returned by the function.

type为函数返回值的类型。

- `name` is the identifier by which the function can be called.

name是可以调用函数的标识符

- `parameters` (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: `int x`), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.

参数(根据需要而定):每个参数由一个类型和一个标识符组成，每个参数之间用逗号分隔。每个形参看起来非常像一个常规变量声明(例如:int x)，实际上在函数中作为一个局部的常规变量。形参的目的是允许从调用函数的位置向函数传递参数。

- `statements` is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

语句是函数的主体。它是一个由大括号{}包围的语句块，用来指定函数的实际功能。

Let's have a look at an example:

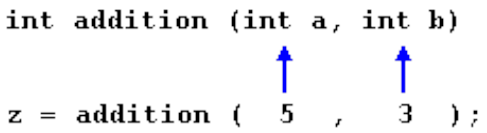
```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

This program is divided in two functions: `addition` and `main`. Remember that no matter the order in which they are defined, a C++ program always starts by calling `main`. In fact, `main` is the only function called automatically, and the code in any other function is only executed if its function is called from `main` (directly or indirectly).

这个程序分为两个功能:加法和主。记住，不管它们的定义顺序如何，c++程序总是从调用main开始。实际上，main是唯一一个自动调用的函数，任何其他函数的代码只有在从main(直接或间接)调用其函数时才会执行。

In the example above, `main` begins by declaring the variable `z` of type `int`, and right after that, it performs the first function call: it calls `addition`. The call to a function follows a structure very similar to its declaration. In the example above, the call to `addition` can be compared to its definition just a few lines earlier:

在上面的例子中，main首先声明int类型的变量z，紧接着，它执行第一个函数调用:它调用加法。函数调用遵循与函数声明非常相似的结构。在上面的例子中，加法的调用可以与它的定义进行比较，就在前面几行:



The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, `5` and `3`, to the function; these correspond to the parameters `a` and `b`, declared for function `addition`.

函数声明中的形参与函数调用中传递的实参有明确的对应关系。调用将两个值5和3传递给函数;这些对应于为函数加法声明的参数a和b。

At the point at which the function is called from within main, the control is passed to function `addition`: here, execution of `main` is stopped, and will only resume once the `addition` function ends. At the moment of the function call, the value of both arguments (`5` and `3`) are copied to the local variables `int a` and `int b` within the function.

当从main中调用函数时，控件被传递给函数加法:在这里，main的执行停止，只有当加法函数结束时才会恢复。在调用函数时，参数(5和3)的值都被复制到函数内的局部变量int a和int b中。

Then, inside `addition`, another local variable is declared (`int r`), and by means of the expression `r=a+b`, the result of `a` plus `b` is assigned to `r`; which, for this case, where `a` is 5 and `b` is 3, means that 8 is assigned to `r`.

然后，在加法内部声明另一个局部变量(int r)，并通过表达式r=a+b，将a+b的结果赋值给r;在这种情况下，a = 5 b = 3，意味着8被赋给r。

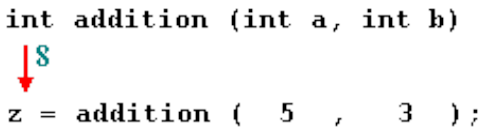
The final statement within the function:

函数中的最后一条语句:

```
return r;
```

Ends function `addition`, and returns the control back to the point where the function was called; in this case: to function `main`. At this precise moment, the program resumes its course on `main` returning exactly at the same point at which it was interrupted by the call to `addition`. But additionally, because `addition` has a return type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended `addition`: in this particular case, the value of the local variable `r`, which at the moment of the `return` statement had a value of 8.

结束函数加法，并将控件返回到调用函数的位置;在本例中:执行main函数。在这个精确的时刻，程序在main返回时正好在被加法调用中断的同一点上恢复它的进程。但此外,因为除了有一个返回类型,调用计算一个值,这个值是结束的返回语句中指定的值增加:在这种特殊情况下,局部变量的值r,此刻的返回语句的值8。



Therefore, the call to `addition` is an expression with the value returned by the function, and in this case, that value, 8, is assigned to `z`. It is as if the entire function call (`addition(5,3)`) was replaced by the value it returns (i.e., 8).

因此，对加法的调用是一个带有函数返回值的表达式，在本例中，这个值8被赋给z。这就好像整个函数调用(加法(5,3))被它返回的值(即8)替换了。

Then main simply prints this value by calling:

然后main简单地通过调用:

```
cout << "The result is " << z;
```

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

一个函数实际上可以在程序中被多次调用，它的参数自然不限于字面量:

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
}
```

Similar to the `addition` function in the previous example, this example defines a `subtract` function, that simply returns the difference between its two parameters. This time, `main` calls this function several times, demonstrating more possible ways in which a function can be called.

与前面示例中的加法函数类似，本示例定义了一个subtract函数，它只是返回两个参数之间的差值。这一次，main多次调用该函数，演示了调用函数的更多可能方式。

Let's examine each of these calls, bearing in mind that each function call is itself an expression that is evaluated as the value it returns. Again, you can think of it as if the function call was itself replaced by the returned value:

让我们检查每一个调用，记住每个函数调用本身就是一个表达式，它的值是它返回的值。同样，你可以把它想象成函数调用本身被返回值替换:

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
z = 5;
cout << "The first result is " << z;
```

With the same procedure, we could interpret:

```
cout << "The second result is " << subtraction (7,2);
```

as:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by `subtraction (7,2)`.

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. The function is called with the values `x` and `y` have at the moment of the call: 5 and 3 respectively, returning 2 as result.

传递给减法的参数是变量而不是文字。这也是有效的，并且工作得很好。函数调用时，值' x '和' y '分别为:5和3，结果返回2。

The fourth call is again similar:

```
z = 4 + subtraction (x,y);
```

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

唯一的加法是，现在函数调用也是加法操作的操作数。同样，结果与函数调用被其结果替换相同:6。注意，由于加法的交换性，上面的式子也可以写成:

```
z = subtraction (x,y) + 4;
```

With exactly the same result. Note also that the semicolon does not necessarily go after the function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

结果完全一样。还需要注意的是，分号不一定在函数调用之后出现，而是像往常一样出现在整个语句的末尾。同样，通过将函数调用替换为它们的返回值，可以很容易地再次看到背后的逻辑:

```
z = 4 + 2;     // same as z = 4 + subtraction (x,y);
z = 2 + 4;     // same as z = subtraction (x,y) + 4;
```

Functions with no type. The use of void

The syntax shown above for functions:

`type name ( argument1, argument2 ...) { statements }`
Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is `void`, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

要求声明以类型开始。这是函数返回值的类型。但是如果函数不需要返回值呢?在本例中，要使用的类型是void，这是一种表示没有值的特殊类型。例如，一个简单打印消息的函数可能不需要返回任何值:

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

`void` can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, `printmessage` could have been declared as:

Void也可以在函数的形参列表中使用，以显式指定函数在调用时不接受实际的形参。例如，printmessage可以声明为:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

In C++, an empty parameter list can be used instead of `void` with same meaning, but the use of `void` in the argument list was popularized by the C language, where this is a requirement.

在c++中，可以使用空形参表来代替具有相同含义的' void '，但' void '在实参表中的使用是由C语言推广的，在C语言中这是一个要求。

Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an empty pair of parentheses shall always be appended to the function name. See how `printmessage` was called in an earlier example:

函数名后面的括号在任何情况下都不是可选的，无论是在函数声明中还是调用函数时都是如此。即使函数不带参数，函数名后面也至少要加一对空括号。看看在前面的例子中' printmessage '是如何被调用的:

```
printmessage ();
```

The parentheses are what differentiate functions from other kinds of declarations or statements. The following would not call the function:

圆括号是函数与其他类型的声明或语句的区别。下面的函数不会调用该函数:

```
printmessage;
```

The return value of main

You may have noticed that the return type of `main` is `int`, but most examples in this and earlier chapters did not actually return any value from `main`.

你可能已经注意到main的返回类型是int，但是本章和前几章中的大多数例子实际上并没有从main返回任何值。

Well, there is a catch: If the execution of `main` ends normally without encountering a `return` statement the compiler assumes the function ends with an implicit return statement:

好吧，这里有一个问题:如果main的执行通常没有遇到return语句，编译器就会假设函数以隐式return语句结束:

```
return 0;
```

Note that this only applies to function `main` for historical reasons. All other functions with a return type shall end with a proper `return` statement that includes a return value, even if this is never used.

注意，由于历史原因，这只适用于main函数。所有其他具有返回类型的函数都应该以包含返回值的正确return语句结束，即使从来没有使用过。

When `main` returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by `main`, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for `main` that are guaranteed to be interpreted in the same way on all platforms are:

当main返回0(隐式或显式)时，环境将其解释为程序成功结束。其他值可能由main返回，有些环境以某种方式让调用者访问该值，尽管这种行为不是必需的，也不一定可以在平台之间移植。main的值保证在所有平台上以相同的方式解释:

value	description
`0`	The program was successful
`EXIT_SUCCESS`	The program was successful (same as above). This value is defined in header `<cstdlib>`.
`EXIT_FAILURE`	The program failed. This value is defined in header `<cstdlib>`.

Because the implicit `return 0;` statement for `main` is a tricky exception, some authors consider it good practice to explicitly write the statement.

因为隐式返回0;Statement for main是一个棘手的例外，一些作者认为显式地编写语句是一种良好的实践。

Arguments passed by value and by reference

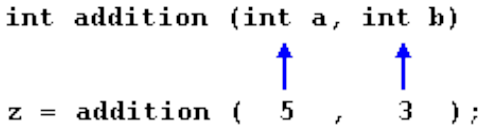
In the functions seen earlier, arguments have always been passed *by value*. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example, take:

在前面看到的函数中，参数总是按值传递。这意味着，当调用函数时，传递给函数的是调用时这些参数的值，这些参数被复制到由函数参数表示的变量中。例如,需要:

```
int x=5, y=3, z;
z = addition ( x, y );
```

In this case, function addition is passed 5 and 3, which are copies of the values of `x` and `y`, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the function's definition, but any modification of these variables within the function has no effect on the values of the variables x and y outside it, because x and y were themselves not passed to the function on the call, but only copies of their values at that moment.

在本例中，函数加法传递了5和3，它们分别是x和y值的副本。这些值(5和3)是用来初始化变量设置为参数的函数的定义,但任何修改这些变量在函数内没有影响外变量x和y的值,因为x和y是自己不传递到函数的调用,但那只是他们价值观的副本。



In certain cases, though, it may be useful to access an external variable from within a function. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the function `duplicate` in this code duplicates the value of its three arguments, causing the variables used as arguments to actually be modified by the call:

不过，在某些情况下，从函数内部访问外部变量可能很有用。为此，参数可以通过引用传递，而不是通过值传递。例如，这段代码中的函数复制了它的三个参数的值，导致作为参数使用的变量实际上被调用修改:

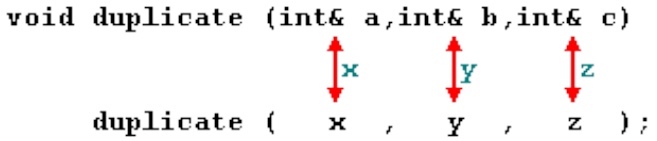
```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (`&`) following the parameter type, as in the parameters taken by `duplicate` in the example above.

为了获得对其参数的访问权，函数将其参数声明为引用。在c++中，参数类型后面用&符号表示引用，就像上面例子中duplicate使用的参数一样。

When a variable is passed *by reference*, what is passed is no longer a copy, but the variable itself, the variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call.

当通过引用传递变量时，传递的不再是副本，而是变量本身，由函数形参标识的变量，以某种方式与传递给函数的实参相关联，对函数中相应局部变量的任何修改都反映在调用中作为参数传递的变量中。



In fact, `a`, `b`, and `c` become aliases of the arguments passed on the function call (`x`, `y`, and `z`) and any change on `a` within the function is actually modifying variable `x` outside the function. Any change on `b` modifies `y`, and any change on `c` modifies `z`. That is why when, in the example, function `duplicate` modifies the values of variables `a`, `b`, and `c`, the values of `x`, `y`, and `z` are affected.

实际上，a、b和c成为传递给函数调用(x、y和z)的参数的别名，函数内部对a的任何更改实际上都是在函数外部修改变量x。对b的任何更改都会修改y，对c的任何更改都会修改z。这就是为什么在本例中，函数duplicate会修改变量a、b和c的值，而x、y和z的值也会受到影响。

If instead of defining duplicate as:

如果不是定义duplicate为:

```
void duplicate (int& a, int& b, int& c)
```

Was it to be defined without the ampersand signs as:

它的定义是不带&号:

```
void duplicate (int a, int b, int c)
```

The variables would not be passed *by reference*, but *by value*, creating instead copies of their values. In this case, the output of the program would have been the values of `x`, `y`, and `z` without being modified (i.e., 1, 3, and 7).

变量不会通过引用传递，而是通过值传递，从而创建变量值的副本。在这种情况下，程序的输出应该是未经修改的x、y和z值(即1、3和7)。

Efficiency considerations and const references

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as `int`, but if the parameter is of a large compound type, it may result on certain overhead. For example, consider the following function:

调用由value接受参数的函数将导致生成值的副本。对于基本类型(如int)来说，这是一种相对简单的操作，但如果形参是大型复合类型，则可能会产生一定的开销。例如，考虑以下函数:

```
string concatenate (string a, string b)
{
  return a+b;
}
```

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces `a` and `b` to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean copying large quantities of data just for the function call.

这个函数接受两个字符串作为参数(按值)，并返回连接它们的结果。通过按值传递参数，函数强制a和b是调用时传递给函数的参数的副本。如果这些是长字符串，这可能意味着仅为函数调用复制大量数据。

But this copy can be avoided altogether if both parameters are made *references*:

但如果两个参数都是引用，则可以完全避免这种复制:

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function. In this regard, the version of `concatenate` taking references is more efficient than the version taking values, since it does not need to copy expensive-to-copy strings.

引用式参数不需要副本。函数直接对作为参数传递的字符串(别名)进行操作，最多可能意味着将某些指针传递给函数。在这方面，接取引用的版本比接取值的版本更有效，因为它不需要复制代价高昂的字符串。

On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.

另一方面，带有引用形参的函数通常被认为是修改传入的参数的函数，因为这就是引用形参实际适用的原因。

The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

解决方案是保证函数的引用参数不会被这个函数修改。这可以通过限定参数为常量来实现:

```
string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

By qualifying them as `const`, the function is forbidden to modify the values of neither `a` nor `b`, but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

通过将它们限定为const，函数既不能修改a也不能修改b的值，但可以作为引用(参数的别名)实际访问它们的值，而不必复制字符串。

Therefore, `const` references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types. That is why they are extremely popular in C++ for arguments of compound types. Note though, that for most fundamental types, there is no noticeable difference in efficiency, and in some cases, const references may even be less efficient!

因此，const引用提供了类似于按值传递参数的功能，但提高了大型类型形参的效率。这就是为什么它们在c++中非常流行，用于复合类型的参数。但是请注意，对于大多数基本类型，在效率上没有明显的区别，在某些情况下，const引用的效率甚至可能更低!

Inline functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

调用函数通常会引起一定的开销(堆叠参数、跳转等)，因此对于非常短的函数，在调用函数的地方简单地插入函数的代码可能更有效，而不是执行正式调用函数的过程。

Preceding a function declaration with the `inline` specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

在函数声明之前加上' inline '说明符，告知编译器对于特定函数，内联展开优于通常的函数调用机制。这根本不会改变函数的行为，只是用来建议编译器在调用函数的每个点插入函数体生成的代码，而不是使用常规的函数调用。

For example, the concatenate function above may be declared inline as:

例如，上面的concatenate函数可以声明为inline:

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

This informs the compiler that when `concatenate` is called, the program prefers the function to be expanded inline, instead of performing a regular call. `inline` is only specified in the function declaration, not when it is called.

这告诉编译器，当调用concatenate时，程序倾向于内联展开函数，而不是执行常规调用。Inline仅在函数声明中指定，而不是在调用时指定。

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the `inline` specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

请注意，大多数编译器在发现有机会提高效率时，即使没有显式地用内联说明符标记，也已经优化代码以生成内联函数。因此，这个说明符仅仅指示编译器，内联是该函数的首选，尽管编译器可以自由地不内联它，或者优化它。在c++中，优化是委托给编译器的任务，只要结果行为是代码指定的行为，编译器就可以自由地生成任何代码。

Default values in parameters

In C++, functions can also have optional parameters, for which no arguments are required in the call, in such a way that, for example, a function with three parameters may be called with only two. For this, the function shall include a default value for its last parameter, which is used by the function when called with fewer arguments. For example:

在c++中，函数也可以有可选形参，在调用时不需要实参，例如，有三个形参的函数可以只用两个来调用。为此，函数应该为其最后一个形参包含一个默认值，该值在调用参数较少的函数时使用。例如:

```
// default values in functions
#include <iostream>
using namespace std;
```

```
int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

In this example, there are two calls to function `divide`. In the first one:

```
divide (12)
```

The call only passes one argument to the function, even though the function has two parameters. In this case, the function assumes the second parameter to be 2 (notice the function definition, which declares its second parameter as `int b=2`). Therefore, the result is 6.

调用只向函数传递一个实参，即使函数有两个形参。在本例中，函数假定第二个形参为2(注意函数定义，它声明第二个形参为' int b=2 ')。因此，结果是6。

In the second call:

```
divide (20,4)
```

The call passes two arguments to the function. Therefore, the default value for `b` (`int b=2`) is ignored, and `b` takes the value passed as argument, that is 4, yielding a result of 5.

调用将两个参数传递给函数。因此，' b '的默认值(' int b=2 ')被忽略，而' b '接受传递的值作为参数，即4，结果为5。

Declaring functions

In C++, identifiers can only be used in expressions once they have been declared. For example, some variable `x` cannot be used before being declared with a statement, such as:

在c++中，标识符只有在声明后才能在表达式中使用。例如，某些变量' x '在用语句声明之前不能使用，例如:

`int x;`

The same applies to functions. Functions cannot be called before they are declared. That is why, in all the previous examples of functions, the functions were always defined before the `main` function, which is the function from where the other functions were called. If `main` were defined before the other functions, this would break the rule that functions shall be declared before being used, and thus would not compile.

同样的道理也适用于函数。函数在声明之前不能被调用。这就是为什么在前面所有函数的例子中，函数总是在main函数之前定义，main函数是调用其他函数的函数。如果' main '是在其他函数之前定义的，这将打破函数必须在使用前声明的规则，因此无法编译。

The prototype of a function can be declared without actually defining the function completely, giving just enough details to allow the types involved in a function call to be known. Naturally, the function shall be defined somewhere else, like later in the code. But at least, once declared like this, it can already be called.

可以在不完全定义函数的情况下声明函数的原型，只提供足够的细节以允许知道函数调用中涉及的类型。当然，函数应该在其他地方定义，比如代码后面的部分。但至少，一旦这样声明，它已经可以被调用了

The declaration shall include all types involved (the return type and the type of its arguments), using the same syntax as used in the definition of the function, but replacing the body of the function (the block of statements) with an ending semicolon.

声明必须包含所有涉及的类型(返回类型及其参数类型)，使用与函数定义中相同的语法，但将函数体(语句块)替换为结束分号。

The parameter list does not need to include the parameter names, but only their types. Parameter names can nevertheless be specified, but they are optional, and do not need to necessarily match those in the function definition. For example, a function called `protofunction` with two int parameters can be declared with either of these statements:

参数列表不需要包含参数名称，而只需要包含它们的类型。参数名仍然可以指定，但它们是可选的，不需要与函数定义中的参数名匹配。例如，一个名为protofunction的函数有两个int形参，可以用以下语句声明:

```
int protofunction (int first, int second);
int protofunction (int, int);
```

Anyway, including a name for each parameter always improves legibility of the declaration.

无论如何，为每个参数包括一个名称总是提高声明的易读性。

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int x);
void even (int x);

int main()
```

```
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

This example is indeed not an example of efficiency. You can probably write yourself a version of this program with half the lines of code. Anyway, this example illustrates how functions can be declared before its definition:

这个例子确实不是效率的例子。你可能只用一半的代码就可以自己编写一个版本的程序。无论如何，这个例子说明了如何在函数定义之前声明函数:

The following lines:

```
void odd (int a);
void even (int a);
```

Declare the prototype of the functions. They already contain all what is necessary to call them, their name, the types of their argument, and their return type (`void` in this case). With these prototype declarations in place, they can be called before they are entirely defined, allowing for example, to place the function from where they are called (`main`) before the actual definition of these functions.

声明函数的原型。它们已经包含了调用它们所需的所有内容、它们的名称、它们的参数类型和它们的返回类型(在本例中为void)。有了这些原型声明，就可以在完全定义它们之前调用它们，例如，允许将调用它们的地方(main)的函数放置在这些函数的实际定义之前。

But declaring functions before being defined is not only useful to reorganize the order of functions within the code. In some cases, such as in this particular case, at least one of the declarations is required, because `odd` and `even` are mutually called; there is a call to `even` in `odd` and a call to `odd` in `even`. And, therefore, there is no way to structure the code so that `odd` is defined before `even`, and `even` before `odd`.

但是，在定义函数之前声明函数不仅有助于在代码中重新组织函数的顺序。在某些情况下，例如在本例中，至少需要一个声明，因为奇数和偶数是相互调用的;在奇数中调用偶数，在偶数中调用奇数。因此，没有办法构造代码使奇数在偶数之前定义，偶数在奇数之前定义。

Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers. For example, in order to obtain the factorial of a number (`n!`) the mathematical formula would be:

递归性是指函数必须被自己调用。它对于某些任务很有用，例如对元素排序或计算数字的阶乘。例如，为了得到一个数(n!)的阶乘，数学公式是:

`n! = n * (n-1) * (n-2) * (n-3) ... * 1`

More concretely, `5!` (factorial of 5) would be:

`5! = 5 * 4 * 3 * 2 * 1 = 120`

And a recursive function to calculate this in C++ could be:

c++中的递归函数可以是:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since, otherwise, the function would perform an infinite recursive loop, in which once it arrived to 0, it would continue multiplying by all the negative numbers (probably provoking a stack overflow at some point during runtime).

注意在阶乘函数包括调用本身,但只有如果传递的参数是大于1,因为,否则,该函数会执行一个无限递归循环,一旦它到达为0,它将继续乘以负数(可能引发堆栈溢出在运行时)。