Name visibility

Scopes

Named entities, such as variables, functions, and compound types need to be declared before being used in C++. The point in the program where this declaration happens influences its visibility:

命名实体，如变量、函数和复合类型，在c++中使用之前需要声明。程序中执行此声明的位置会影响其可见性:

An entity declared outside any block has *global scope*, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function or a selective statement, has *block scope*, and is only visible within the specific block in which it is declared, but not outside it.

在任何块之外声明的实体都具有全局作用域，这意味着它的名称在代码中的任何地方都是有效的。而在块内声明的实体(如函数或选择性语句)具有块作用域，并且只在声明它的特定块内可见，而在块外不可见。

Variables with block scope are known as *local variables*.

块作用域的变量称为局部变量。

For example, a variable declared in the body of a function is a *local variable* that extends until the end of the the function (i.e., until the brace `}` that closes the function definition), but not outside it:

例如，在函数体中声明的变量是一个局部变量，它会扩展到函数的末尾(即，直到大括号}关闭函数定义)，但不会扩展到函数外部:

```
int foo;        // global variable

int some_function ()
{
  int bar;      // local variable
  bar = 0;
}

int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from this function
}
```

In each scope, a name can only represent one entity. For example, there cannot be two variables with the same name in the same scope:

在每个作用域中，一个名称只能代表一个实体。例如，在同一作用域中不能有两个同名的变量:

```
int some_function ()
{
  int x;
  x = 0;
  double x;   // wrong: name already used in this scope
  x = 0.0;
}
```

The visibility of an entity with *block scope* extends until the end of the block, including inner blocks. Nevertheless, an inner block, because it is a different block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside. While outside it, it will still refer to the original entity. For example:

具有块作用域的实体的可见性将一直扩展到块的末尾，包括内部块。然而，由于内部块是不同的块，因此可以重新利用外部作用域中存在的名称来引用不同的实体;在这种情况下，名称将仅在内部块中引用一个不同的实体，隐藏它在外部命名的实体。在它之外，它仍然是指原始实体。例如:

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner scope.
    x = 50;  // sets value to inner x
    y = 50;  // sets value to (outer) y
```

```
    cout << "inner block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "outer block:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

Note that `y` is not hidden in the inner block, and thus accessing `y` still accesses the outer variable.

注意，y并没有隐藏在内部块中，因此访问y仍然访问外部变量。

Variables declared in declarations that introduce a block, such as function parameters and variables declared in loops and conditions (such as those declared on a for or an if) are local to the block they introduce.

在引入块的声明中声明的变量，比如函数形参和在循环中声明的变量，以及条件(比如在for或if中声明的变量)是它们引入的块的局部变量。

Namespaces

Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc...

在特定的范围内，只有一个具有特定名称的实体可以存在。对于局部名称来说，这很少是一个问题，因为块往往是相对较短的，名称在其中有特定的用途，如命名计数器变量、参数等.....

But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

但是非局部名称带来了更多的名称冲突的可能性，特别是考虑到库可能声明许多函数、类型和变量，它们本质上都不是局部的，其中一些非常泛型。

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

名称空间允许我们将具有全局作用域的命名实体分组到更窄的作用域，从而赋予它们名称空间作用域。这允许将程序的元素组织到通过名称引用的不同逻辑范围中。

The syntax to declare a namespaces is:

声明命名空间的语法是:

```
namespace identifier
{
 named_entities
}
```

Where `identifier` is any valid identifier and `named_entities` is the set of variables, types and functions that are included within the namespace. For example:

其中' identifier '是任何有效的标识符，' named_entities '是命名空间中包含的变量、类型和函数的集合。例如:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`.

在本例中，变量a和b是在名为myNamespace的命名空间中声明的普通变量。

These variables can be accessed from within their namespace normally, with their identifier (either `a` or `b`), but if accessed from outside the `myNamespace` namespace they have to be properly qualified with the scope operator `::`. For example, to access the previous variables from outside `myNamespace` they should be qualified like:

这些变量通常可以通过其标识符(a或b)从名称空间内部访问，但如果从myNamespace名称空间外部访问，则必须使用作用域操作符::对它们进行适当的限定。例如，要从myNamespace外部访问前面的变量，它们应该像这样限定:

```
myNamespace::a
myNamespace::b
```

Namespaces are particularly useful to avoid name collisions. For example:

名称空间对于避免名称冲突特别有用。例如:

```
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}
```

```
int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

In this case, there are two functions with the same name: `value`. One is defined within the namespace `foo`, and the other one in `bar`. No redefinition errors happen thanks to namespaces. Notice also how `pi` is accessed in an unqualified manner from within namespace `bar` (just as `pi`), while it is again accessed in `main`, but here it needs to be qualified as `bar::pi`.

在本例中，有两个同名的函数:value。一个在命名空间foo中定义，另一个在bar中定义。由于名称空间，没有发生重新定义错误。还要注意pi是如何在命名空间栏中以非限定方式访问的(就像pi一样)，而在main中再次访问它，但这里需要限定为bar::pi。

Namespaces can be split: Two segments of a code can be declared in the same namespace:

命名空间可以拆分:可以在同一个命名空间中声明两段代码:

```
namespace foo { int a; }
namespace bar { int b; }
namespace foo { int c; }
```

This declares three variables: `a` and `c` are in namespace `foo`, while `b` is in namespace `bar`. Namespaces can even extend across different translation units (i.e., across different files of source code).

这声明了三个变量:a和c在命名空间foo中，而b在命名空间bar中。名称空间甚至可以跨不同的翻译单元扩展(例如，跨不同的源代码文件)。

using

The keyword `using` introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

using关键字将名称引入到当前声明区域(如块)，从而避免了对名称进行限定的需要。例如:

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

namespace first
{
  int x = 5;
  int y = 10;
}
```

```
namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

Notice how in `main`, the variable `x` (without any name qualifier) refers to `first::x`, whereas `y` refers to `second::y`, just as specified by the `using` declarations. The variables `first::y` and `second::x` can still be accessed, but require fully qualified names.

注意在main中，变量x(没有任何名称限定符)是如何指向first::x的，而y指向second::y的，正如using声明所指定的。变量first::y和second::x仍然可以访问，但需要完全限定名。

The keyword `using` can also be used as a directive to introduce an entire namespace:

关键字using也可以作为一个指令来引入整个命名空间:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
```

```
  cout << second::y << '\n';
  return 0;
}
```

In this case, by declaring that we were using namespace `first`, all direct uses of `x` and `y` without name qualifiers were also looked up in namespace `first`.

在本例中，通过声明我们先使用名称空间，所有没有名称限定符的x和y的直接使用也都是先在名称空间中查找的。

`using` and `using namespace` have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

使用和使用名称空间仅在声明它们的同一块中有效，如果它们直接在全局作用域中使用，则在整个源代码文件中有效。例如，可以先使用一个名称空间的对象，然后再使用另一个名称空间的对象，方法是将代码分割到不同的块中:

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

Namespace aliasing

Existing namespaces can be aliased with new names, with the following syntax:

现有名称空间可以使用新名称进行别名，语法如下:

```
namespace new_name = current_name;
```

The std namespace

All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the `std` namespace. Most examples in these tutorials, in fact, include the following line:

标准c++库的所有实体(变量、类型、常量和函数)都在std命名空间中声明。事实上，这些教程中的大多数示例都包含以下代码行:

```
using namespace std;
```

This introduces direct visibility of all the names of the `std` namespace into the code. This is done in these tutorials to facilitate comprehension and shorten the length of the examples, but many programmers prefer to qualify each of the elements of the standard library used in their programs. For example, instead of:

这在代码中引入了std命名空间的所有名称的直接可见性。在这些教程中这样做是为了便于理解并缩短示例的长度，但是许多程序员更喜欢限定他们程序中使用的标准库的每个元素。例如，代替:

```
cout << "Hello world!";
```

It is common to instead see:

```
std::cout << "Hello world!";
```

Whether the elements in the `std` namespace are introduced with `using` declarations or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

无论std名称空间中的元素是通过using声明引入的，还是每次使用都是完全限定的，都不会以任何方式改变结果程序的行为或效率。这主要是一个风格偏好的问题，尽管对于混合库的项目，显式限定倾向于首选。

Storage classes

The storage for variables with *global* or *namespace scope* is allocated for the entire duration of the program. This is known as *static storage*, and it contrasts with the storage for *local variables* (those declared within a block). These use what is known as automatic storage. The storage for local variables is only available during the block in which they are declared; after that, that same storage may be used for a local variable of some other function, or used otherwise.

具有全局或命名空间作用域的变量的存储空间是在程序的整个过程中分配的。这被称为静态存储，与局部变量(在块中声明的变量)的存储不同。它们使用的是所谓的自动存储。局部变量的存储仅在声明它们的块中可用;在此之后，可以将相同的存储空间用于其他函数的局部变量，或用于其他用途。

But there is another substantial difference between variables with *static storage* and variables with *automatic storage*:

但是静态存储的变量和自动存储的变量之间还有一个实质性的区别:

- Variables with *static storage* (such as global variables) that are not explicitly initialized are automatically initialized to zeroes.

未显式初始化的静态存储变量(如全局变量)将自动初始化为零。

- Variables with *automatic storage* (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

没有显式初始化的自动存储变量(如局部变量)将保持未初始化，因此具有未知值。

For example:

```
// static vs automatic storage
#include <iostream>
using namespace std;

int x;

int main ()
{
  int y;
  cout << x << '\n';
  cout << y << '\n';
  return 0;
}
```

The actual output may vary, but only the value of `x` is guaranteed to be zero. `y` can actually contain just about any value (including zero).

实际输出可能会有所不同，但只有' x '的值保证为零。' y '实际上可以包含任何值(包括零)。