C++ 修饰符类型 C++ 运算符

C++ 存储类

存储类定义 C++ 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C++ 程序中可用的存储类：

- auto
- register
- static
- extern
- mutable
- thread_local (C++11)

从 C++ 11 开始，auto 关键字不再是 C++ 存储类说明符，且 register 关键字被弃用。

auto 存储类

自 C++ 11 以来，**auto** 关键字用于两种情况：声明变量时根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符。

C++98标准中auto关键字用于自动变量的声明，但由于使用极少且多余，在C++11中已删除这一用法。

根据初始化表达式自动推断被声明的变量的类型，如：

```
auto f=3.14; //double
auto s("hello"); //const char*
auto z = new auto(9); // int*
auto x1 = 5, x2 = 5.0, x3='r';//错误，必须是初始化为同一类型
```

register 存储类

register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

```
{
register int miles;
}
```

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

static 存储类

static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

在 C++ 中，当 static 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

实例

```
#include <iostream>
// 函数声明
void func(void);
static int count = 10; /* 全局变量 */
int main()
{
while(count--)
{
func();
}
return 0;
}
// 函数定义
void func( void )
{
static int i = 5; // 局部静态变量
i++;
std::cout << "变量 i 为 " << i ;
std::cout << " , 变量 count 为 " << count << std::endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
变量 i 为 6 , 变量 count 为 9
变量 i 为 7 , 变量 count 为 8
变量 i 为 8 , 变量 count 为 7
变量 i 为 9 , 变量 count 为 6
变量 i 为 10 , 变量 count 为 5
变量 i 为 11 , 变量 count 为 4
变量 i 为 12 , 变量 count 为 3
变量 i 为 13 , 变量 count 为 2
变量 i 为 14 , 变量 count 为 1
变量 i 为 15 , 变量 count 为 0
```

extern 存储类

extern 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：main.cpp

实例

```
#include <iostream>
int count ;
extern void write_extern();
int main()
{
count = 5;
write_extern();
}
```

第二个文件：support.cpp

实例

```
#include <iostream>
extern int count;
void write_extern(void)
{
std::cout << "Count is " << count << std::endl;
}
```

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.cpp 中定义的 count。现在 ，编译这两个文件，如下所示：

```
$ g++ main.cpp support.cpp -o write
```

这会产生 **write** 可执行程序，尝试执行 **write**，它会产生下列结果：

```
$ ./write
Count is 5
```

mutable 存储类

mutable 说明符仅适用于类的对象，这将在本教程的最后进行讲解。它允许对象的成员替代常量。也就是说，mutable 成员可以通过 const 成员函数修改。

thread_local 存储类

使用 thread_local 说明符声明的变量仅可在它在其上创建的线程上访问。 变量在创建线程时创建，并在销毁线程时销毁。 每个线程都有其自己的变量副本。

thread_local 说明符可以与 static 或 extern 合并。

可以将 thread_local 仅应用于数据声明和定义，thread_local 不能用于函数声明或定义。

以下演示了可以被声明为 thread_local 的变量：

```
thread_local int x; // 命名空间下的全局变量
class X
{
static thread_local std::string s; // 类的static成员变量
};
```

```
static thread_local std::string X::s; // X::s 是需要定义的
void foo()
{
thread_local std::vector<int> v; // 本地变量
}
```

C++ 修饰符类型 C++ 运算符