

C/C++ 中 static 的用法全局变量与局部变量

分类 [编程技术](#)

1. 什么是static?

static 是 C/C++ 中很常用的修饰符，它被用来控制变量的存储方式和可见性。

1.1 static 的引入

我们知道在函数内部定义的变量，当程序执行到它的定义处时，编译器为它在栈上分配空间，函数在栈上分配的空间在此函数执行结束时会释放掉，这样就产生了一个问题: 如果想将函数中此变量的值保存至下一次调用时，如何实现？最容易想到的方法是定义为全局的变量，但定义一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅只受此函数控制）。static 关键字则可以很好的解决这个问题。

另外，在 C++ 中，需要一个数据对象为整个类而非某个对象服务,同时又力求不破坏类的封装性,即要求此成员隐藏在类的内部，对外不可见时，可将其定义为静态数据。

1.2 静态数据的存储

全局（静态）存储区：分为 DATA 段和 BSS 段。DATA 段（全局初始化区）存放初始化的全局变量和静态变量；BSS 段（全局未初始化区）存放未初始化的全局变量和静态变量。程序运行结束时自动释放。其中BBS段在程序执行之前会被系统自动清0，所以未初始化的全局变量和静态变量在程序执行之前已经为0。存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。

在 C++ 中 static 的内部实现机制：静态数据成员要在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所以静态数据成员不能在任何函数内分配空间和初始化。

这样，它的空间分配有三个可能的地方，一是作为类的外部接口的头文件，那里有类声明；二是类定义的内部实现，那里有类的成员函数定义；三是应用程序的 main() 函数前的全局数据声明和定义处。

教程列表

ADO 教程	Ajax 教程	Android 教
Angular2	AngularJS	AppML 教
ASP 教程	ASP.NET	Bootstrap
Bootstrap4	Bootstrap5	C 教程
C# 教程	C++ 教程	Chart.js 教
CSS 参考	CSS 教程	CSS3 教程
Django 教	Docker 教	DTD 教程
ECharts 教	Eclipse 教	FastAPI 教
Firebug 教	Font	Foundation
Git 教程	Go 语言教	Google 地
Highcharts	HTML	HTML 参考
HTML 字符	HTML 教程	HTTP 教程
ionic 教程	iOS 教程	Java 教程
JavaScript	Javascript	jQuery
jQuery	jQuery UI	jQuery 教
JSON 教程	JSP 教程	Julia 教程
Kotlin 教程	Linux 教程	Lua 教程
Markdown	Matplotlib	Maven 教
Memcached	MongoDB	MySQL 教
Node.js 教	NumPy 教	Pandas 教
Perl 教程	PHP 教程	PostgreSQL
Python 3	Python 基	Python 量
R 教程	RDF 教程	React 教程
Redis 教程	RSS 教程	Ruby 教程
Rust 教程	Sass 教程	Scala 教程
SciPy 教程	Servlet 教	SOAP 教程
SQL 教程	SQLite 教	SVG 教程
SVN 教程	Swift 教程	TCP/IP 教

静态数据成员要实际地分配空间，故不能在类的声明中定义（只能声明数据成员）。类声明只声明一个类的"尺寸和规格"，并不进行实际的内存分配，所以在类声明中写成定义是错误的。它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。

static 被引入以告知编译器，将变量存储在程序的静态存储区而非栈上空间，静态数据成员按定义出现的先后顺序依次初始化，注意静态成员嵌套时，要保证所嵌套的成员已经初始化了。消除时的顺序是初始化的反顺序。

优势：可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。

2. 在 C/C++ 中static的作用

2.1 总的来说

- （1）在修饰变量的时候，static 修饰的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放。
- （2）static 修饰全局变量的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是 extern 外部声明也不可以。
- （3）static 修饰一个函数，则这个函数的只能在本文件中调用，不能被其他文件调用。static 修饰的变量存放在全局数据区的静态变量区，包括全局静态变量和局部静态变量，都在全局数据区分配内存。初始化的时候自动初始化为 0。
- （4）不想被释放的时候，可以使用static修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束释放可以使用 static 修饰。
- （5）考虑到数据安全性（当程序想要使用全局变量的时候应该先考虑使用 static）。

2.2 静态变量与普通变量

静态全局变量有以下特点：

- （1）静态变量都在全局数据区分配内存，包括后面将要提到的静态局部变量；
- （2）未经初始化的静态全局变量会被程序自动初始化为0（在函数体内声明的自动变量的值是随机的，除非它被显式初始化，而在函数体外被声明的自动变量也会被初始化为 0）；

TypeScript	VBScript	Vue.js 教程
Vue3 教程	W3C 教程	Web
WSDL 教	XLink 教程	XML DOM
XML	XML 教程	XPath 教程
XQuery 教	XSLFO 教	XSLT 教程
数据结构	正则表达式	测验
浏览器	网站品质	网站建设指
网站服务器	设计模式	

- (3) 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的。

优点：静态全局变量不能被其它文件所用；其它文件中可以定义相同名字的变量，不会发生冲突。

(1) 全局变量和全局静态变量的区别

- 1) 全局变量是不显式用 `static` 修饰的全局变量，全局变量默认是有外部链接性的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 `extern` 全局变量名的声明，就可以使用全局变量。
- 2) 全局静态变量是显式用 `static` 修饰的全局变量，作用域是声明此变量所在的文件，其他的文件即使用 `extern` 声明也不能使用。

2.3 静态局部变量有以下特点：

- (1) 该变量在全局数据区分配内存；
- (2) 静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；
- (3) 静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；
- (4) 它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束。

一般程序把新产生的动态数据存放在堆区，函数内部的自动变量存放在栈区。自动变量一般会随着函数的退出而释放空间，静态数据（即使是函数内部的静态局部变量）也存放在全局数据区。全局数据区的数据并不会因为函数的退出而释放空间。

看下面的例子：

实例

```
//example:
#include <stdio.h>
#include <stdlib.h>
int k1 = 1;
int k2;
static int k3 = 2;
static int k4;
int main()
{
    static int m1 = 2, m2;
    int i = 1;
    char*p;
    char str[10] = "hello";
    char*q = "hello";
```

```

p = (char *)malloc(100);
free(p);
printf("栈区-变量地址    i: %p\n", &i);
printf("栈区-变量地址    p: %p\n", &p);
printf("栈区-变量地址 str: %p\n", str);
printf("栈区-变量地址    q: %p\n", &q);
printf("堆区地址-动态申请: %p\n", p);
printf("全局外部有初值 k1: %p\n", &k1);
printf("    外部无初值 k2: %p\n", &k2);
printf("静态外部有初值 k3: %p\n", &k3);
printf("    外静无初值 k4: %p\n", &k4);
printf("    内静态有初值 m1: %p\n", &m1);
printf("    内静态无初值 m2: %p\n", &m2);
printf("        文字常量地址: %p, %s\n", q, q);
printf("        程序区地址: %p\n", &main);
return 0;
}

```

输出结果如下：

```

栈区-变量地址    i: 00F0FEA8
栈区-变量地址    p: 00F0FE9C
栈区-变量地址 str: 00F0FE88
栈区-变量地址    q: 00F0FE7C
堆区地址-动态申请: 0122E9F8
全局外部有初值 k1: 002B9034
    外部无初值 k2: 002B9500
静态外部有初值 k3: 002B9038
    外静无初值 k4: 002B9504
    内静态有初值 m1: 002B903C
    内静态无初值 m2: 002B9508
        文字常量地址: 002B6BD4, hello
        程序区地址: 002B12B2

```

3. static 用法

3.1 在 C++ 中

static 关键字最基本的用法是：

- 1、被 static 修饰的变量属于类变量，可以通过 **类名.变量名** 直接引用，而不需要 new 出一个类来
- 2、被 static 修饰的方法属于类方法，可以通过 **类名.方法名** 直接引用，而不需要 new 出一个类来

被 static 修饰的变量、被 static 修饰的方法统一属于类的静态资源，是类实例之间共享的，换言之，一处变、处处变。

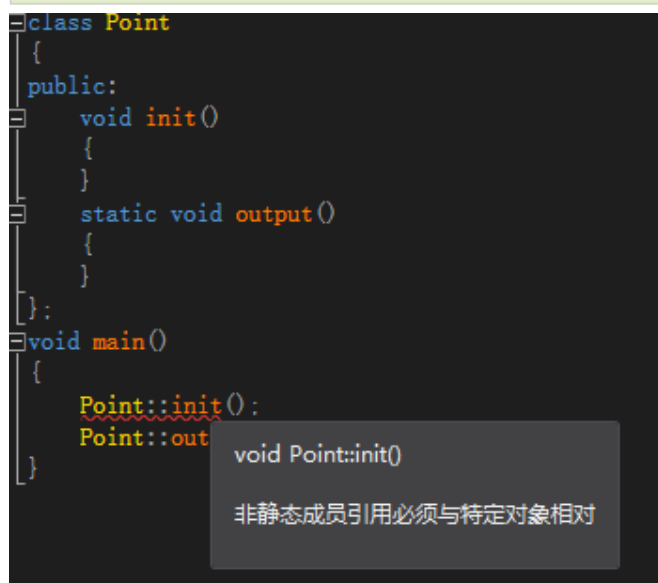
在 C++ 中，静态成员是属于整个类的而不是某个对象，静态成员变量只存储一份供所有对象共用。所以在所有对象中都可以共享它。使用静态成员变量实现多个对象之间的数据共享不会破坏隐藏的原则，保证了安全性还可以节省内存。

静态成员的定义或声明要加个关键 `static`。静态成员可以通过双冒号来使用即 `<类名>::<静态成员名>`。

3.2 静态类相关

通过类名调用静态成员函数和非静态成员函数:

```
class Point
{
public:
    void init()
    {
    }
    static void output()
    {
    }
};
void main()
{
    Point::init();
    Point::output();
}
```



```
class Point
{
public:
    void init()
    {
    }
    static void output()
    {
    }
};
void main()
{
    Point::init();
    Point::output();
}
```

void Point::init()
非静态成员引用必须与特定对象相对

报错:

```
'Point::init' : illegal call of non-static member function
```

结论 1: 不能通过类名来调用类的非静态成员函数。

通过类的对象调用静态成员函数和非静态成员函数。

```
class Point
{
public:
    void init()
    {
    }
    static void output()
    {
    }
};
```

```

    {
    }
};
void main()
{
    Point pt;
    pt.init();
    pt.output();
}

```

编译通过。

结论 2：类的对象可以使用静态成员函数和非静态成员函数。

在类的静态成员函数中使用类的非静态成员。

```

#include <stdio.h>
class Point
{
public:
    void init()
    {
    }
    static void output()
    {
        printf("%d\n", m_x);
    }
private:
    int m_x;
};
void main()
{
    Point pt;
    pt.output();
}

```

编译出错：

```

error C2597: illegal reference to data member 'Point::m_x'
in a static member function

```

因为静态成员函数属于整个类，在类实例化对象之前就已经分配空间了，而类的非静态成员必须在类实例化对象后才有内存空间，所以这个调用就出错了，就好比没有声明一个变量却提前使用它一样。

结论3：静态成员函数中不能引用非静态成员。

在类的非静态成员函数中使用类的静态成员。

```

class Point
{
public:
    void init()
    {
    }
}

```

```

        output();
    }
    static void output()
    {
    }
};
void main()
{
    Point pt;
    Pt.init();
    pt.output();
}

```

编译通过。

结论 4：类的非静态成员函数可以调用用静态成员函数，但反之不能。

使用类的静态成员变量。

```

#include <stdio.h>
class Point
{
public:
    Point()
    {
        m_nPointCount++;
    }
    ~Point()
    {
        m_nPointCount--;
    }
    static void output()
    {
        printf("%d\n", m_nPointCount);
    }
private:
    static int m_nPointCount;
};
void main()
{
    Point pt;
    pt.output();
}

```

按 **Ctrl+F7** 编译无错误，按 F7 生成 EXE 程序时报链接错误。

```

error LNK2001: unresolved external symbol "private: static
int Point::m_nPointCount" (?m_nPointCount@Point@@0HA)

```

这是因为类的静态成员变量在使用前必须先初始化。

在 **main()** 函数前加上 **int Point::m_nPointCount = 0;** 再编译链接无错误，运行程序将输出 1。

结论 5：类的静态成员变量必须先初始化再使用。

思考总结：静态资源属于类，但是是独立于类存在的。从 J 类的加载机制的角度讲，静态资源是类初始化的时候加载的，而非静态资源是类实例化对象的时候加载的。类的初始化早于类实例化对象，比如 `Class.forName("xxx")` 方法，就是初始化了一个类，但是并没有实例化对象，只是加载这个类的静态资源罢了。所以对于静态资源来说，它是不可能知道一个类中有哪些非静态资源的；但是对于非静态资源来说就不一样了，由于它是实例化对象出来之后产生的，因此属于类的这些东西它都能认识。所以上面的几个问题答案就很明确了：

- 1) 静态方法能不能引用非静态资源？不能，实例化对象的时候才会产生的东西，对于初始化后就存在的静态资源来说，根本不认识它。
- 2) 静态方法里面能不能引用静态资源？可以，因为都是类初始化的时候加载的，大家相互都认识。
- 3) 非静态方法里面能不能引用静态资源？可以，非静态方法就是实例方法，那是实例化对象之后才产生的，那么属于类的内容它都认识。

(static 修饰类：这个用得相对比前面的用法少多了，static 一般情况来说是不可以修饰类的，如果 static 要修饰一个类，说明这个类是一个静态内部类（注意 static 只能修饰一个内部类），也就是匿名内部类。像线程池 `ThreadPoolExecutor` 中的四种拒绝机制 `CallerRunsPolicy`、`AbortPolicy`、`DiscardPolicy`、`DiscardOldestPolicy` 就是静态内部类。静态内部类相关内容会在写内部类的时候专门讲到。)

3.3 总结：

- (1) 静态成员函数中不能调用非静态成员。
- (2) 非静态成员函数中可以调用静态成员。因为静态成员属于类本身，在类的对象产生之前就已经存在了，所以在非静态成员函数中是可以调用静态成员的。
- (3) 静态成员变量使用前必须先初始化(如 `int MyClass::m_nNumber = 0;`)，否则会在 linker 时出错。

一般总结：在类中，static 可以用来修饰静态数据成员和静态成员方法。

静态数据成员

- (1) 静态数据成员可以实现多个对象之间的数据共享，它是类的所有对象的共享成员，它在内存中只占一份空间，如果改变它的值，则各对象中这个数据成员的值都被改变。
- (2) 静态数据成员是在程序开始运行时被分配空间，到程序结束之后才释放，只要类中指定了静态数据成员，即使不定义对象，也会为静态数据成

员分配空间。

- (3) 静态数据成员可以被初始化，但是只能在类体外进行初始化，若未对静态数据成员赋初值，则编译器会自动为其初始化为 0。
- (4) 静态数据成员既可以通过对象名引用，也可以通过类名引用。

静态成员函数

- (1) 静态成员函数和静态数据成员一样，他们都属于类的静态成员，而不是对象成员。
- (2) 非静态成员函数有 this 指针，而静态成员函数没有 this 指针。
- (3) 静态成员函数主要用来方位静态数据成员而不能访问非静态成员。

再给一个利用类的静态成员变量和函数的例子以加深理解，这个例子建立一个学生类，每个学生类的对象将组成一个双向链表，用一个静态成员变量记录这个双向链表的表头，一个静态成员函数输出这个双向链表。

实例

```
#include <stdio.h>
#include <string.h>
const int MAX_NAME_SIZE = 30;

class Student
{
public:
    Student(char *pszName);
    ~Student();
public:
    static void PrintfAllStudents();
private:
    char    m_name[MAX_NAME_SIZE];
    Student *next;
    Student *prev;
    static Student *m_head;
};

Student::Student(char *pszName)
{
    strcpy(this->m_name, pszName);

    //建立双向链表，新数据从链表头部插入。
    this->next = m_head;
    this->prev = NULL;
    if (m_head != NULL)
        m_head->prev = this;
    m_head = this;
}
```

```

Student::~~Student ()//析构过程就是节点的脱离过程
{
    if (this == m_head) //该节点就是头节点。
    {
        m_head = this->next;
    }
    else
    {
        this->prev->next = this->next;
        this->next->prev = this->prev;
    }
}

void Student::PrintfAllStudents()
{
    for (Student *p = m_head; p != NULL; p = p->next)
        printf("%s\n", p->m_name);
}

Student* Student::m_head = NULL;

void main()
{
    Student studentA("AAA");
    Student studentB("BBB");
    Student studentC("CCC");
    Student studentD("DDD");
    Student student("MoreWindows");
    Student::PrintfAllStudents();
}

```

程序将输出：

```

MoreWindows
DDD
CCC
BBB
AAA
Press any key to continue

```

原文地址：<https://www.cnblogs.com/33debug/p/7223869.html>

← Java 正则表达式的捕获组(capture group)

配置 redis 外网可访问 →

📄 点我分享笔记