# C++ 中的类型限定符

类型限定符提供了变量的额外信息。

|  |  |
| --- | --- |
| **限定符** | **含义** |
| const | **const** 类型的对象在程序执行期间不能被修改改变。 |
| volatile | 修饰符 **volatile** 告诉编译器不需要优化volatile声明的变量，  让程序可以直接从内存中读取变量。对于一般的变量编译器会对变量进行优化，  将内存中的变量值放在寄存器中以加快读写效率。 |
| restrict | 由 **restrict** 修饰的指针是唯一一种访问它所指向的对象的方式。  只有 C99 增加了新的类型限定符 restrict。 |

# 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++ 17 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。

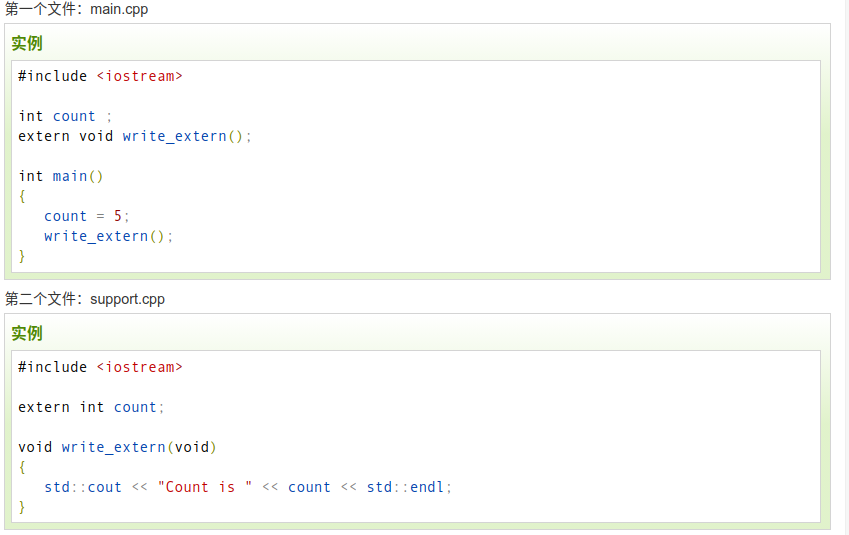
在 C++ 中，当 static 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

## 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，



## 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。

# 杂项运算符

下表列出了 C++ 支持的其他一些重要的运算符。

|  |  |
| --- | --- |
| **运算符** | **描述** |
| sizeof | [sizeof 运算符](https://www.runoob.com/cplusplus/cpp-sizeof-operator.html)返回变量的大小。例如，sizeof(a) 将返回 4，其中 a 是整数。 |
| Condition ? X : Y | [条件运算符](https://www.runoob.com/cplusplus/cpp-conditional-operator.html)。如果 Condition 为真 ? 则值为 X : 否则值为 Y。 |
| , | [逗号运算符](https://www.runoob.com/cplusplus/cpp-comma-operator.html)会顺序执行一系列运算。  整个逗号表达式的值是以逗号分隔的列表中的最后一个表达式的值。 |
| .（点）和 ->（箭头） | [成员运算符](https://www.runoob.com/cplusplus/cpp-member-operators.html)用于引用类、结构和共用体的成员。 |
| Cast | [强制转换运算符](https://www.runoob.com/cplusplus/cpp-casting-operators.html)把一种数据类型转换为另一种数据类型。例如，int(2.2000) 将返回 2。 |
| & | [指针运算符 &](https://www.runoob.com/cplusplus/cpp-pointer-operators.html) 返回变量的地址。例如 &a; 将给出变量的实际地址。 |
| \* | [指针运算符 \*](https://www.runoob.com/cplusplus/cpp-pointer-operators.html) 指向一个变量。例如，\*var; 将指向变量 var。 |

## C++ 中的运算符优先级

|  |  |  |
| --- | --- | --- |
| **类别** | **运算符** | **结合性** |
| 后缀 | () [] -> . ++ - - | 从左到右 |
| 一元 | + - ! ~ ++ - - (type)\* & sizeof | 从右到左 |
| 乘除 | \* / % | 从左到右 |
| 加减 | + - | 从左到右 |
| 移位 | << >> | 从左到右 |
| 关系 | < <= > >= | 从左到右 |
| 相等 | == != | 从左到右 |
| 位与 AND | & | 从左到右 |
| 位异或 XOR | ^ | 从左到右 |
| 位或 OR | | | 从左到右 |
| 逻辑与 AND | && | 从左到右 |
| 逻辑或 OR | || | 从左到右 |
| 条件 | ?: | 从右到左 |
| 赋值 | = += -= \*= /= %=>>= <<= &= ^= |= | 从右到左 |
| 逗号 | , | 从左到右 |

# C++ goto 语句

**goto** 语句允许把控制无条件转移到同一函数内的被标记的语句。

**注意：**在任何编程语言中，都不建议使用 goto 语句。因为它使得程序的控制流难以跟踪，使程序难以理解和难以修改。任何使用 goto 语句的程序可以改写成不需要使用 goto 语句的写法。

## 语法

C++ 中 **goto**语句的语法：

goto label;...

label: statement;

在这里，**label** 是识别被标记语句的标识符，可以是任何除 C++ 关键字以外的纯文本。标记语句可以是任何语句，放置在标识符和冒号（:）后边。

goto 语句一个很好的作用是退出深嵌套例程。一个简单的 **break** 语句在这里不会起到作用，因为它只会使程序退出最内层循环。

# 参数的默认值

当您定义一个函数，您可以为参数列表中后边的每一个参数指定默认值。当调用函数时，如果实际参数的值留空，则使用这个默认值。

int sum(int a, int b=20)

# C++ 数学运算

在 C++ 中，除了可以创建各种函数，还包含了各种有用的函数供您使用。这些函数写在标准 C 和 C++ 库中，叫做**内置**函数。您可以在程序中引用这些函数。

C++ 内置了丰富的数学函数，可对各种数字进行运算。下表列出了 C++ 中一些有用的内置的数学函数。

为了利用这些函数，您需要引用数学头文件 **<cmath>**。

|  |  |
| --- | --- |
| **序号** | **函数 & 描述** |
| 1 | **double cos(double);** 该函数返回弧度角（double 型）的余弦。 |
| 2 | **double sin(double);** 该函数返回弧度角（double 型）的正弦。 |
| 3 | **double tan(double);** 该函数返回弧度角（double 型）的正切。 |
| 4 | **double log(double);** 该函数返回参数的自然对数。 |
| 5 | **double pow(double, double);** 假设第一个参数为 x，第二个参数为 y，则该函数返回 x 的 y 次方。 |
| 6 | **double hypot(double, double);** 该函数返回两个参数的平方总和的平方根，也就是说，参数为一个直角三角形的两个直角边，  函数会返回斜边的长度。 |
| 7 | **double sqrt(double);** 该函数返回参数的平方根。 |
| 8 | **int abs(int);** 该函数返回整数的绝对值。 |
| 9 | **double fabs(double);** 该函数返回任意一个浮点数的绝对值。 |
| 10 | **double floor(double);** 该函数返回一个小于或等于传入参数的最大整数。 |

# C++ 随机数

在许多情况下，需要生成随机数。关于随机数生成器，有两个相关的函数。一个是 **rand()**，该函数只返回一个伪随机数。生成随机数之前必须先调用 **srand()** 函数。

下面是一个关于生成随机数的简单实例。实例中使用了 **time()** 函数来获取系统时间的秒数，通过调用 rand() 函数来生成随机数：

#include <ctime>

#include <cstdlib>

srand()为初始化随机数发生器，用于设置rand()产生随机数时的种子。传入的参数seed为unsigned int类型，通常我们会使用time(0)或time(NULL)的返回值作为seed。

# C++ 从函数返回数组

C++ 不允许返回一个完整的数组作为函数的参数。但是，您可以通过指定不带索引的数组名来返回一个指向数组的指针。

如果您想要从函数返回一个一维数组，您必须声明一个返回指针的函数，

另外，C++ 不支持在函数外返回局部变量的地址，除非定义局部变量为 **static** 变量。

# 字符串操作函数

C++ 中有大量的函数用来操作以 null 结尾的字符串：supports a wide range of functions that manipulate null-terminated strings:

#include <cstring>

|  |  |
| --- | --- |
| **序号** | **函数 & 目的** |
| 1 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。 |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回值小于 0；如果 s1>s2 则返回值大于 0。 |
| 5 | **strchr(s1, ch);** 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。 |
| 6 | **strstr(s1, s2);** 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。 |

# C++ 中的 String 类

C++ 标准库提供了 **string** 类类型，支持上述所有的操作，另外还增加了其他更多的功能。

// 复制 str1 到 str3

str3 = str1;

// 连接 str1 和 str2

str3 = str1 + str2;

// 连接后，str3 的总长度

len = str3.size();

# C++ Null 指针

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为**空**指针。

# C++ 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。

## C++ 引用 vs 指针

引用很容易与指针混淆，它们之间有三个主要的不同：

* 不存在空引用。引用必须连接到一块合法的内存。
* 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
* 引用必须在创建时被初始化。指针可以在任何时间被初始化。

## C++ 中创建引用

试想变量名称是变量附属在内存位置中的标签，您可以把引用当成是变量附属在内存位置中的第二个标签。因此，您可以通过原始变量名称或引用来访问变量的内容。

int i = 17;

我们可以为 i 声明引用变量，如下所示：

int& r = i;double& s = d;

在这些声明中，& 读作**引用**。因此，第一个声明可以读作 "r 是一个初始化为 i 的整型引用"，第二个声明可以读作 "s 是一个初始化为 d 的 double 型引用"。

# C++ 把引用作为参数

// 函数声明 void swap(int& x, int& y);

# C++ 把引用作为返回值

通过使用引用来替代指针，会使 C++ 程序更容易阅读和维护。C++ 函数可以返回一个引用，方式与返回一个指针类似。

当函数返回一个引用时，则返回一个指向返回值的隐式指针。这样，函数就可以放在赋值语句的左边。

double& setValues( int i )；

当返回一个引用时，要注意被引用的对象不能超出作用域。所以返回一个对局部变量的引用是不合法的，但是，可以返回一个对静态变量的引用。

# C++ 日期 & 时间

C++ 标准库没有提供所谓的日期类型。C++ 继承了 C 语言用于日期和时间操作的结构和函数。为了使用日期和时间相关的函数和结构，需要在 C++ 程序中引用 <ctime> 头文件。

有四个与时间相关的类型：**clock\_t、time\_t、size\_t** 和 **tm**。类型 clock\_t、size\_t 和 time\_t 能够把系统时间和日期表示为某种整数。

结构类型 **tm** 把日期和时间以 C 结构的形式保存，tm 结构的定义如下：

struct tm {

int tm\_sec; // 秒，正常范围从 0 到 59，但允许至 61

int tm\_min; // 分，范围从 0 到 59

int tm\_hour; // 小时，范围从 0 到 23

int tm\_mday; // 一月中的第几天，范围从 1 到 31

int tm\_mon; // 月，范围从 0 到 11

int tm\_year; // 自 1900 年起的年数

int tm\_wday; // 一周中的第几天，范围从 0 到 6，从星期日算起

int tm\_yday; // 一年中的第几天，范围从 0 到 365，从 1 月 1 日算起

int tm\_isdst; // 夏令时

}

下面是 C/C++ 中关于日期和时间的重要函数。所有这些函数都是 C/C++ 标准库的组成部分，您可以在 C++ 标准库中查看一下各个函数的细节。

|  |  |
| --- | --- |
| **序号** | **函数 & 描述** |
| 1 | **[time\_t time(time\_t \*time);](https://www.runoob.com/cplusplus/c-function-time.html)** 该函数返回系统的当前日历时间，自 1970 年 1 月 1 日以来经过的秒数。如果系统没有时间，则返回 .1。 |
| 2 | **[char \*ctime(const time\_t \*time);](https://www.runoob.com/cplusplus/c-function-ctime.html)** 该返回一个表示当地时间的字符串指针，字符串形式 *day month year hours:minutes:seconds year\n\0*。 |
| 3 | **[struct tm \*localtime(const time\_t \*time);](https://www.runoob.com/cplusplus/c-function-localtime.html)** 该函数返回一个指向表示本地时间的 **tm** 结构的指针。 |
| 4 | **[clock\_t clock(void);](https://www.runoob.com/cplusplus/c-function-clock.html)** 该函数返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。如果时间不可用，则返回 .1。 |
| 5 | **[char \* asctime ( const struct tm \* time );](https://www.runoob.com/cplusplus/c-function-asctime.html)** 该函数返回一个指向字符串的指针，字符串包含了 time 所指向结构中存储的信息，  返回形式为：day month date hours:minutes:seconds year\n\0。 |
| 6 | **[struct tm \*gmtime(const time\_t \*time);](https://www.runoob.com/cplusplus/c-function-gmtime.html)** 该函数返回一个指向 time 的指针，time 为 tm 结构，用协调世界时（UTC）  也被称为格林尼治标准时间（GMT）表示。 |
| 7 | **[time\_t mktime(struct tm \*time);](https://www.runoob.com/cplusplus/c-function-mktime.html)** 该函数返回日历时间，相当于 time 所指向结构中存储的时间。 |
| 8 | **[double difftime ( time\_t time2, time\_t time1 );](https://www.runoob.com/cplusplus/c-function-difftime.html)** 该函数返回 time1 和 time2 之间相差的秒数。 |
| 9 | **[size\_t strftime();](https://www.runoob.com/cplusplus/c-function-strftime.html)** 该函数可用于格式化日期和时间为指定的格式。 |

# C++ 基本的输入输出

C++ 标准库提供了一组丰富的输入/输出功能，我们将在后续的章节进行介绍。本章将讨论 C++ 编程中最基本和最常见的 I/O 操作。

C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做**输入操作**。如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做**输出操作**。

## I/O 库头文件

下列的头文件在 C++ 编程中很重要。

|  |  |
| --- | --- |
| **头文件** | **函数和描述** |
| <iostream> | 该文件定义了 **cin、cout、cerr** 和 **clog** 对象，分别对应于标准输入流、标准输出流、  非缓冲标准错误流和缓冲标准错误流。 |
| <iomanip> | 该文件通过所谓的参数化的流操纵器（比如 **setw** 和 **setprecision**），  来声明对执行标准化 I/O 有用的服务。 |
| <fstream> | 该文件为用户控制的文件处理声明服务。我们将在文件和流的相关章节讨论它的细节。 |

## 标准输出流（cout）

## 标准输入流（cin）

# C++定义结构

为了定义结构，您必须使用 **struct** 语句。

## 访问结构成员

为了访问结构的成员，我们使用**成员访问运算符（.）**。

## 结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。

void printBook( struct Books book );

// 输出 Book1 信息 printBook( Book1 );

## 指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

struct Books \*struct\_pointer;

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

struct\_pointer = &Book1;

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

struct\_pointer->title;

## typedef 关键字

下面是一种更简单的定义结构的方式，您可以为创建的类型取一个"别名"。例如：

typedef struct Books{

char title[50];

char author[50];

char subject[100];

int book\_id;}Books;

现在，您可以直接使用 *Books* 来定义 *Books* 类型的变量，而不需要使用 struct 关键字。下面是实例：

Books Book1, Book2;

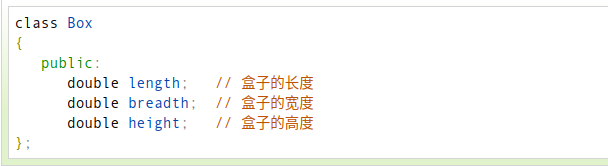
您可以使用 **typedef** 关键字来定义非结构类型，如下所示：

typedef long int \*pint32;

pint32 x, y, z;

x, y 和 z 都是指向长整型 long int 的指针。

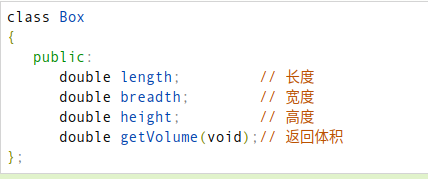
## C++ 类定义



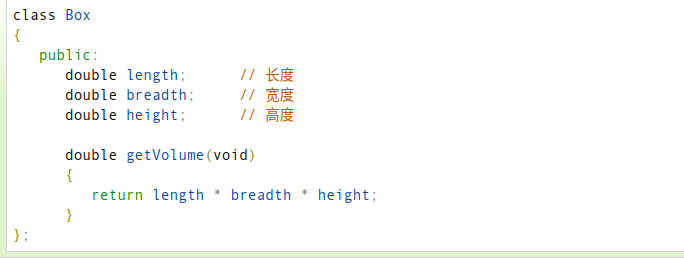
### 访问数据成员

类的对象的公共数据成员可以使用直接成员访问运算符 (.) 来访问。

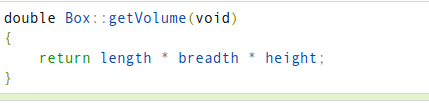
# C++ 类成员函数



成员函数可以定义在类定义内部，或者单独使用**范围解析运算符 ::** 来定义。在类定义中定义的成员函数把函数声明为**内联**的，即便没有使用 inline 标识符。所以您可以按照如下方式定义 **Volume()** 函数：



您也可以在类的外部使用**范围解析运算符 ::** 定义该函数，如下所示：



在这里，需要强调一点，在 :: 运算符之前必须使用类名。调用成员函数是在对象上使用点运算符（**.**），这样它就能操作与该对象相关的数据

# C++ 类访问修饰符

数据封装是面向对象编程的一个重要特点，它防止函数直接访问类类型的内部成员。类成员的访问限制是通过在类主体内部对各个区域标记 **public、private、protected** 来指定的。关键字 **public、private、protected** 称为访问修饰符。

一个类可以有多个 public、protected 或 private 标记区域。每个标记区域在下一个标记区域开始之前或者在遇到类主体结束右括号之前都是有效的。成员和类的默认访问修饰符是 private。

## 私有（private）成员

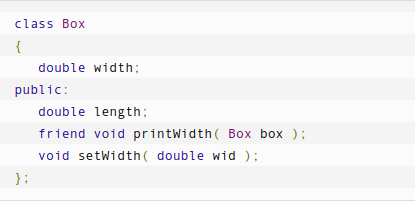
**私有**成员变量或函数在类的外部是不可访问的，甚至是不可查看的。只有类和友元函数可以访问私有成员。

# C++ 友元函数

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 **friend**，如下所示：



声明类 ClassTwo 的所有成员函数作为类 ClassOne 的友元，需要在类 ClassOne 的定义中放置如下声明：

friend class ClassTwo;



**友元函数的使用**

因为友元函数没有this指针，则参数要有三种情况：

要访问非static成员时，需要对象做参数；

要访问static成员或全局变量时，则不需要对象做参数；

如果做参数的对象是全局对象，则不需要对象做参数.

可以直接调用友元函数，不需要通过对象或指针



## 保护（protected）成员

**保护**成员变量或函数与私有成员十分相似，但有一点不同，保护成员在派生类（即子类）中是可访问的。

## 继承中的特点

有public, protected, private三种继承方式，它们相应地改变了基类成员的访问属性。

1.**public 继承：**基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：public, protected, private

2.**protected 继承：**基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：protected, protected, private

3.**private 继承：**基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：private, private, private

但无论哪种继承方式，上面两点都没有改变：

1.private 成员只能被本类成员（类内）和友元访问，不能被派生类访问；

2.protected 成员可以被派生类访问。

class B : public A{

class B : protected A{

class B : private A{

# C++ 类构造函数 & 析构函数

## 类的构造函数

类的**构造函数**是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。



## 类的析构函数

类的**析构函数**是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。

析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数。析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。

# C++ 内联函数

C++ **内联函数**是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。

对内联函数进行任何修改，都需要重新编译函数的所有客户端，因为编译器需要重新更换一次所有的代码，否则将会继续使用旧的函数。

如果想把一个函数定义为内联函数，则需要在函数名前面放置关键字 **inline**，在调用函数之前需要对函数进行定义。如果已定义的函数多于一行，编译器会忽略 inline 限定符。

在类定义中的定义的函数都是内联函数，即使没有使用 **inline** 说明符。

提升函数运行的速度，但是占用更多的内存；

# C++ this 指针

在 C++ 中，每一个对象都能通过 **this** 指针来访问自己的地址。**this** 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 **this** 指针，因为友元不是类的成员。只有成员函数才有 **this** 指针。

# C++ 指向类的指针

一个指向 C++ 类的指针与指向结构的指针类似，访问指向类的指针的成员，需要使用成员访问运算符 **->**，就像访问指向结构的指针一样。与所有的指针一样，您必须在使用指针之前，对指针进行初始化。

# C++ 类的静态成员

我们可以使用 **static** 关键字来把类成员定义为静态的。当我们声明类的成员为静态时，这意味着无论创建多少个类的对象，静态成员都只有一个副本。

静态成员在类的所有对象中是共享的。如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。我们不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 **::** 来重新声明静态变量从而对它进行初始化，

// 初始化类 Box 的静态成员

int Box::objectCount = 0;

## 静态成员函数

如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，**静态函数**只要使用类名加范围解析运算符 **::** 就可以访问。

静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。

静态成员函数有一个类范围，他们不能访问类的 this 指针。您可以使用静态成员函数来判断类的某些对象是否已被创建。

***静态成员函数与普通成员函数的区别：***

* 静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。
* 普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针。

## 访问控制和继承

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private。

我们可以根据访问权限总结出不同的访问类型，如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **访问** | **public** | **protected** | **private** |
| 同一个类 | yes | yes | yes |
| 派生类 | yes | yes | no |
| 外部的类 | yes | no | no |

一个派生类继承了所有的基类方法，但下列情况除外：

* 基类的构造函数、析构函数和拷贝构造函数。
* 基类的重载运算符。
* 基类的友元函数。

## 继承类型

当一个类派生自基类，该基类可以被继承为 **public、protected** 或 **private** 几种类型。继承类型是通过上面讲解的访问修饰符 access-specifier 来指定的。

我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。当使用不同类型的继承时，遵循以下几个规则：

* **公有继承（public）：**当一个类派生自**公有**基类时，基类的**公有**成员也是派生类的**公有**成员，基类的**保护**成员也是派生类的**保护**成员，基类的**私有**成员不能直接被派生类访问，但是可以通过调用基类的**公有**和**保护**成员来访问。
* **保护继承（protected）：** 当一个类派生自**保护**基类时，基类的**公有**和**保护**成员将成为派生类的**保护**成员。
* **私有继承（private）：**当一个类派生自**私有**基类时，基类的**公有**和**保护**成员将成为派生类的**私有**成员。

## 多继承

多继承即一个子类可以有多个父类，它继承了多个父类的特性。

C++ 类可以从多个类继承成员，语法如下：

class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…{<派生类类体>};

其中，访问修饰符继承方式是 **public、protected** 或 **private** 其中的一个，用来修饰每个基类，各个基类之间用逗号分隔，如上所示。

# C++ 重载运算符和重载函数

C++ 允许在同一作用域中的某个**函数**和**运算符**指定多个定义，分别称为**函数重载**和**运算符重载**。

重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

当您调用一个**重载函数**或**重载运算符**时，编译器通过把您所使用的参数类型与定义中的参数类型进行比较，决定选用最合适的定义。选择最合适的重载函数或重载运算符的过程，称为**重载决策**。

## C++ 中的函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。