**工作中的学习总结**

**一 基础知识内容大纲：**

1. C++语法知识：c++ primer plus 整本书的梳理与总结。
2. Qt/MFC框架常用技能：
3. 熟悉各自的消息传输机制，理解
4. 模板库、工具类、控件、对话框、窗体
5. 事件
6. 多线程
7. 文件IO
8. 重绘、图形、通讯
9. 数据结构与算法：
10. 《数据结构c++》总结与理解
11. 《LeetCode题集》总结与理解
12. 数据库

**二 工作中有大概几个能力模块：**

1. 市场调查能力，要对自己要做得产品在市场的需求非常熟悉。
2. 与人沟通能力，了解需求，但是各个客户的需求不一样，与供应商的需求沟通，与同事、下属、领导的沟通。
3. 整合业务能力，一个产品做出来，需要什么，我们的优势是什么？不仅仅是 自己写软件，只关心软件，产品设计的逻辑流程，以及过程中需要的东西，这是整体框架，非常重要。就算只是做维护工作，第一件事也是必须了解我们需要维护的产品的运行逻辑是什么。
4. 自己的专业能力，做软件的，那么就是写代码能力了。

**三 实施**

具体实施就是根据实际工作中，对公司的业务，市场，产品以及自己的专业能力，加强训练！！

**第一部分**

**基础知识内容大纲**

**C++语法知识：**

**c++ primer plus 的梳理与总结。**

目录

[第一、 二章 1](#_Toc32079)

[关于C++简介以及总结 1](#_Toc29531)

[第三章 处理数据 2](#_Toc4880)

[第一 命名规范、字节与位以及位运算 2](#_Toc6360)

[一 命名规范 2](#_Toc22902)

[第二 Const与#define的区别 4](#_Toc24488)

[一 const的作用 4](#_Toc31520)

[二 Const的使用 5](#_Toc19458)

[第三 c++算术运算符(拓展为所有运算符) 9](#_Toc10030)

[一 算术运算符 9](#_Toc17702)

[二 关系运算符 10](#_Toc20986)

[三 逻辑运算符 10](#_Toc22454)

[四 位运算符 10](#_Toc24714)

[五 赋值运算符 12](#_Toc23525)

[六 杂项运算符 13](#_Toc15486)

[七 C++ 中的运算符优先级 13](#_Toc9765)

[第四 运算符重载 14](#_Toc31884)

[一 为什么要对运算符进行重载 14](#_Toc24118)

[二 C++运算符重载的实质 14](#_Toc28105)

[三 可重载运算符/不可重载运算符 14](#_Toc14282)

[第五 类型转换 15](#_Toc11782)

[一 在C++基本的数据类型中，可以分为四类：整型，浮点型，字符型，布尔型。其中数值型包括 整型与浮点型；字符型即为char。 15](#_Toc24303)

[二 c++强制类型转换 15](#_Toc30855)

[第四章复合类型 21](#_Toc2995)

[第一 数组 21](#_Toc21620)

[一 声明数组 21](#_Toc4903)

[第二 字符串 23](#_Toc29323)

[一 c风格字符串 23](#_Toc29204)

[二 c++引入的string类类型 23](#_Toc2508)

[第三 LPCTSTR用法 34](#_Toc29025)

[第四CString用法 36](#_Toc6487)

[第四 常用的对字符串的操作函数 42](#_Toc1154)

[第五 结构 44](#_Toc23624)

[一 结构作为函数参数 44](#_Toc468)

[第六 共用体（略） 47](#_Toc24715)

[第七 枚举 47](#_Toc5256)

[一 枚举类型的定义 47](#_Toc3480)

[二 枚举格式说明 48](#_Toc4864)

[三 应用举例 48](#_Toc25074)

[四 重要提示 48](#_Toc1231)

[第八指针 48](#_Toc9615)

[一 指针定义 48](#_Toc15869)

[二 c++Null指针 48](#_Toc151)

[三 C++指针的算术运算 48](#_Toc24904)

[四 C++指针vs数组 49](#_Toc8486)

[第九 引用 53](#_Toc26988)

[第十vector和array类（在后面讨论模板一起，还有数组，string一起） 53](#_Toc12262)

[第五章 循环和关系表达式 54](#_Toc6594)

[第一 for循环 54](#_Toc6076)

[第二 while循环 54](#_Toc31573)

[第三 do..while循环 55](#_Toc28075)

[第四 循环控制语句（break，continue,goto语句） 55](#_Toc2167)

[一 C++ 中 break 语句有以下两种用法： 55](#_Toc18199)

[二 Continue语句 55](#_Toc28888)

[第6章 分支语句 56](#_Toc19106)

[第一 if语句（略） 56](#_Toc24656)

[第二 if..else语句（略） 56](#_Toc32149)

[第三 嵌套if语句（略） 56](#_Toc29835)

[第四 switch语句（略） 56](#_Toc9804)

[第五 嵌套switch语句（略） 56](#_Toc1101)

[第六 数字 56](#_Toc10990)

[第六 数字 56](#_Toc26062)

[一 C++ 数学运算 57](#_Toc5846)

[二 C++ 随机数 57](#_Toc21750)

[第七， 八章函数 58](#_Toc1117)

[第一 定义函数，函数原型（声明）和调用 58](#_Toc29914)

[一 定义函数 58](#_Toc2610)

[二 函数声明 59](#_Toc32552)

[三 调用函数 59](#_Toc18360)

[第二 函数参数和传递参数（按值传递，指针传递，引用传递），默认参数 59](#_Toc25694)

[一 函数参数 59](#_Toc9562)

[第三 函数和数组 67](#_Toc28619)

[第四 函数与C-风格字符串（略） 67](#_Toc12732)

[第五 函数与结构（略） 68](#_Toc27593)

[第六 递归函数 68](#_Toc21233)

[第七内联函数 74](#_Toc24255)

[第八回调函数 74](#_Toc13304)

[第十， 十一章，类 78](#_Toc21781)

[第一 类定义，对象和访问数据成员 79](#_Toc15951)

[一 C++ 类定义 79](#_Toc17384)

[二 访问数据成员 79](#_Toc28593)

[第二 类成员函数 79](#_Toc15696)

[第三 类访问修饰符（public，private，protected） 80](#_Toc3444)

[第四 构造函数和析构函数 81](#_Toc11826)

[一 默认构造函数 81](#_Toc9188)

[二 带参数的构造函数 82](#_Toc22715)

[三 类的析构函数 84](#_Toc24417)

[第五 c++拷贝构造函数 85](#_Toc26403)

[第六 友元函数和友元类 88](#_Toc6958)

[一 友元函数 88](#_Toc455)

[二 友元类 89](#_Toc22458)

[第七 c++内联函数（略） 91](#_Toc17116)

[第八 c++中的this指针 91](#_Toc23544)

[第九 智能指针 93](#_Toc23756)

[一 什么是智能指针 93](#_Toc8908)

[二 智能指针的使用 93](#_Toc12421)

[三智能指针的设计和实现 100](#_Toc31900)

[第十c++中指向类的指针 102](#_Toc15172)

[第十一 c++类的静态成员 103](#_Toc20084)

[一静态成员变量 103](#_Toc16284)

[二 静态成员函数 104](#_Toc25454)

[第十二 重载函数和运算符重载 106](#_Toc27591)

[一 C++ 中的函数重载 106](#_Toc10812)

[二 运算符重载 107](#_Toc3980)

[第十三， 十四章，继承 110](#_Toc18429)

[第一 继承 110](#_Toc18499)

[一 基类 & 派生类 110](#_Toc19569)

[二 访问控制和继承 111](#_Toc9340)

[三 继承类型 112](#_Toc12042)

[四 多继承 112](#_Toc22015)

[第二 多态 114](#_Toc20783)

[第三 数据抽象 118](#_Toc931)

[一 访问标签强制抽象 118](#_Toc25521)

[二 数据抽象的好处 119](#_Toc29432)

[三 数据抽象的实例 119](#_Toc13756)

[第四 接口（抽象类） 120](#_Toc24282)

[第十五章 十六章 异常，模板，和其他 122](#_Toc1455)

[第一 异常 122](#_Toc14710)

[一 抛出异常 123](#_Toc1348)

[二 捕获异常 123](#_Toc22662)

[三 C++ 标准的异常 125](#_Toc11249)

[二 定义新的异常 126](#_Toc13965)

[第二 模板 127](#_Toc719)

[一函数模板 127](#_Toc10687)

[二 类模板 128](#_Toc25083)

[第三 其他（在stl和数据结构里面） 130](#_Toc399)

[第十七章输入输出，文件（基本不用一般是使用Qt/MFC） 130](#_Toc7081)

[第一 基本的输入输出 130](#_Toc1910)

[一 I/O 库头文件 130](#_Toc12155)

[第二 文件和流 132](#_Toc4606)

[一 打开文件 133](#_Toc30960)

[二 关闭文件 133](#_Toc32281)

[三 写入文件 134](#_Toc20557)

[四 读取文件 134](#_Toc17812)

[五 读取 & 写入实例 134](#_Toc12107)

[六 文件位置指针 135](#_Toc11764)

[第九章， 十二章内存模型和名称空间 136](#_Toc30014)

[第一 命名空间 136](#_Toc12668)

[一 定义命名空间 137](#_Toc29339)

[二 using 指令 138](#_Toc2344)

[三 嵌套的命名空间 139](#_Toc17785)

[第二 内存模型 142](#_Toc10123)

[一 内存管理 142](#_Toc14198)

[二 内存泄露 144](#_Toc13250)

[三 内存回收 144](#_Toc3940)

1. **二章**

**关于C++简介以及总结**

去年刚学习c++的时候，和接触到公司软件时候有个疑问：using namespace std没怎么看到，并且看到用了string，现在能明白了，第一公司用的是MFC一般使用CString，路径之类的使用string，但是一般是在已经封装好的库里面，那里面已经使用了using namespace std，在#include的时候已经包含进去了。头文件的时候可以用老式的。

命名空间为了区分两个厂商相同的名字内容。一般如果用后面的重载，多态。封装。

返回与不返回函数的区别！不返回函数一般是对数据的操作！

1. **处理数据**

***知识点概括：***

**第一：命名规范、字节与位以及位运算**

**第二：Const与#define的区别**

**第三：c++算术运算符(拓展为所有运算符)**

**第四：运算符重载**

**第五：类型转换**

**第一 命名规范、字节与位以及位运算**

**一 命名规范**

1. **类命名规范**

CMyPalyerManager MyPalyerManager

C为前缀，每个单词首字母大写。

1. **变量命名规范**
2. **整型前缀**

Short sId;//至少16位

int iId; //至少32位

Unsignd unId;//unsigned 与不加大小一样，只是会表示只是正数

Long lId; //至少32位

1个位=0或1，位是最小的单位，一个字节=8个位

1. **浮点型前缀**

float fValue;

double dValue;

1. **字符型前缀**

char chChar;

1. **字符串前缀**

char szPath;

string strPath;

CString strPath;//MFC中的

1. **布尔型前缀**

bool bIsOK;

BOOL bIsOk;//MFC的

1. **指针型前缀**

char\* pPath;//p是前缀

1. **数组前缀**

int arrnNum;//arr是前缀+n

CString arrstrName;//arr+str

1. **结构体前缀**

结构体前缀是tag，之后以大写字母开头

1. **枚举前缀**

enum emWeek;

1. **字节前缀**

BYTE byIP; //by是前缀

1. **字的前缀**

DWORD dwMsgID//4位

WORD wMsgId//2位

1. **字符指针前缀**

LPCTSTR ptszName //

LPCSTR pszName

LPSTR pszName

1. **STL容器前缀**

Vector vecValue;

1. **RECT矩形结构前缀**

RECT rcChild;

CRECT rcChild;//rc

1. **句柄前缀**

HWND hWndDlg;//h

HBRUSH hBr//h

HPEN hPen//h

1. **windows颜色前缀**

COLORREF crFont;

1. **成员变量**

CString m\_strName;//m\_+str

1. **全局变量**

统一用g\_前缀

1. **定义结构体采用typedef**

typedef struct \_tag STUDENT

{

Int nId;

CString strName;

} TAG\_STUDENT \*PSTUDENT;

TAG\_STUDENT tagXiaoZhang;

1. **定义枚举**

typedef enum \_EM\_FILE\_OPEN\_MODE

{

}

EM\_FILE\_OPEN\_MODE

**第二 Const与#define的区别**

**一 const的作用**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **作用** | **说明** | **参考代码** |
| 1 | 可以定义const常量 |  | const int Max = 100; |
| 2 | 便于进行类型检查 | const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误 | void f(const int i) { .........}       //对传入的参数进行类型检查，不匹配进行提示 |
| 3 | 可以保护被修饰的东西 | 防止意外的修改，增强程序的健壮性。 | void f(const int i) { i=10;//error! }       //如果在函数体内修改了i，编译器就会报错 |
| 4 | 可以很方便地进行参数的调整和修改 | 同宏定义一样，可以做到不变则已，一变都变 |  |
| 5 | 为函数重载提供了一个参考 |  | class A {            ......   void f(int i)       {......} //一个函数   void f(int i) const {......} //上一个函数的重载            ...... }; |
| 6 | 可以节省空间，避免不必要的内存分配 | const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝 | #define PI 3.14159         //常量宏  const doulbe  Pi=3.14159;  //此时并未将Pi放入ROM中  ......  double i=Pi;   //此时为Pi分配内存，以后不再分配！  double I=PI;  //编译期间进行宏替换，分配内存  double j=Pi;  //没有内存分配  double J=PI;  //再进行宏替换，又一次分配内存 |
| 7 | 提高了效率 | 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高 |  |

**二 Const的使用**

**（一）定义常量**

1. **const修饰变量，**

以下两种定义形式在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。

 TYPE const ValueName = value;

  const TYPE ValueName = value;

1. **将const改为外部连接**

作用于扩大至全局,编译时会分配内存,并且可以不进行初始化,仅仅作为声明,编译器认为在程序其他地方进行了定义。

     extend const int ValueName = value;

**（二）指针使用const**

**(1)指针本身是常量不可变**

     char\* const pContent; （pContent是不变的，比如pContent=1，但是可以开辟其他空间放pContent）

**(2)指针所指向的内容是常量不可变**

     const char \*pContent; （变量的地址不变）

**(3)两者都不可变**

      const char\* const pContent;

**(4)还有其中区别方法，沿着\*号划一条线：**

如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；

如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

**（三）函数中使用const**

1. **const修饰函数参数**

a.传递过来的参数在函数内不可以改变(无意义，因为Var本身就是形参)，保护参数

void function(const int Var);

b.参数指针所指内容为常量不可变

void function(const char\* Var);（具体怎么用还是不太明白）

c.参数指针本身为常量不可变(也无意义，因为char\* Var也是形参)

void function(char\* const Var);

d.参数为引用，为了增加效率同时防止修改。修饰引用参数时：

void function(const Class& Var); //引用参数在函数内不可以改变

void function(const TYPE& Var); //引用参数在函数内为常量不可变

这样的一个const引用传递和最普通的函数按值传递的效果是一模一样的,他禁止对引用的对象的一切修改,唯一不同的是按值传递会先建立一个类对象的副本, 然后传递过去,而它直接传递地址,所以这种传递比按值传递更有效.

另外只有引用的const传递可以传递一个临时对象,因为临时对象都是const属性, 且是不可见的,他短时间存在一个局部域中,所以不能使用指针,只有引用的const传递能够捕捉到这个家伙.

1. **const 修饰函数返回值**

    const修饰函数返回值其实用的并不是很多，它的含义和const修饰普通变量以及指针的含义基本相同。

    a.const int fun1() //这个其实无意义，因为参数返回本身就是赋值。

    b. const int \* fun2() //调用时 const int \*pValue = fun2();

                          //我们可以把fun2()看作成一个变量，即指针内容不可变。

    c.int\* const fun3()   //调用时 int \* const pValue = fun2();

                          //我们可以把fun2()看作成一个变量，即指针本身不可变。

一般情况下，函数的返回值为某个对象时，如果将其声明为const时，多用于操作符的重载。通常，不建议用const修饰函数的返回值类型为某个对象或对某个对象引用的情况。原因如下：如果返回值为某个对象为const（const A test = A 实例）或某个对象的引用为const（const A& test = A实例） ，则返回值具有const属性，则返回实例只能访问类A中的公有（保护）数据成员和const成员函数，并且不允许对其进行赋值操作，这在一般情况下很少用到。

例如：305中：

public:

CLSID const& GetClsid()

{

static CLSID const clsid

= { 0x9AE86E92, 0x97BC, 0x11D1, { 0xA4, 0x9B, 0x0, 0x0, 0x21, 0x63, 0x31, 0x68 } };

return clsid;

}

**（四）类相关const**

1. **const修饰成员变量**

const修饰类的成员函数，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。

    class A

    {

        …

        const int nValue;         //成员常量不能被修改

        …

        A(int x): nValue(x) { } ; //只能在初始化列表中赋值

     }

1. **const修饰成员函数**

const修饰类的成员函数，则该成员函数不能修改类中任何非const成员函数。一般写在函数的最后来修饰。

    class A

    {

        …

       void function()const; //常成员函数, 它不改变对象的成员变量.

//也不能调用类中任何非const成员函数。

}

对于const类对象/指针/引用，只能调用类的const成员函数，因此，const修饰成员函数的最重要作用就是限制对于const对象的使用。

a. const成员函数不被允许修改它所在对象的任何一个**数据成员（都是在类里面的）**。

b. const成员函数能够访问对象的const成员，而其他成员函数不可以。

CTime CTime::Sum(const CTime &t) const

{

CTime sum;

sum.m\_nMinutes = m\_nHours + t.m\_nMinutes;

sum.m\_nHours = m\_nMinutes +t.m\_nHours +sum.m\_nMinutes%60;

sum.m\_nMinutes &=60;

return sum;

}

在这个函数里面返回的都是这个类成员，没有改变这个类成员变量。

1. **const修饰类对象/对象指针/对象引用**

a.    const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。

 b.    const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。

例如：

class CMyFunction

{

    void func1();

void func2() const;

}

const CMyFunction aObj; （对象）

aObj.func1(); ×

aObj.func2(); 正确

Const CMyFunction\* aObj = new CMyFunction(); （new指针时const的，非const函数与变量都不可以访问）

aObj-> func1(); ×

aObj-> func2(); 正确

**（五）将Const类型转化为非Const类型的方法**

采用const\_cast 进行转换。

用法：const\_cast <type\_id>  (expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。

**（1）**  常量指针被转化成非常量指针，并且仍然指向原来的对象；

**（2）** 常量引用被转换成非常量引用，并且仍然指向原来的对象；

**（3）** 常量对象被转换成非常量对象。

**（六）使用const的一些建议**

**（1）**多尝试使用const，它有益于我们对编程的理解；

**（2）**要避免最一般的赋值操作错误，如将const变量赋值；

**（3）**在参数中使用const应该使用引用或指针，而不是一般的对象实例;

**（4）** const在成员函数中的三种用法（参数、返回值、函数）要很好的使用；

**（5）**不要轻易的将函数的返回s值类型定为const;

**（6）**除了重载操作符外一般不要将返回值类型定为对某个对象的const引用;

**（7）** 任何不会修改数据成员的函数都应该声明为const 类型。

**（七）补充重要说明**

**（1）** 类内部的常量限制：使用这种类内部的初始化语法的时候，常量必须是被一个常量表达式初始化的整型或枚举类型，而且必须是static和const形式。

**（2）**如何初始化类内部的常量？：一种方法就是static 和 const 并用，在外部初始化，例如：

class A

{

public:

A() {}

private:

static const int i; //注意必须是静态的！

}；

const int A::i=3;

另一个很常见的方法就是初始化列表：

class A {

public:

A(int i=0):

test(i) {}

private:

const int i;

}；

**（3）**如果在非const成员函数中，this指针只是一个类类型的；如果在const成员函数中，this指针是一个const类类型的；如果在volatile成员函数中,this指针就是一个volatile类类型的。

**（4）** new返回的指针必须是const类型的。

**第三 c++算术运算符(拓展为所有运算符)**

**一 算术运算符**

假设变量 A 的值为 10，变量 B 的值为 20，则：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| + | 把两个操作数相加 | A + B 将得到 30 |
| - | 从第一个操作数中减去第二个操作数 | A - B 将得到 -10 |
| \* | 把两个操作数相乘 | A \* B 将得到 200 |
| / | 分子除以分母 | B / A 将得到 2，在实际代码中这个是比较常见的。详细讲一下， |
| % | 取模运算符，整除后的余数 | B % A 将得到 0 |
| ++ | [自增运算符](https://www.w3cschool.cn/cpp/cpp-increment-decrement-operators.html)，整数值增加 1 | A++ 将得到 11，先使用A=10执行表达式，之后再是a=a+1=11，++A是先A=A+1=11,之后才是执行表达式 |
| -- | [自减运算符](https://www.w3cschool.cn/cpp/cpp-increment-decrement-operators.html)，整数值减少 1 | A-- 将得到 9 |

分子除以分母详细说下：

如果分子、分母两个操作数都是整数，那么/将执行整数除法，结果中的小数部分将被丢弃，如果其中一个（或者两个）操作数是浮点型，那么结果就将保留小数点。

nPass = g\_pWafer->PassCnt()；

nTotal = g\_pWafer->ProbeCnt();

double dbYiele =nPass\* 100.0/nTotal ;

这是计算正确率的，如果没有\*100.0，那么两个操作数都是整数，只是保留0，如果是\*100那么还是整数，小数部分还是不会保存。

对比一下

int nColNumFrist = 365;

int nTimesA = nColNumFrist / 100 ;

这个时候nTimesA = 3

### 二 关系运算符

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| == | 检查两个操作数的值是否相等，如果相等则条件为真。 | (A == B) 不为真。 |
| != | 检查两个操作数的值是否相等，如果不相等则条件为真。 | (A != B) 为真。 |
| > | 检查左操作数的值是否大于右操作数的值，如果是则条件为真。 | (A > B) 不为真。 |
| < | 检查左操作数的值是否小于右操作数的值，如果是则条件为真。 | (A < B) 为真。 |
| >= | 检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。 | (A >= B) 不为真。 |
| <= | 检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。 | (A <= B) 为真。 |

### 三 逻辑运算符

假设变量 A 的值为 1，变量 B 的值为 0，则：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| && | 称为逻辑与运算符。如果两个操作数都非零，则条件为真。 | (A && B) 为假。 |
| || | 称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。 | (A || B) 为真。 |
| ! | 称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。 | !(A && B) 为真。 |

### 四 位运算符

位运算符总共6个，这些运算符只能用于整型操作数，也就是带符号或者无符号的char，short，int与long类型，在实际的工作中一般使用unsigned整型，

A = 3,B = 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p | q | p&q | p|q | p^q |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| & | 如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。 | 1(00000000 00000001) |
| | | 如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。 | 7 (00000000 00000111) |
| ^ | 如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。 | X^Y = 6(00000000 00000111) |
| ~ | 二进制补码运算符是一元运算符，具有"翻转"位效果。位求反。 | (11111111 11111100)=65532或者-4 |
| << | 二进制左移运算符。左操作数的值向左移动右操作数指定的位数。 | 6 (00000000 00000110) |
| >> | 二进制右移运算符。左操作数的值向右移动右操作数指定的位数。 | A >> 2 将得到 15，即为 0000 1111 |

#include <iostream>

using namespace *std*;

int *main*()

{

unsigned short x=3,y=5;

*cout*<<"~x= "<<(unsigned short)~x<<*endl*;

*cout*<<"~x= "<<~x<<*endl*;

*cout*<<"x&y= "<<(x&y)<<*endl*;

*cout*<<"x^y= "<<(x^y)<<*endl*;

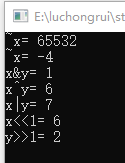
*cout*<<"x|y= "<<(x|y)<<*endl*;

*cout*<<"x<<1= "<<(x<<1)<<*endl*;

*cout*<<"y>>1= "<<(y>>1)<<*endl*;

return 0;

}



Short 是16位整数 ，所以用x，y的二进制表示如下

X = 3 (00000000 00000011)

Y = 5 (00000000 00000101)

~x =65532 (11111111 11111100)或者-4

X = 3 (00000000 00000011)

Y = 5 (00000000 00000101)

X&Y = 1 (00000000 00000001)

X = 3 (00000000 00000011)

Y = 5 (00000000 00000101)

X^Y = 6 (00000000 00000110)

X = 3 (00000000 00000011)

Y = 5 (00000000 00000101)

X|Y = 7 (00000000 00000111)

X = 3 (00000000 00000011)

x<<1 6 (00000000 00000110)

Y = 5 (00000000 00000101)

1. >1 (00000000 00000010)

实际中用到的比较多位运算是在与硬件打交道上。

BOOL CTemperatureControl::WriteSingleRegister(WORD addr, WORD data, BOOL bWaitFor /\* = TRUE \*/, DWORD dwTimeOut /\* = DEFAULT\_TIMEOUT \*/)

{

BYTE SendArr[10] = {0};

SendArr[0] = m\_nModAddress;

SendArr[1] = WRITE\_BYTE;

SendArr[2] = addr >> 8;

SendArr[3] = addr & 0xFF;

SendArr[4] = data >> 8;

SendArr[5] = data & 0xFF;

return WriteDataToTAC3000(SendArr, 6, bWaitFor, dwTimeOut);

}

### 五 赋值运算符

下表列出了 C++ 支持的赋值运算符：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| = | 简单的赋值运算符，把右边操作数的值赋给左边操作数 | C = A + B 将把 A + B 的值赋给 C |
| += | 加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数 | C += A 相当于 C = C + A |
| -= | 减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数 | C -= A 相当于 C = C - A |
| \*= | 乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数 | C \*= A 相当于 C = C \* A |
| /= | 除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数 | C /= A 相当于 C = C / A |
| %= | 求模且赋值运算符，求两个操作数的模赋值给左边操作数 | C %= A 相当于 C = C % A |
| <<= | 左移且赋值运算符 | C <<= 2 等同于 C = C << 2 |
| >>= | 右移且赋值运算符 | C >>= 2 等同于 C = C >> 2 |
| &= | 按位与且赋值运算符 | C &= 2 等同于 C = C & 2 |
| ^= | 按位异或且赋值运算符 | C ^= 2 等同于 C = C ^ 2 |
| |= | 按位或且赋值运算符 | C |= 2 等同于 C = C | 2 |

### 六 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.w3cschool.cn/cpp/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.w3cschool.cn/cpp/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.w3cschool.cn/cpp/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.w3cschool.cn/cpp/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.w3cschool.cn/cpp/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.w3cschool.cn/cpp/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.w3cschool.cn/cpp/cpp-pointer-operators.html) 指向一个变量。例如，\*var; 将指向变量 var。 |

### 七 C++ 中的运算符优先级

|  |  |  |
| --- | --- | --- |
| **类别** | **运算符** | **综合性** |
| 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 |

**第四 运算符重载**

**一 为什么要对运算符进行重载**

C++预定义中的运算符的操作对象只局限于基本的内置数据类型，但是对于我们自定义的类型（类）是没有办法操作的。但是大多时候我们需要对我们定义的类型进行类似的运算，这个时候就需要我们对这么运算符进行重新定义，赋予其新的功能，以满足自身的需求。

**二 C++运算符重载的实质**

运算符重载的实质就是函数重载或函数多态。运算符重载是一种形式的C++多态。目的在于让人能够用同名的函数来完成不同的基本操作。要重载运算符，需要使用被称为运算符函数的特殊函数形式，运算符函数形式：operatorp（argument-list）//operator 后面的'p'为要重载的运算符符号。

<返回类型说明符> operator <运算符符号>(<参数表>)

{

<函数体>

}

下面的例子好好体会下！！

inline int CCellRange::operator==(const CCellRange& rhs)

{

return ((m\_nMinRow == rhs.m\_nMinRow) && (m\_nMinCol == rhs.m\_nMinCol) &&

(m\_nMaxRow == rhs.m\_nMaxRow) && (m\_nMaxCol == rhs.m\_nMaxCol));

}

inline int CCellRange::operator!=(const CCellRange& rhs)

{

return !operator==(rhs);

}

**三 可重载运算符/不可重载运算符**

**（一）可重载运算符**

|  |  |
| --- | --- |
| 双目算术运算符 | + (加)，-(减)，\*(乘)，/(除)，% (取模) |
| 关系运算符 | ==(等于)，!= (不等于)，< (小于)，> (大于>，<=(小于等于)，>=(大于等于) |
| 逻辑运算符 | ||(逻辑或)，&&(逻辑与)，!(逻辑非) |
| 单目运算符 | + (正)，-(负)，\*(指针)，&(取地址) |
| 自增自减运算符 | ++(自增)，--(自减) |
| 位运算符 | | (按位或)，& (按位与)，~(按位取反)，^(按位异或),，<< (左移)，>>(右移) |
| 赋值运算符 | =, +=, -=, \*=, /= , % = , &=, |=, ^=, <<=, >>= |
| 空间申请与释放 | new, delete, new[ ] , delete[] |
| 其他运算符 | **()**(函数调用)，**->**(成员访问)，**,**(逗号)，**[]**(下标) |

**（二）不可重载运算符**

**（1）.**：成员访问运算符

**（2）.\***, **->\***：成员指针访问运算符

**（3）::**：域运算符

**（4）sizeof**：长度运算符

**（5）?:**：条件运算符

**（6）#**： 预处理符号

**第五 类型转换**

**一 在C++基本的数据类型中，可以分为四类：整型，浮点型，字符型，布尔型。其中数值型包括 整型与浮点型；字符型即为char。**

下面是一些c风格的强制转换

**（1）**将浮点型数据赋值给整型变量时，舍弃其小数部分。

**（2）**将整型数据赋值给浮点型变量时，数值不变，但是以指数形式存储。

**（3）**将double型数据赋值给float型变量时，注意数值范围溢出。

**（4）**字符型数据可以赋值给整型变量，此时存入的是字符的ASCII码。

**（5）**将一个int，short或long型数据赋值给一个char型变量，只将低8位原封不动的送到char型变量中。   
**（6）**将有符号型数据赋值给长度相同的无符号型变量，连同原来的符号位一起传送。

**二 c++强制类型转换**

在C++语言中新增了四个关键字static\_cast、const\_cast、reinterpret\_cast和dynamic\_cast。这四个关键字都是用于强制类型转换的。

新类型的强制转换可以提供更好的控制强制转换过程，允许控制各种不同种类的强制转换。

C++中风格是static\_cast<type>(content)。C++风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么。程序员只要扫一眼这样的代码，就能立即知道一个强制转换的目的。

**（一）Static\_cast**

在C++语言中static\_cast用于数据类型的强制转换，强制将一种数据类型转换为另一种数据类型。例如将整型数据转换为浮点型数据.

例如：c采用的类型转换方式

int a = 10;

int b = 3;

**double result = (double)a / (double)b;这里就是将a转换成double**

在C++语言中，我们可以采用static\_cast关键字来进行强制类型转换，如下所示。  
[例2]static\_cast关键字的使用：

int a = 10;

int b = 3;

double result = static\_cast<double>(a) / static\_cast<double>(b);

在本例中同样是将整型变量a转换为双精度浮点型。采用static\_cast进行强制数据类型转换时，将想要转换成的数据类型放到尖括号中，将待转换的变量或表达式放在元括号中，其格式可以概括为如下形式：

用法：static\_cast <类型说明符> （变量或表达式）

它主要有如下几种用法：    **（1）**用于类层次结构中基类和派生类之间指针或引用的转换  
      进行上行转换（把派生类的指针或引用转换成基类表示）是安全的  
      进行下行转换（把基类的指针或引用转换为派生类表示），由于没有动态类型检查，所以是不安全的  
    **（2）**用于基本数据类型之间的转换，如把int转换成char。这种转换的安全也要开发人员来保证  
    **（3）**把空指针转换成目标类型的空指针  
    **（4）**把任何类型的表达式转换为void类型  
    注意：static\_cast不能转换掉expression的const、volitale或者\_\_unaligned属性。

static\_cast:可以实现C++中内置基本数据类型之间的相互转换。

如果涉及到类的话，static\_cast只能在有相互联系的类型中进行相互转换,不一定包含虚函数。

**（二）const\_cast**

在C语言中，const限定符通常被用来限定变量，用于表示该变量的值不能被修改。而const\_cast则正是用于强制去掉这种不能被修改的常数特性，但需要特别注意的是const\_cast不是用于去除变量的常量性，而是去除指向常数对象的指针或引用的常量性，其去除常量性的对象必须为指针或引用。

用法：const\_cast<type\_id> (expression)  
    该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。  
    常量指针被转化成非常量指针，并且仍然指向原来的对象；  
    常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

了解了const\_cast的使用场景后，可以知道使用const\_cast通常是一种无奈之举，同时也建议大家在今后的C++程序设计过程中一定不要利用const\_cast去掉指针或引用的常量性并且去修改原始变量的数值，这是一种非常不好的行为。

1. **reinterpret\_cast**

在C++语言中，reinterpret\_cast主要有三种强制转换用途：

1. 改变指针或引用的类型、
2. 将指针或引用转换为一个足够长度的整形
3. 将整型转换为指针或引用类型。

用法：reinterpret\_cast<type\_id> (expression)  
    type-id必须是一个指针、引用、算术类型、函数指针或者成员指针。  
    它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。  
    在使用reinterpret\_cast强制转换过程仅仅只是比特位的拷贝，因此在使用过程中需要特别谨慎！

例7：

int \*a = new int;

double \*d = reinterpret\_cast<double \*>(a);

在例7中，将整型指针通过reinterpret\_cast强制转换成了双精度浮点型指针。

reinterpret\_cast可以将指针或引用转换为一个足够长度的整形，此中的足够长度具体长度需要多少则取决于操作系统，如果是32位的操作系统，就需要4个字节及以上的整型，如果是64位的操作系统则需要8个字节及以上的整型。

**（四）dynamic\_cast**

用法：dynamic\_cast<type\_id> (expression)

**（1）**其他三种都是编译时完成的，dynamic\_cast是运行时处理的，运行时要进行类型检查。

**（2）**不能用于内置的基本数据类型的强制转换。

**（3）**dynamic\_cast转换如果成功的话返回的是指向类的指针或引用，转换失败的话则会返回NULL。

**（4）**使用dynamic\_cast进行转换的，基类中一定要有虚函数，否则编译不通过。

        需要检测有虚函数的原因：类中存在虚函数，就说明它有想要让基类指针或引用指向派生类对象的情况，此时转换才有意义。

        这是由于运行时类型检查需要运行时类型信息，而这个信息存储在类的虚函数表（关于[虚函数表](http://baike.baidu.com/view/3750123.htm)的概念，详细可见<Inside c++ object model>）中，

        只有定义了虚函数的类才有虚函数表。

**（5）**在类的转换时，在类层次间进行上行转换时，dynamic\_cast和[static\_cast](http://baike.baidu.com/view/1745207.htm)的效果是一样的。在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

        向上转换，即为子类指针指向父类指针（一般不会出问题）；向下转换，即将父类指针转化子类指针。

       向下转换的成功与否还与将要转换的类型有关，即要转换的指针指向的对象的实际类型与转换以后的对象类型一定要相同，否则转换失败。

        在C++中，编译期的类型转换有可能会在运行时出现错误，特别是涉及到类对象的指针或引用操作时，更容易产生错误。Dynamic\_cast操作符则可以在运行期对可能产生问题的类型转换进行测试。

例 1

#include<iostream>

using namespace std;

class base

{

public :

    void m(){cout<<"m"<<endl;}

};

class derived : public base

{

public:

    void f(){cout<<"f"<<endl;}

};

int main()

{

    derived \* p;

    p = new base;

    p = static\_cast<derived \*>(new base);

    p->m();

    p->f();

    return 0;

}

本例中定义了两个类：base类和derived类，这两个类构成继承关系。在base类中定义了m函数，derived类中定义了f函数。在前面介绍多态时，我们一直是用基类指针指向派生类或基类对象，而本例则不同了。

本例主函数中定义的是一个派生类指针，当我们将其指向一个基类对象时，这是错误的，会导致编译错误。

但是通过强制类型转换我们可以将派生类指针指向一个基类对象，p = static\_cast<derived \*>(new base);语句实现的就是这样一个功能，这样的一种强制类型转换时合乎C++语法规定的，但是是非常不明智的，它会带来一定的危险。

在程序中p是一个派生类对象，我们将其强制指向一个基类对象，首先通过p指针调用m函数，因为基类中包含有m函数，这一句没有问题，之后通过p指针调用f函数。一般来讲，因为p指针是一个派生类类型的指针，而派生类中拥有f函数，因此p->f();这一语句不会有问题，

但是本例中p指针指向的确实基类的对象，而基类中并没有声明f函数，虽然p->f();这一语句虽然仍没有语法错误，但是它却产生了一个运行时的错误。换言之，p指针是派生类指针，这表明程序设计人员可以通过p指针调用派生类的成员函数f，但是在实际的程序设计过程中却误将p指针指向了一个基类对象，这就导致了一个运行期错误。

产生这种运行期的错误原因在于static\_cast强制类型转换时并不具有保证类型安全的功能，而C++提供的dynamic\_cast却能解决这一问题，dynamic\_cast可以在程序运行时检测类型转换是否类型安全。

当然dynamic\_cast使用起来也是有条件的，它要求所转换的操作数必须包含多态类类型（即至少包含一个虚函数的类）。

例2

#include<iostream>

using namespace std;

class base

{

public :

    void m(){cout<<"m"<<endl;}

};

class derived : public base

{

public:

    void f(){cout<<"f"<<endl;}

};

int main()

{

     derived \* p;

     p = new base;

     p = dynamic\_cast<derived \*>(new base);

     p->m();

     p->f();

     return 0;//操作有点骚气！！！

}

在本例中利用dynamic\_cast进行强制类型转换，但是因为base类中并不存在虚函数，因此p = dynamic\_cast<derived \*>(new base);这一句会编译错误。

为了解决本例中的语法错误，我们可以将base类中的函数m声明为虚函数，virtual void m(){cout<<"m"<<endl;}。  
  
dynamic\_cast还要求<>内部所描述的目标类型必须为指针或引用。

例3

#include<iostream>

#include<cstring>

using namespace std;

class Cbase

{

public:

Cbase(){};

virtual void show() {cout<<"base\n "<<endl;};

};

class Cderived: public Cbase

{

public:

Cderived() {};

virtual void show(){cout<<" derived"<<endl;};

};

int main()

{

Cbase\* base = new Cderived;

if (Cderived\* ptr1 = dynamic\_cast<Cderived\*>(base))

{

cout<<"转换成功"<<endl;

ptr1->show();

cout<<endl;

}

else

cout<<"转换失败！！hahahha"<<endl;

base = new Cbase;

if (Cderived\* ptr2 = dynamic\_cast<Cderived\*>(base))

{

cout<<"转换成功"<<endl;

ptr2->show();

cout<<endl;

}

else

{

cout<<"转换失败！！"<<endl;

}

delete(base);

return 0;

}

**(1)**对于类的dynamic\_cast转换，基类中一定要有虚函数，否则编译不通过。

**(2)**对指针进行dynamic\_cast，失败返回null，成功返回正常cast后的对象指针。

**(3)**对引用进行dynamic\_cast，失败抛出一个异常bad\_cast，成功返回正常cast后的对象引用。

**(4)**dynamic\_cast在运行期强制转换，运行时进行类型检查，较安全。

**(5)**不能用于内置的基本数据类型的强制转换。

**第四章复合类型**

***知识点概括：***

**第一：数组**

**第二：字符串**

**第三：结构**

**第四：共用体（略）**

**第五：枚举**

**第六：指针**

**第七：引用**

**第八：vector和array类（和后面的string类，以及stl放到leetecode一起总结一下）**

**第九：内存（加上第九章）放在最后面搞，很重要。也很难理解**

**第一 数组**

**一 声明数组**

声明一维数组：type arrayName [arraySize]

声明二维数组：type arrayName [arraySize][arraySize];以此类推

**(一)初始化数组**

初始化一维数组：double balace [5] = {78.0, 222,33, 99.8}{}里面的数字个数不大于arraySize

**(二)初始化二维数组**

double balace [3][4]={

{35.4, 35.9,33.9, 45.8},

{35.4, 35.9,33.9, 45.8},

{35.4, 35.9,33.9, 45.8},

}

**（三）数组用法**

**（1）指向数组的指针**

doblue balance [50];

balance 是指向&balance[0]的指针，也就是说balance是数组balance的第一个元素的地址。指针的值是地址。

double \*p;

double balance[10];

p = balance;

因此，\*(balance + 4) 是一种访问 balance[4] 数据的合法方式。一旦您把第一个元素的地址存储在 p 中，您就可以使用 \*p、\*(p+1)、\*(p+2) 等来访问数组元素。

**（2）传递数组给函数**

**C++中**可以通过指定不带索引的数组名来传递一个指向数组的指针，数组给一个函数，数组类型自动转换为指针类型，因而传的实际是地址。有以下3种方式都可以：

**方式1**

形式参数是一个指针：

Void myFunction (int \* param)

{

}

**方式2**

形式参数是一个已定义大小的数组：

Void myFunction (int param [10])

{

}

**方式3**

形式参数是一个未定义大小的数组：

Void myFunction (int param [])

{

}

**（3）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，如下

int \*myFunction()

{

}

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。

**第二 字符串**

**一 c风格字符串**

C风格的字符串起源于 C 语言，并在 C++ 中继续得到支持。**字符串实际上是使用 null 字符 '\0' 终止的一维字符数组**。因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

Char greeting [6] = {‘H’,’e’,’l’,’l’,’o’,’\0’};

依据数组初始化规则，您可以把上面的语句写成以下语句：

Char greeting [] = “Hello”;

**在工作中常常会使用const char\*,也要总结一下这些区别！！**

**二 c++引入的string类类型**

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。后面我们将学习 C++ 标准库中的这个类(stl)，我会在后面整理和总结stl。现在让我们先来看看下面这个实例：

现在您可能还无法透彻地理解这个实例，因为到目前为止我们还没有讨论类和对象。所以现在您可以只是粗略地看下这个实例，等理解了面向对象的概念之后再回头来理解这个实例。

|  |  |
| --- | --- |
| 函数名称 | 功能 |
| 构造函数 | 产生或复制字符串 |
| 析构函数 | 销毁字符串 |
| =，assign | 赋以新值 |
| Swap | 交换两个字符串的内容 |
| + =，append( )，push\_back() | 添加字符 |
| insert () | 插入字符 |
| erase() | 删除字符 |
| clear () | 移除全部字符 |
| resize () | 改变字符数量 |
| replace() | 替换字符 |
| + | 串联字符串 |
| ==，！ =，<，<=，>，>=，compare() | 比较字符串内容 |
| size()，length() | 返回字符数量 |
| max\_size () | 返回字符的最大可能个数 |
| empty () | 判断字符串是否为空 |
| capacity () | 返回重新分配之前的字符容量 |
| reserve() | 保留内存以存储一定数量的字符 |
| [],at() | 存取单一字符 |
| >>，getline() | 从 stream 中读取某值 |
| << | 将值写入 stream |
| copy() | 将内容复制为一个 C - string |
| c\_str() | 将内容以 C - string 形式返回 |
| data() | 将内容以字符数组形式返回 |
| substr() | 返回子字符串 |
| find() | 搜寻某子字符串或字符 |
| begin( )，end() | 提供正向迭代器支持 |
| rbegin()，rend() | 提供逆向迭代器支持 |
| get\_allocator() | 返回配置器 |

**（一）String用法详解：**

**（1）string的初始化**

String s1; //只是定义，没有初始化，编译器会将默认值””赋给s1，

String s2 = “c plus plus”;//变量 s2 在定义的同时被初始化为"c plus plus"。与C风格的字符串不同，string 的结尾没有结束标志'\0'。

String s3 = s2;//变量 s3 在定义的时候直接用 s2 进行初始化，因此 s3 的内容也是"c plus plus"。

string s4 (5, 's');//变量 s4 被初始化为由 5 个's'字符组成的字符串，也就是"sssss"。

从上面的代码可以看出，**string 变量可以直接通过赋值操作符=进行赋值**。string 变量也可以用C风格的字符串进行赋值，例如，s2 是用一个字符串常量进行初始化的，而 s3 则是通过 s2 变量进行初始化的。

**（2）转换为C风格的字符串**

虽然 C++ 提供了 string 类来替代C语言中的字符串，**但是在实际编程中，有时候必须要使用C风格的字符串（例如打开文件时的路径）**，为此，string 类为我们提供了一个转换函数 c\_str()，该函数能够将 string 字符串转换为C风格的字符串，并返回该字符串的 const 指针（const char\*）。

String path = “D:\\demo.text”;

FILE \*fp = fopen(path.c\_str(),”rt”);

**为了使用C语言中的 fopen() 函数打开文件，必须将 string 字符串转换为C风格的字符串。(也就是说，如果我使用c++风格的打开文件方式就可以不转换了？)**

**（3）string字符串的输入输出**

string 类重载了输入输出运算符，可以像对待普通变量那样对待 string 变量，也就是用>>进行输入，用<<进行输出.

例如：

String s;

Cin>>s;

Cout<<s<<endl;

在输入的时候如果有空格，那么默认会忽略空格，认为是输入结束！

**（4）访问字符串中的字符**

string 字符串也可以像C风格的字符串一样按照下标来访问其中的每一个字符。string 字符串的起始下标仍是从 0 开始。

String s = “1234567890”;

for(int i=0,len=s.length(); i<len; i++)

{

cout<<s[i]<<" ";

}

cout<<endl;

s[5] = '5';

cout<<s<<endl;

运行结果：  
1 2 3 4 5 6 7 8 9 0  
1234557890

**（5）字符串的拼接**

有了 string 类，我们可以使用+或+=运算符来直接拼接字符串，非常方便，再也不需要使用C语言中的 strcat()、strcpy()、malloc() 等函数来拼接字符串了，再也不用担心空间不够会溢出了。  
 用+来拼接字符串时，运算符的两边可以都是 string 字符串，也可以是一个 string 字符串和一个C风格的字符串，还可以是一个 string 字符串和一个字符数组，或者是一个 string 字符串和一个单独的字符。**注意！！必须得有个string型！！**请看下面的例子

string s1 = "first ";

string s2 = "second ";

char \*s3 = "third ";

char s4[] = "fourth ";

char ch = '@';

string s5 = s1 + s2;

string s6 = s1 + s3;

string s7 = s1 + s4;

string s8 = s1 + ch;

cout<<s5<<endl<<s6<<endl<<s7<<endl<<s8<<endl;

运行结果：

first second  
first third  
first fourth  
first @

**关于string的拼接还有一种使用append方法**

1. **、向string后面加c风格的字符串或者部分**

String s= “hello”;

Const char \*c = “out here”;

S.append(c);//// 把c类型字符串s连接到当前字符串结尾

S= “hello out here”;

S.append(c,3);// 把c类型字符串s的前n个字符连接到当前字符串结尾

s = “hello out”;

1. **、向string的后面加string或者部分**

string s1 = “hello “;

string s2 = “wide “;

string s3 = “world “;

S1.append(s2); //s1 = “hello wide “;

S1+=s3;//s1 = “hello wide world “

string s1 = “hello “, s2 = “wide world “;

s1.append(s2, 5, 5); ////把字符串s2中从5开始的5个字符连接到当前字符串的///结尾

s1 = “hello world”;  
string str1 = “hello “, str2 = “wide world “;  
str1.append(str2.begin()+5, str2.end()); //把s2的迭代器begin()+5和end()之间的

//部分连接到当前字符串的结尾  
str1 = “hello world”;

1. **、向string后面加多个字符**

string s1 = “hello “;  
s1.append(4,’!’); //在当前字符串结尾添加4个字符!  
s1 = “hello !!!!”;

**（6）string字符串的增删改查**

1. **、插入字符串**

insert() 函数可以在 string 字符串中指定的位置插入另一个字符串，它的一种原型为：

string& insert (size\_t pos, const string& str);

pos 表示要插入的位置，也就是下标；str 表示要插入的字符串，它可以是 string 字符串，也可以是C风格的字符串。

#include <iostream>

#include <string>

using namespace std;

int main(){

string s1, s2, s3;

s1 = s2 = "1234567890";

s3 = "aaa";

s1.insert(5, s3);

cout<< s1 <<endl;

s2.insert(5, "bbb");

cout<< s2 <<endl;

return 0;

}

运行结果：  
12345aaa67890  
12345bbb67890

**insert() 函数的第一个参数有越界的可能，如果越界，则会产生运行时异常**

**2)、删除字符串**

erase() 函数可以删除 string 中的一个子字符串。它的一种原型为：

string& erase (size\_t pos = 0, size\_t len = npos);

pos 表示要删除的子字符串的起始下标，len 表示要删除子字符串的长度。如果不指明 len 的话，那么直接删除从 pos 到字符串结束处的所有字符（此时 len = str.length - pos）。

#include <iostream>

#include <string>

using namespace std;

int main(){

string s1, s2, s3;

s1 = s2 = s3 = "1234567890";

s2.erase(5);

s3.erase(5, 3);

cout<< s1 <<endl;

cout<< s2 <<endl;

cout<< s3 <<endl;

return 0;

}

有读者担心，在 pos 参数没有越界的情况下， len 参数也可能会导致要删除的子字符串越界。但实际上这种情况不会发生，erase() 函数会从**len 的值**和**字符串长度减去 pos 的值**这两个值中取出最小的一个作为待删除子字符串的长度  
**说得简单一些，待删除字符串最多只能删除到字符串结尾。**

**3)、提取提取子字符串**

substr() 函数用于从 string 字符串中提取子字符串，它的原型为：

string substr (size\_t pos = 0, size\_t len = npos) const;

pos 为要提取的子字符串的起始下标，len 为要提取的子字符串的长度。

#include <iostream>

#include <string>

using namespace std;

int main(){

string s1 = "first second third";

string s2;

s2 = s1.substr(6, 6);

cout<< s1 <<endl;

cout<< s2 <<endl;

return 0;

}

**运行结果：  
first second third  
second**

**系统对 substr() 参数的处理和 erase() 类似：**

**如果 pos 越界，会抛出异常；**

**如果 len 越界，会提取从 pos 到字符串结尾处的所有字符。**

1. **、字符串查找**

**Find()、rfind()、find\_first\_of()函数这三个函数**

1. **Find()**

find()函数用于在 string 字符串中查找子字符串出现的位置，它其中的两种原型为：

size\_t find (const string& str, size\_t pos = 0) const;  
size\_t find (const char\* s, size\_t pos = 0) const;

第一个参数为待查找的子字符串，它可以是 string 字符串，也可以是C风格的字符串。第二个参数为**开始查找的位**置（下标）；如果不指明，则从第0个字符开始查找。

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s1 = "first second third";

string s2 = "second";

int index = s1.find(s2,5);

if(index < s1.length())

cout<<"Found at index : "<< index <<endl;

else

cout<<"Not found"<<endl;

return 0;

}

运行结果：  
Found at index : 6

**find() 函数最终返回的是子字符串第一次出现在字符串中的起始下标**。本例最终是在下标**6处找到了 s2 字符串**。如果没有查找到子字符串，那么会返回一个无穷大值 4294967295。

其实也就是在s1的“first ”这里是6个字符，包括一个空格，在第7个字符就是s2了，第7个字符就是下标为6.

1. **rfind()函数**

rfind() 和 find() 很类似，同样是在字符串中查找子字符串，**不同的是 find() 函数从第二个参数开始往后查找，而 rfind() 函数则最多查找到第二个参数处**，如果到了第二个参数所指定的下标还没有找到子字符串，则返回一个无穷大值4294967295。（注意与上者之间的区别）

#include <iostream>

#include <string>

using namespace std;

int main(){

string s1 = "first second third";

string s2 = "second";

int index = s1.rfind(s2,6);

if(index < s1.length())

cout<<"Found at index : "<< index <<endl;

else

cout<<"Not found"<<endl;

return 0;

}

**3、find\_first\_of()函数**

find\_first\_of() 函数用于查找子字符串和字符串共同具有的字符在字符串中首次出现的位置。请看下面的代码：

#include <iostream>

#include <string>

using namespace std;

int main(){

string s1 = "first second second third";

string s2 = "asecond";

int index = s1.find\_first\_of(s2);

if(index < s1.length())

cout<<"Found at index : "<< index <<endl;

else

cout<<"Not found"<<endl;

return 0;

}

运行结果：  
Found at index : 3

**本例中 s1 和 s2 共同具有的字符是 ’s’，该字符在 s1 中首次出现的下标是3，故查找结果返回3。**

**类似的还有：**

**find\_….of函数:**

find\_first\_of(args) 查找args中任何一个字符第一次出现的位置

find\_last\_of(args) 最后一个出现的位置

find\_fist\_not\_of(args) 查找第一个不在args中的字符

find\_last\_not\_of 查找最后一个不在args中出现的字符

1. 、交换

交换(swap())

string a = "Alpha Beta Gamma Delta" ;

string b = "abcdefghijklmn";

cout<< "a:"<< a << endl;

cout<< "b:"<< b << endl;

b.swap(a);

cout<< "a:"<< a << endl;

cout<< "b:"<< b << endl;

运行结果:

a: Alpha Beta Gamma Delta

b: abcdefghijklmn

a: abcdefghijklmn

b: Abcdefghijklmn

1. **、其他函数**

**push\_back函数**

void push\_back(value\_type\_Ch);

//在字符串之后插入一个字符pop\_back函ˉ数簓

void pop\_back();

//弹出字符串的最后一个字符 , 有效减少它的长度

**at函数**

reference at( size\_type index );

//at()函数返回一个引用，指向在index位置的字符. 如果index

//不在字符串范围内, at() 将报告"out of range"错误，并抛出out\_of\_range异常

**begin函数**

iterator begin();

//begin()函数返回一个迭代器,指向字符串的第一个元素

**end函数**

iterator end();

//返回一个迭代器，指向字符串的末尾(最后一个字符的下一个位置)

**c\_str函数**

const char \*c\_str();

//返回一个指向正规C字符串的指针, 内容与本字符串相同

**capacity函数**

size\_type capacity();

//返回在重新申请更多的空间前字符串可以

//容纳的字符数. 这个数字至少与 size()一样大

**copy函数**

size\_type copy( char \*str, size\_type num, size\_type index );

//拷贝自己的num个字符到str中（从索引index开始）。返回值是拷贝的字符数

**data函数簓**

const char \*data();

//返回指向自己的第一个字符的指针

**empty函数**

bool empty();

//如果字符串为空则empty()返回真(true)，否则返回假(false)

**get\_allocator函数**

allocator\_type get\_allocator();

//返回本字符串的配置器

**length函数**

size\_type length();

//返回字符串的长度. 这个数字应该和size()返回的数字相同

**max\_size**

size\_type max\_size();

//返回字符串能保存的最大字符数

**rbegin函数**

rbegin();//返回一个逆向迭代器，指向最后一个字符

**rend函数**

rend();//返回一个逆向迭代器，指向第一个元素的前一个位置

**reserve函数**

reserve( size\_type num );

//保留一定容量以容纳字符串（设置capacity值）

**resize函数**

void resize( size\_type num );

//改变本字符串的大小到num, 新空间的内容不确定

void resize( size\_type num, char ch );

//也可以指定用ch填充

**size函数**

size();

//返回字符串中字符的数量

**第三 LPCTSTR用法**

L表示long指针 ， 这是为了兼容Windows  3.1等16位操作系统遗留下来的，在win32中以及其他的32为操作系统中，  long指针和near指针及far修饰符都是为了兼容的作用。没有实际意义。P表示这是一个指针；**C表示是一个常量**；**T表示在Win32环境中，  有一个\_T宏，这个宏用来表示你的字符是否使用UNICODE,  如果你的程序定义了UNICODE或者其他相关的宏，那么这个字符或者字符串将被作为UNICODE字符串**，否则就是标准的ANSI字符串；STR表示这  个变量是一个字符串。

所以LPCTSTR就表示一个指向常固定地址的可以根据一些宏定义改变语义的字符串。同样，  **LPCSTR**就只能是一个ANSI字符串，在程序中我们大部分时间要使用带T的类型定义。

   LPCTSTR == const  TCHAR \*，

CString 和 LPCTSTR 可以说通用。 原因在于CString定义的自动类型转换，没什么奇特的，最简单的C++操作符重载而已。 

 常量字符串ansi和unicode的区分是由宏\_T来决定的。但是用\_T("abcd")时，  字符串"abcd"就会根据编译时的是否定一\_UNICODE来决定是char\* 还是 w\_char\*。 同样，TCHAR 也是相同目的字符宏。  看看定义就明白了。简单起见，**下面只介绍 ansi 的情况，unicode 可以类推。**

  ansi情况下，LPCTSTR 就是 const  char\*, 是常量字符串（不能修改的）。   
而LPTSTR 就是 char\*, 即普通字符串（非常量，可修改的）。

**这两种都是基本类型，  而CString 是 C++类， 兼容这两种基本类型是最起码的任务了。**   
      由于const char\*  最简单（常量，不涉及内存变更，操作迅速）， CString 直接定义了一个类型转换函数 operator LPCTSTR() {......}，  直接返回他所维护的字符串。

当你需要一个const char\* 而传入了CString时， C++编译器自动调用  CString重载的操作符 LPCTSTR()来进行隐式的类型转换。 当需要CString , 而传入了 const char\* 时（其实 char\*  也可以），C++编译器则自动调用CString的构造函数来构造临时的 CString对象。

所以LPCTSTR 和CString基本可以通用。

**但是 LPTSTR又不同了，他是 char\*，** 意味着你随时可能修改里面的数据，这就需要内存管理了(如字符串变长，原来的存贮空间就不够了，则需要重新调整分配内存)。   
所以 不能随便的将 const  char\* 强制转换成 char\* 使用。

**CString  转LPCTSTR:**  
CString cStr;  
const char  \*lpctStr=(LPCTSTR)cStr;

LPCTSTR lpctStr = (LPCTSTR)cStr;//这种不行吗？

**LPCTSTR转CString:**  
LPCTSTR lpctStr;  
CString  cStr=lpctStr;

TCHAR 就是当你的字符设置为什么就是什么

例如：程序编译为 ANSI， TCHAR 就是相当于 CHAR  
当程序编译为 UNICODE， TCHAR 就相当于 WCHAR

char :单字节变量类型，最多表示256个字符，

wchar\_t :宽字节变量类型，用于表示Unicode字符，

它实际定义在<string.h>里：typedef unsigned short wchar\_t。

为了让编译器识别Unicode字符串，必须以在前面加一个“L”,定义宽字节类型方法如下：

   wchar\_t c = `A' ;   
IMG_256wchar\_t \* p = L"Hello!" ;   
IMG_257wchar\_t a[] = L"Hello!" ;

**TCHAR / \_T( ) :**如果在程序中既包括ANSI又包括Unicode编码，需要包括头文件tchar.h。TCHAR是定义在该头文件中的宏，它视你是否定义了\_UNICODE宏而定义成：   
定义了\_UNICODE：    typedef wchar\_t TCHAR ;   
没有定义\_UNICODE： typedef char TCHAR ;

#ifdef UNICODE   
typedef char TCHAR;   
#else   
typede wchar\_t TCHAR;   
#endif   
\_T( )也是定义在该头文件中的宏，视是否定义了\_UNICODE宏而定义成：   
定义了\_UNICODE：    #define \_T(x) L##x   
没有定义\_UNICODE： #define \_T(x) x   
**注意：如果在程序中使用了TCHAR，那么就不应该使用ANSI的strXXX函数或者Unicode的wcsXXX函数了，而必须使用tchar.h中定义的\_tcsXXX函数**

**我看资料的时候，发现字符集的使用还挺麻烦的，尽量使用有宏的，如TCHAR，LPCTSTR，注意LPCTSTR是一个指针 const char \*或者const wchar**

**第四CString用法**

|  |  |
| --- | --- |
| 函数名称 | 功能 |
| 构造函数 | 产生或复制字符串 |
| 析构函数 | 销毁字符串 |
| int GetLength( ) const; | 返回字符串的长度，不包含结尾的空字符。 |
| void MakeReverse( ); | 颠倒字符串的顺序 |
| void MakeUpper( ); | 将小写字母转换为大写字母 |
| void MakeLower( ); | 将大写字母转换为小写字母 |
| int Compare( LPCTSTR lpsz ) const; | 区分大小写比较两个字符串，相等时返回0，大于时返回1，小于时返回-1 |
| int CompareNoCase( LPCTSTR lpsz ) const; | 不区分大小写比较两个字符串，相等时返回0，大于时返回1，小于时返回-1 |
| int Delete( int nIndex, int nCount = 1 ) | 删除字符，删除从下标nIndex开始的nCount个字符 |
| int Insert( int nIndex, TCHAR ch ) int Insert( int nIndex, LPCTSTR pstr ) | 在下标为nIndex的位置，插入字符或字符串。返回插入后对象的长度 |
| int Remove( TCHAR ch ); | 移除对象内的指定字符。返回移除的数目 |
| int Replace( TCHAR chOld, TCHAR chNew ); int Replace( LPCTSTR lpszOld, LPCTSTR lpszNew ); | 替换字串 |
| void TrimLeft( ); void TrimLeft( TCHAR chTarget ); void TrimLeft( LPCTSTR lpszTargets ); | 从左删除字符，被删的字符与chTarget或lpszTargets匹配，一直删到第一个不匹配的字符为止。  说明：如果没有参数,从左删除字符(\n\t空格等),至到遇到一个非此类字符. 当然你也可以指定删除那些字符. 如果指定的参数是字符串,那么遇上其中的一个字符就删除. \n  换行符 \t  TAB字符 |
| void Empty( ); | 清空 |
| BOOL IsEmpty( ) const; | 测试对象是否为空，为空时返回零，不为空时返回非零 |
| int Find( TCHAR ch ) const; int Find( LPCTSTR lpszSub ) const; int Find( TCHAR ch, int nStart ) const; int Find( LPCTSTR pstr, int nStart ) const; | 查找字串，nStart为开始查找的位置。未找到匹配时返回-1，否则返回字串的开始位置 |
| int FindOneOf( LPCTSTR lpszCharSet ) const; | 查找lpszCharSet中任意一个字符在CString对象中的匹配位置。未找到时返回-1，否则返回字串的开始位置 |
| CString SpanExcluding( LPCTSTR lpszCharSet ) const; | 返回对象中与lpszCharSet中任意匹配的第一个字符之前的子串 |
| CString SpanIncluding( LPCTSTR lpszCharSet ) const; | 从对象中查找与lpszCharSe中任意字符不匹配的字符，并返回第一个不匹配字符之前的字串 |
| int ReverseFind( TCHAR ch ) const; | 从后向前查找第一个匹配，找到时返回下标。没找到时返回-1 |
| void Format( LPCTSTR lpszFormat, ... ); void Format( UINT nFormatID, ... ); | 格式化对象，与C语言的sprintf函数用法相同 |
| TCHAR GetAt( int nIndex ) const; | 返回下标为nIndex的字符，与字符串的[]用法相同 |
| void SetAt( int nIndex, TCHAR ch ); | 给下标为nIndex的字符重新赋值 |
| CString Left( int nCount ) const; | 从左取字串，返回的字符串是前nCount个字符 |
| CString Right( int nCount ) const; | 从右取字串，返回值： 返回的字符串是最后nCount个字符。 |
| CString Mid( int nFirst ) const; CString Mid( int nFirst, int nCount ) const; | 从中间开始取字串 |
| LPTSTR GetBuffer( int nMinBufLength ); | 申请新的空间，并返回指针 |
| void ReleaseBuffer( int nNewLength = -1 ); | 使用GetBuffer后，必须使用ReleaseBuffer以更新对象内部数据 |
| LPTSTR GetBufferSetLength( int nNewLength ); | 申请新的空间，并返回指针 |
| BOOL LoadString( UINT nID ); | 返回值：如果加载资源成功则返回非零值；否则返回0。  nID  一个Windows 字符串资源ID。  **说明： 此成员函数用来读取一个由nID 标识的Windows 字符串资源，并放入一个已有CString 对象中。** |
|  |  |

CString是一个类，LPCTSTR是一个指针。

**（1）CString类对象的初始化**

CString str;

CString str1(\_T("abc"));

CString str2 = \_T("defg");

TCHAR szBuf[] = \_T("kkk");

CString str3(szBuf);

CString str4 = szBuf;

TCHAR \*p = \_T("1k2");

//TCHAR \* 转换为 CString

CString str5(p);

CString str6 = p;

CString str7(**str1**);

CString str8 = str7;

**（2）字符串基本操作：**

**1）长度：GetLength();**

CString str(\_T("abc"));

int len = str.GetLength(); //len == 3

**2）是否为空，即不含字符：IsEmpty();**

**3）清空字符串：Empty();**

CString str(\_T("abc"));

BOOL mEmpty = str.IsEmpty(); //mEmpty == FALSE

str.Empty();

mEmpty = str.IsEmpty(); //mEmpty == TRUE

1. **转换大小写：MakeUpper、MakeLower**

**5） 转换顺序：MakeReverse**

CString str(\_T("Abc"));

str.MakeUpper(); //str == ABC

str.MakeLower(); //str == abc

str.MakeReverse(); //str == cba

**6）字符串的连接：+、+=**

CString str(\_T("abc"));

str = \_T("de") + str + \_T("kp"); //str == deabckp

str += \_T("123"); //str == deabckp123

TCHAR szBuf[] = \_T("789");

str += szBuf; //str == deabckp123789

**7）字符串的比较：**

==、!=、(<、>、<=、>= 不常用)、Compare(区分大小写)、CompareNoCase(不区分大小写)

CString str1(\_T("abc"));

CString str2 = \_T("aBc");

if (str1 == str2){

MessageBox(\_T("str1 等于 str2"));

}else{

MessageBox(\_T("str1 不等于 str2"));

}

**8）字符串的查找**

**Find、ReverseFind、FindOneOf 三个函数可以实现字符串的查找操作**

Find 从指定位置开始查找指定的字符或者字符串，返回其位置，找不到返回 -1；

举例：

CString str(\_T("abcdefg"));

int idx = str.Find(\_T("cde"), 0); //idx 的值为2;

ReverseFind 从字符串末尾开始查找指定的字符，返回其位置，找不到返回 -1，虽然是从后向前查找，但是位置为从开始算起；

CString str(\_T("abcdefg"));

int idx = str.ReverseFind('e'); //idx 的值为4;

FindOneOf 查找参数中给定字符串中的任意字符，返回第一次出现的位置，找不到返回 -1；

CString str(\_T("abcabcd"));

int idx = str.FindOneOf(\_T("cbd")); //idx 的值为1;

**9）字符串的替换与删除**

Replace 替换 CString 对象中的指定的字符或者字符串，返回替换的个数，无匹配字符返回 0；

CString str(\_T("abcdabc"));

int num = str.Replace('b', 'k'); //str == akcdakc, num == 2

CString str(\_T("abcdabc"));

int num = str.Replace(\_T("bc"), \_T("kw")); //str == akwdakw, num == 2

Remove 删除 CString 对象中的指定字符，返回删除字符的个数，有多个时都会删除；

CString str(\_T("abcdabcb"));

int num = str.Remove('b'); //str == acdac, num == 3

Delete 删除 CString 对象中的指定位置的字符，返回处理后的字符串长度；

CString str(\_T("abcd"));

int num = str.Delete(1, 3); //str == a, num == 1

**10）字符串的提取**

Left、Mid、Right 三个函数分别实现从 CString 对象的 左、中、右 进行字符串的提取操作；

CString str(\_T("abcd"));

CString strResult = str.Left(2); //strResult == ab

strResult = str.Mid(1); //strResult == bcd

strResult = str.Mid(0, 2); //strResult == ab

strResult = str.Right(2); //strResult == cd

**11）单个字符的修改**

GetAt、SetAt 可以获取与修改 CString 对象中的单个 TCHAR 类型字符；

操作符也可以获取 CString 对象中的单个字符，但为只读的，不能进行修改；

CString str(\_T("abcd"));

str.SetAt(0, 'k'); //str == kbck

TCHAR ch = str.GetAt(2); //ch == c

**12）其他类型与 CString 对象类型的转换**

**1、格式化字符串：Format 方法，实现从 int、long 等数值类型、TCHAR、TCHAR \* 等类型向 CString 类型的转换；**

int num = 6;

CString str;

str.Format(\_T("%d"), num);

**2、CString 类型向 int 等数值类型、TCHAR \* 类型的转换：**

TCHAR \*pszBuf = str.GetBuffer();

str.ReleaseBuffer();

TCHAR \*p = (LPTSTR)(LPCTSTR)str;

CString str1(\_T("123"));

int num = \_ttoi(str1);

**13）CString 对象的 Ansi 与 Unicode 转换**

大家可以直接使用上节课给大家讲解的方法，此外这里给大家介绍一种从 Ansi 转换到 Unicode 的隐含方法：

//当前工程环境为Unicode

CString str;

str = "abc";

char \*p = "defg";

str = p;

**14） CString 对象字符串所占用的字节数**

CString str = \_T("abc");

 错误的求法：sizeof(CString)、sizeof(str)

 正确的求法：str.GetLength()\*sizeof(TCHAR)

**15）当作为 TCHAR \* 类型传参时**

确保申请了足够用的空间，比如使用 GetModuleFileName 函数；

**第四 常用的对字符串的操作函数**

|  |  |
| --- | --- |
| 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。 |
| 7 | Memcpy, **C库函数 #include<string.h>**  Void \*memcpy(void \*str1, const void \*str2, size\_t n);  从**存储区** str2 复制 n 个字节到**存储区**str1。返回值是该函数**返回一个指向目标存储区 str1 的指针** |
| 8 | Memset **C库函数 #include<string.h>**  Void \*memset(void \*str,int c,size\_t n);**复制字符 c**（一个无符号字符）到参数 st**r 所指向的字符串的前 n 个字符**  返回值：该值返回一个指向存储区 str 的指针。 |
| 9 | lstrcpy  **winAPI的函数**，函数原型：LPTSTR lstrcpy(LPWSTR lpString1,LPCWSTR lpString2);  lpString1：指向接收由参数lpString2指向字符串内容的缓冲区。缓冲区必须足够大来容纳字符串，还包括最后的NULL终止符。  lpString2：指向待复制的以NULL为终止符的字符串。  返回值：若函数运行成功，返回值是缓冲区的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)；若函数运行失败，返回值是NULL。若想获得更多错误信息，请调用GetLastError函数。 |
| 10 | \_tcscpy  \_tcscpy是一个拷贝字符串，源字符串指针是strSource，有strcpy、wcscpy、\_mbscpy三种类型，这三种形式都返回目标字符串。 |
| 11 | lstrcmp  在windows开发中少使用strcmp多用lstrcmp，  函数功能：比较两个字符串，此比较不区分大小写  函数原型：int lstrcmp(LPCTSTR lpString1,LPCTSTR lpString2);  lpString1：指向将被比较的第一个字符串。  lpString2：指向将被比较的第二个字符串。 返回值：若第一个字符串比第二个字符串小则返回值为负；若第一个字符串比第二个字符串大则返回值为正；若两个字符串相等则返回值为0。将被比较的第二个字符串。 |

下面就是关于上面函数的例子以及讲解：

#include <iostream>

#include<stdio.h>

#include<string.h>

int main()

{

const char src[50] = "http://www.runoob.com";

char dest[50] = "kdsifsh";

*memcpy*(dest, src, 5);

//memcpy(dest, src, strlen(src) + 1);

*printf*("dest = %s\n", dest);

return(0);

}

**dest = http:sh**

memset

C库函数 #include<string.h>

Void \*memset(void \*str,int c,size\_t n);**复制字符 c**（一个无符号字符）到参数 st**r 所指向的字符串的前 n 个字符**

**返回值：该值返回一个指向存储区 str 的指针。**

#include <iostream>

#include<stdio.h>

#include<string.h>

int main()

{

char str[50];

*strcpy\_s*(str, "this is string.h library function");

*puts*(str);

*memset*(str, '$', 7);

*puts*(str);

return 0;

}

This is string.h library function

$$$$$$$ string.h library function

**从结果上看，两个是一样的。**

**第五 结构**

struct type\_name {

member\_type1 member\_name1;

member\_type2 member\_name2;

member\_type3 member\_name3;

. .

} object\_names;

**object\_names是对象。**

**一 结构作为函数参数**

**（一） 直接将结构作为参数传递，并在需要的时候作为返回值返回，使用的情况较小**

#include "pch.h"

#include<cstring>

#include <iostream>

struct travel\_time

{

int hours;

int mins;

};

const int Mins\_per\_hr = 60;

travel\_time sum(travel\_time t1, travel\_time t2);

void show\_time(travel\_time t);

int *main*()

{

using namespace *std*;

travel\_time day1 = { 5, 45 }; // 5 hrs, 45 min

travel\_time day2 = { 4, 55 }; // 4 hrs, 55 min

travel\_time trip = sum(day1, day2);

*cout* << "Two-day total: ";

show\_time(trip);

travel\_time day3 = { 4, 32 };

*cout* << "Three-day total: ";

show\_time(sum(trip, day3));

// cin.get();

return 0;

}

travel\_time sum(travel\_time t1, travel\_time t2) //结构作为函数参数直接传入

{

travel\_time total;

total.mins = (t1.mins + t2.mins) % Mins\_per\_hr;

total.hours = t1.hours + t2.hours +

(t1.mins + t2.mins) / Mins\_per\_hr;

return total;

//<span style = "white-space:pre;"> < / span>//结构作为函数结果直接传出

}

void show\_time(travel\_time t)

{

using namespace *std*;

*cout* << t.hours << " hours, "

<< t.mins << " minutes\n";

}

**（二）传递结构的地址，这种方法适合于结构数据较大时**

**（1）调用函数时，将结构的地址（&）而不是结构本身传给它**

**（2）将形参声明为指向poor的指针，由于函数不应该修改结构，因此使用了const**

**（3）由于形参是指针而不是结构，因此应该用间接成员运算符（->）,而不是成员运算符（.）。**

#include <iostream>

#include <cmath>

// structure templates

struct polar

{

double distance; // distance from origin

double angle; // direction from origin

};

struct rect

{

double x; // horizontal distance from origin

double y; // vertical distance from origin

};

// prototypes

void rect\_to\_polar(const rect \* pxy, polar \* pda);

void show\_polar(const polar \* pda);

int *main*()

{

using namespace *std*;

rect rplace;

polar pplace;

*cout* << "Enter the x and y values: ";

while (*cin* >> rplace.x >> rplace.y)

{

rect\_to\_polar(&rplace, &pplace); // pass addresses

show\_polar(&pplace); // pass address

*cout* << "Next two numbers (q to quit): ";

}

*cout* << "Done.\n";

return 0;

}

// show polar coordinates, converting angle to degrees

void show\_polar(const polar \* pda)

{

using namespace *std*;

const double Rad\_to\_deg = 57.29577951;

*cout* << "distance = " << pda->*distance*;

*cout* << ", angle = " << pda->angle \* Rad\_to\_deg;

*cout* << " degrees\n";

}

// convert rectangular to polar coordinates

void rect\_to\_polar(const rect \* pxy, polar \* pda)//结构通过指针传入函数

{

using namespace *std*;

pda->*distance* =

*sqrt*(pxy->x \* pxy->x + pxy->y \* pxy->y);

pda->angle = *atan2*(pxy->y, pxy->x); //通过在函数中修改结构的成员

}

**（三） 按引用传递**

参考c++中引用变量的使用

1. **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[50];

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;

也就是说pint32是long int的一个别名

X,y,z都是指向长整形long int的指针。

**第六 共用体（略）**

**第七 枚举**

**一 枚举类型的定义**

++中的一种派生数据类型，由用户定义的若干枚举常量的集合。枚举定义的格式是  
enum<类型名>{<枚举常量表>};

二 **枚举格式说明**

**（1）**关键字enum--指明后面的标志符是枚举类型的名字

**（2）**枚举常量表--由枚举常量构成。"枚举常量"或称"枚举成员"，是以标识符形式表示的整型量，表示枚举类型的取值。枚举常量表列出枚举类型的所有取值，各枚举常量之间以"，"间隔，且必须各不相同。取值类型与条件表达式相同。

**三 应用举例**

enum color\_set1{RED,BLUE,WHITE,BLACK};//定义枚举类型color\_set1

enum week{SUN,MON ,TUE, WED,THU,FRI,SAT}//定义枚举类型week

**四 重要提示**

**（1）**枚举常量代表该枚举类型的变量可能取的值，编译系统为每个枚举常量指定一个整数值，默认状态下，这个整数就是所列举元素的序号，序号从0开始。 可以在定义枚举类型时为部分或全部枚举常量指定整数值，在指定值之前的枚举常量仍按默认方式取值，而指定值之后的枚举常量按依次加1的原则取值。 各枚举常量的值可以重复。

**（2）**枚举变量可以输出，但不能输入，cout>>color3;//非法

**（3）**不能直接将常量赋给枚举变量，color3=1；//非法

**（4）**不同类型的枚举变量之间不能相互赋值，color1=color3;//非法

**（5）**枚举变量的输入和输出一般采取switch语句。

**第八指针**

**一 指针定义**

通过指针，可以简化一些 C++ 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。**指针**是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

type \* var-name;

Type是指针的基类型，它必须是一个有效的c++数据类型。

**二 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。例如 int \*ptr =NULL;

**三 C++指针的算术运算**

指针是一个用数值表示的地址。因此，是可以对指针执行算术运算。可以对指针进行+,-,++,--，四种运算。

假如ptr是指向地址1000的整型指针，是一个32位（4\*8）的整数，让我们对该指针执行ptr++运算，那么，ptr将指向1004的位置。因为ptr每增加一次，它将指向下一个整数位置，也就是向后移动4个字节。

**一般情况下会使用指针代替数组，因为变量指针可以递增，而数组不能递增，因为数组是一个常量指针。**

**四 C++指针vs数组**

指针和数组是密切相关的。事实上，指针和数组在很多情况下是可以互换的。例如，一个指向数组开头的指针，可以通过使用指针的算术运算或数组索引来访问数组。

#include <iostream>

using namespace *std*;

const int *MAX* = 3;

int main()

{

int var[*MAX*] = { 10, 100, 200 };

int \*ptr;

// 指针中的数组地址

ptr = var;

for (int i = 0; i < *MAX*; i++)

{

*cout* << "var[" << i << "]的内存地址为 ";

*cout* << ptr << *endl*;

*cout* << "var[" << i << "] 的值为 ";

*cout* << \*ptr << *endl*;

// 移动到下一个位置

ptr++;

}

return 0;

}

运行结果是

var[0]的内存地址为 0x7fff59707adc

var[0] 的值为 10

var[1]的内存地址为 0x7fff59707ae0

var[1] 的值为 100

var[2]的内存地址为 0x7fff59707ae4

var[2] 的值为 200

1. 指针和数组并不是完全互换的。如下

#include <iostream>

using namespace *std*;

const int *MAX* = 3;

int main()

{

int var[*MAX*] = { 10, 100, 200 };

for (int i = 0; i < *MAX*; i++)

{

\*var = i; // 这是正确的语法

var++; // 这是不正确的

}

return 0;

}

指针运算符\*应用到 var 上是完全可以的，但修改 var 的值是非法的。这是因为 var 是一个指向数组开头的常量，不能作为左值。由于一个数组名对应一个指针常量，只要不改变数组的值，仍然可以用指针形式的表达式。例如，下面是一个有效的语句，把 var[2] 赋值为 500。

\*（ var +2）=500;

1. **C++指针数组(略)**
2. **C++指向指针的指针（略）**
3. **函数指针**

**（1）什么是函数指针？**

在用的过程中有遇到回调函数，就接触到了函数指针，函数指针也是一种指针。它是一种指向某种特定类型的函数，函数的类型由参数以及返回类型共同决定，与函数名字没有关系。

举例如下：

int add(int nLeft,int nRight);//函数定义

 该函数类型为int(int,int),要想声明一个指向该类函数的指针，只需用指针替换函数名即可：

int (\*pf)(int,int);//未初始化

则pf可指向int(int,int)类型的函数。pf前面有\*，说明pf是指针，右侧是形参列表，表示pf指向的是函数，左侧为int，说明pf指向的函数返回值为int。则pf可指向int(int,int)类型的函数。而add类型为int(int,int),则pf可指向add函数。

pf = add;//通过赋值使得函数指针指向某具体函数

**注意：\*pf两端的括号必不可少，否则若为如下定义：**

**int \*pf(int,int);//此时pf是一个返回值为int\*的函数，而非函数指针**

**（2）函数指针的使用(c语言中的使用)**

**1） 函数指针的定义**

**int (\*pf)(int,int);**

**或者**

**typedef int (\*PF)(int,int);**

**PF pf; //此时，为指向某种类型函数的函数指针类型，而不是具体指针，用它可定义具体指针**

1. **函数指针的使用**

pf =add;

pf(100,100);//与其指向的函数用法无异

(\*pf)(100,100);//此处\*pf两端括号必不可少

**注意：add类型必须与pf可指向的函数类型完全匹配**

**3）函数指针作为形参**

//第二个形参为函数类型，会自动转换为指向此类函数的指针

Void fuc(int nValue,int pf(int,int));

//等价的声明，显示的将形参定义为指向函数的指针

Void fuc(int nValue,int (\*pf)(int,int));

Void fuc(int nValue,PF);

 //形参中有函数指针的函数调用，以fuc为例：

pf = add;//pf是函数指针

fuc(1,add);//add自动转换为函数指针

fuc(1,pf);

**4） 返回指向函数的指针**

**（3）c++中函数指针的使用**

**注意：c中所有的关于函数指针的情况都能被c++兼容，**

**C++主要是类，继承，重载。从这三个方面讲解一下函数指针的不同**

1. **普通成员函数指针的举例**

class A//定义类A

{

private:

       int add(int nLeft, int nRight)

       {

              return (nLeft + nRight);

       }

public:

       void fuc()

       {

              printf("Hello  world\n");

       }

};

**typedef void(A::\*PF1)();//指针名前需加上类名限定**

**PF1 pf1 = &A::fuc; //必须有&**

**A a;//成员函数地址解引用必须附驻与某个对象地址，所以必须创建一个队形**

1. **\*pf1)();//使用成员函数指针调用函数**

**2)继承中函数指针的举例**

**class A**

**{**

**public:**

**void fuc()**

**{**

**printf("Hello fuc()\n");**

**}**

**void fuc2()**

**{**

**printf("Hello A::fuc2()\n");**

**}**

**};**

**class B:public A**

**{**

**public:**

**virtual void fuc2()**

**{**

**printf("Hello B::fuc2()\n");**

**}**

**};**

**typedef void(A::\*PF1)();**

**typedef void(B::\*PF2)();**

**PF1 pf1 = &A::fuc;**

**int main()**

**{**

**A a;**

**B b;**

**(a.\*pf1)();  //调用A::fuc**

**(b.\*pf1)();   //调用A::fuc**

**pf1 = &A::fuc2;**

**(a.\*pf1)();  //调用A::fuc2**

**(b.\*pf1)();  //调用A::fuc2**

**PF2 pf2 = &A::fuc2;**

**(b.\*pf2)(); //调用A::fuc2**

**}**

**有点糊涂了和回调函数的使用那块！！**

**第九 引用**

**（一）定义**

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。

1. **C++引用与指针区别**

**（1）不存在空引用，引用必须连接到一块合法的内存。**

**（2）一旦引用被初始化为一个对象，就不能被指向另一个对象，指针可以在任何时候指向另一个对象**

**（3）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。**

例如 int i =17；

Int& r =i;

Double& s =d;

在这里&读作引用。

**引用在函数中的使用**

把引用作为参数

把引用作为返回值

（对比指针的函数使用

**第十vector和array类（在后面讨论模板一起，还有数组，string一起）**

1. **循环和关系表达式**

**第一 for循环**

**第二 while循环**

**第三 do..while循环**

**第四 循环控制语句（break，continue,goto语句）**

**第一 for循环**

**For 循环允许您编写一个执行特定次数的循环的重复控制结构。**

**C++ 中 for 循环的语法：**

**for ( init; condition; increment )**

**{**

**statement(s);**

**}**

**下面是 for 循环的控制流：**

**（1）**init 会首先被执行，且只会执行一次。这一步允许您声明并初始化任何循环控制变量。您也可以不在这里写任何语句，只要有一个分号出现即可。

**（2）**接下来，会判断 condition。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 for 循环的下一条语句。

**（3）**在执行完 for 循环主体后，控制流会跳回上面的 increment 语句。该语句允许您更新循环控制变量。该语句可以留空，只要在条件后有一个分号出现即可。

**（4）**条件再次被判断。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，for 循环终止。

**第二 while循环**

**只要给定的条件为真，while 循环语句会重复执行一个目标语句。**

C++ 中 while 循环的语法：

while(condition)

{

statement(s);

}

在这里，statement(s) 可以是一个单独的语句，也可以是几个语句组成的代码块。condition 可以是任意的表达式，当为任意非零值时都为真。当条件为真时执行循环。当条件为假时，程序流将继续执行紧接着循环的下一条语句。

**第三 do..while循环**

不像 for 和 while 循环，它们是在循环头部测试循环条件。do...while 循环是在循环的尾部检查它的条件。

do...while 循环与 while 循环类似，但是 do...while 循环会确保至少执行一次循环。

**语法**

C++ 中 do...while 循环的语法：

do{

statement(s);

}while( condition );

请注意，条件表达式出现在循环的尾部，所以循环中的 statement(s) 会在条件被测试之前至少执行一次。如果条件为真，控制流会跳转回上面的 do，然后重新执行循环中的 statement(s)。这个过程会不断重复，直到给定条件变为假为止。

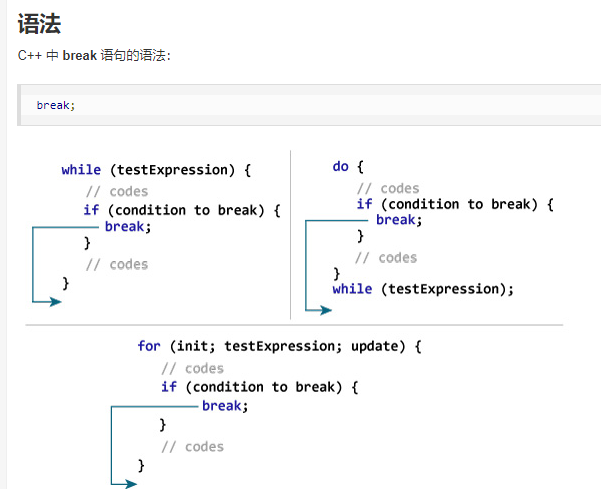
**第四 循环控制语句（break，continue,goto语句）**

**一 C++ 中 break 语句有以下两种用法：**

**（1）当 break 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。**

**（2）它可用于终止 switch 语句中的一个 case。**

如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），break 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。



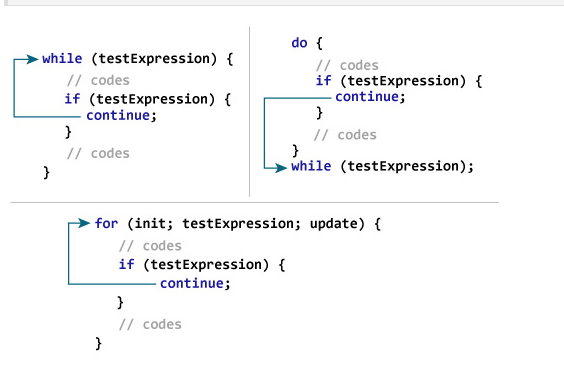
**二 Continue语句**

C++ 中的 continue 语句有点像 break 语句。但它不是强迫终止，continue 会跳过当前循环中的代码，强迫开始下一次循环。对于 for 循环，continue 语句会导致执行条件测试和循环增量部分。对于 while 和 do...while 循环，continue 语句会导致程序控制回到条件测试上。

**语法**

C++ 中 continue 语句的语法：

continue;



**第6章 分支语句**

**第一 if语句（略）**

**第二 if..else语句（略）**

**第三 嵌套if语句（略）**

**第四 switch语句（略）**

**第五 嵌套switch语句（略）**

**第六 数字**

**第六 数字**

**一 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。 |

**二 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 <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

int main ()

{ int i,j; // 设置种子

srand( (unsigned)time( NULL ) ); /\* 生成 10 个随机数 \*/

for( i = 0; i < 10; i++ )

{ // 生成实际的随机数

j= rand();

cout <<"随机数： " << j << endl;

}

return 0;

}

1. **八章函数**

**第一 定义函数，函数原型（声明）和调用**

**第二 函数参数和传递参数（按值传递，指针传递，引用传递），默认参数**

**第三 函数和数组**

**第四 函数与C-风格字符串（略）**

**第五 函数与结构（略）**

**第六 递归函数**

**第七 c++内联函数（略）**

**第八 回调函数**

**第一 定义函数，函数原型（声明）和调用**

**一 定义函数**

C++ 中的函数定义的一般形式如下：

return\_type function\_name( parameter list )

{

body of the function

}

在 C++ 中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

**（一）返回类型：A.** 一个函数可以返回一个值。函数返回的值的数据类型是return\_type 。可以是常量，变量，也可以是表达式。**返回值类型除了数组，其他类型都可以。可以将数组作为结构，对象的组成部分返回。B.有些函数执行所需的操作而不返回值，在这种情况下，return\_type 是关键字 void。**

**（二）函数名称：**这是函数的实际名称。函数名和参数列表一起构成了函数签名。

**（三）参数：**参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。

**（四）函数主体**：函数主体包含一组定义函数执行任务的语句。

**二 函数声明**

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

return\_type function\_name( parameter list );

例如：int max（int numl , int num2）;

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

**三 调用函数**

创建 C++ 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。

**注意：同一工程中不同项目之间的调用函数方法，两个方法：**

**1.相对路径 "..\rfb\Logger\_stdio.h"之类的，根据自己情况调整，灵活运用“.\”和“..\”，他俩什么意思您应该知道的  
  
2.在调用工程中加附加包含目录，这样可以直接<Logger\_stdio.h>**

**第二 函数参数和传递参数（按值传递，指针传递，引用传递），默认参数**

**一 函数参数**

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有三种向函数传递参数的方式：

**（一）传值调用**

**向函数传递参数的传值调用方法，把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。**

**默认情况下，C++ 使用传值调用方法来传递参数。一般来说，这意味着函数内的代码不会改变用于调用函数的实际参数。函数 swap() 定义如下：**

#include "pch.h"

#include <iostream>

// 函数声明

void swap(int x, int y);

int *main*()

{

using namespace *std*;

// 局部变量声明

int a = 100;

int b = 200;

*cout* << "交换前，a 的值：" << a << *endl*;

*cout* << "交换前，b 的值：" << b << *endl*;

// 调用函数来交换值

swap(a, b);

*cout* << "交换后，a 的值：" << a << *endl*;

*cout* << "交换后，b 的值：" << b << *endl*;

return 0;

}

void swap(int x, int y)

{

int temp;

temp = x; /\* 保存 x 的值 \*/

x = y; /\* 把 y 赋值给 x \*/

y = temp; /\* 把 x 赋值给 y \*/

return;

}

**结果**

**交换前，a 的值： 100**

**交换前，b 的值： 200**

**交换后，a 的值： 100**

**交换后，b 的值： 200**

**为什么？？**

**用于接受传递值的变量叫做形参，在这里就是void swap(int x, int y);中的x，y。传递给函数的值被称为实参，在这里就是swap(a, b);中的a,b。在调用swap(a, b);时，swap（）函数就会创建int x，int y。就会把a赋值给x，b赋值给y。那么在swap函数中运算的是x，y，而不是a ，b。因为a，b没有进行计算，进行计算的是a，b的副本。**

**在函数swap(x, y)中声明的变量（包括参数）是函数swap私有的，当这个函数被调用的时候，那么计算机会给这些变量分配内存，当函数结束时候，计算机会释放这些变量使用的内存。这些变量是局部变量。**

**（二）指针调用**

**向函数传递参数的指针调用方法，把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。**

**按指针传递值，参数指针被传递给函数，就像传递其他值给函数一样。因此相应地，在下面的函数 swap() 中，您需要声明函数参数为指针类型，该函数用于交换参数所指向的两个整数变量的值。**

#include "pch.h"

#include <iostream>

using namespace *std*;

// 函数声明

void swap(int \*x, int \*y);

int *main*()

{

// 局部变量声明

int a = 100;

int b = 200;

*cout* << "交换前，a 的值：" << a << *endl*;

*cout* << "交换前，b 的值：" << b << *endl*;

/\* 调用函数来交换值

\* &a 表示指向 a 的指针，即变量 a 的地址

\* &b 表示指向 b 的指针，即变量 b 的地址

\*/

swap(&a, &b);

*cout* << "交换后，a 的值：" << a << *endl*;

*cout* << "交换后，b 的值：" << b << *endl*;

return 0;

}

void swap(int \*x, int \*y)

{

int temp;

temp = \*x; /\* 保存地址 x 的值 \*/

\*x = \*y; /\* 把 y 赋值给 x \*/

\*y = temp; /\* 把 x 赋值给 y \*/

return;

}

交换前，a 的值： 100

交换前，b 的值： 200

交换后，a 的值： 200

交换后，b 的值： 100

**注意与之前的指针章节对比，返回指针的函数**

**声明指针函数**

int \* myFunction()

{...

}

**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。**

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace *std*;

// 要生成和返回随机数的函数

int \* getRandom()

{

**static int r[10];**

// 设置种子

*srand*((unsigned)*time*(*NULL*));

for (int i = 0; i < 10; ++i)

{

r[i] = *rand*();

*cout* << r[i] << *endl*;

}

**return r;**

}

// 要调用上面定义函数的主函数

int *main*()

{

// 一个指向整数的指针

int \*p;

p = getRandom();

for (int i = 0; i < 10; i++)

{

*cout* << "\*(p + " << i << ") : ";

*cout* << \*(p + i) << *endl*;

}

return 0;

}

1. **引用调用**

向函数传递参数的引用调用方法，把引用的地址复制给形式参数。在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

按引用传递值，参数引用被传递给函数，就像传递其他值给函数一样。因此相应地，在下面的函数 swap() 中，您需要声明函数参数为引用类型，该函数用于交换参数所指向的两个整数变量的值。

#include <iostream>

using namespace *std*;

void add(int a,int b);

void adds(int& a, int& b);

int *main*()

{

int c = 10;

int d = 20;

add(c, d);

*cout* << "c = "<<c << *endl*;

adds(c, d);

*cout* << "c = "<<c << *endl*;

return 0;

}

void add(int a,int b)

{

a = a + b;

}

void adds(int& a, int& b)

{

a = a + b;

}

**结果**

**C=10；**

**C = 30;**

**（三.1）注意：指针调用与引用调用的区别：**

**（1）指针和引用的定义和性质区别**

1)指针：**指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元**；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。如：

int a=1;int \*p=&a;

int a=1;int &b=a;

上面定义了一个整形变量和一个指针变量p，该指针变量指向a的存储单元，即p的值是a存储单元的地址。

而下面2句定义了一个整形变量a和这个整形a的引用b，**事实上a和b是同一个东西，在内存占有同一个存储单元。**

2)可以有const指针，但是没有const引用；

3)指针可以有多级，但是引用只能是一级（int \*\*p；合法 而 int &&a是不合法的）

4)指针的值可以为空，但是引用的值不能为NULL，**并且引用在定义的时候必须初始化；**

5)指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。

6)"sizeof引用"**得到的是所指向的变量(对象)的大小**，而"sizeof指针"**得到的是指针本身的大小**；

7)指针和引用的自增(++)运算意义不一样；

**（2）指针和引用作为函数参数进行传递时的区别**

1)指针作为参数进行传递：

[IMG_256](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

#include<iostream>

using namespace std;

void swap(int \*a,int \*b)

{

　　int temp=\*a;

　　\*a=\*b;

　　\*b=temp;

}

int main(void)

{

　　int a=1,b=2;

　　swap(&a,&b);

　　cout<<a<<" "<<b<<endl;

　　system("pause");

　　return 0;

}

[IMG_257](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

**结果为2 1；**

用指针传递参数，可以实现对实参进行改变的目的，是因为传递过来的是实参的地址，因此使用\*a实际上是取存储实参的内存单元里的数据，即是对实参进行改变，因此可以达到目的。

再看一个程序;

[IMG_258](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

#include<iostream>using namespace std;

void test(int \*ps)

{

　　int a=1;

　　ps=&a;

　　cout<<ps<<" "<<\*ps<<endl;

}

int main(void)

{

int \*p=NULL;

test(p);

if(p==NULL)

cout<<"指针p为NULL"<<endl;

system("pause");

return 0;

}

[IMG_259](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

**运行结果为：**

**0x22ff44 1**

**指针p为NULL**

大家可能会感到奇怪，怎么回事，不是传递的是地址么，怎么p回事NULL？事实上，在main函数中声明了一个指针p，并赋值为NULL，当调用test函数时，事实上传递的也是地址，只不过传递的是指地址。**也就是说将指针作为参数进行传递时，事实上也是值传递，只不过传递的是地址。**当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，**即上面程序main函数中的p和test函数中使用的p不是同一个变量，存储2个变量p的单元也不相同（只是2个p指向同一个存储单元），那么在test函数中对p进行修改，并不会影响到main函数中的p的值。**

如果要想达到也同时修改的目的的话，就得使用引用了。

2.将引用作为函数的参数进行传递。

在讲引用作为函数参数进行传递时，实质上传递的是实参本身，即传递进来的不是实参的一个拷贝，因此对形参的修改其实是对实参的修改，所以在用引用进行参数传递时，不仅节约时间，而且可以节约空间。

看下面这个程序：

[IMG_260](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

#include<iostream>using namespace std;

void test(int &ab)

{

　　cout<<&ab<<" "<<ab<<endl;

}

int main(void)

{

int a=1;

cout<<&a<<" "<<a<<endl;

test(a);

system("pause");

return 0;

}

[IMG_261](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

输出结果为： 0x22ff44 1

          0x22ff44 1

再看下这个程序：

**这足以说明用引用进行参数传递时，事实上传递的是实参本身，而不是拷贝。**

所以在上述要达到同时修改指针的目的的话，就得使用引用了。

[IMG_262](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

#include<iostream>using namespace std;

void test(int \*&p)

{

　　int a=1;

　　p=&a;

　　cout<<p<<" "<<\*p<<endl;

}

int main(void)

{

int \*p=NULL;

test(p);

if(p!=NULL)

cout<<"指针p不为NULL"<<endl;

system("pause");

return 0;

}

[IMG_263](https://www.cnblogs.com/dolphin0520/archive/2011/04/03/javascript:void(0);)

**输出结果为：0x22ff44 1**

**指针p不为NULL**

**（三.2）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。例如，请看下面这个简单的程序：**

#include "pch.h"

#include <iostream>

using namespace *std*;

double vals[] = { 10.1, 12.6, 33.1, 24.1, 50.0 };

double& setValues(int i)

{

return vals[i]; // 返回第 i 个元素的引用

}

// 要调用上面定义函数的主函数

int *main*()

{

*cout* << "改变前的值" << *endl*;

for (int i = 0; i < 5; i++)

{

*cout* << "vals[" << i << "] = ";

*cout* << vals[i] << *endl*;

}

setValues(1) = 20.23; // 改变第 2 个元素

setValues(3) = 70.8; // 改变第 4 个元素

*cout* << "改变后的值" << *endl*;

for (int i = 0; i < 5; i++)

{

*cout* << "vals[" << i << "] = ";

*cout* << vals[i] << *endl*;

}

return 0;

}

**（四）默认参数**

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。

这是通过在函数定义中使用赋值运算符来为参数赋值的。调用函数时，如果未传递参数的值，则会使用默认值，如果指定了值，则会忽略默认值，使用传递的值。

**第三 函数和数组**

与函数与指针类似

**第四 函数与C-风格字符串（略）**

**第五 函数与结构（略）**

**第六 递归函数**

**有一个例子加上去**

#include <iostream>

#include <vector>

using namespace *std*;

void do\_digui(*vector*<*vector*<*string*> >& vec, int cur\_indx, *vector*<*string* >& tmp, *vector*<*vector*<*string*> >& ret)

{

if (cur\_indx >= vec.*size*())

{

return;

}

*vector*<*string*> cur\_vec = vec[cur\_indx];

int size = cur\_vec.*size*();

for (int i = 0; i < size; i++)

{

*vector*<*string* > t = tmp;

t.*push\_back*(cur\_vec[i]);

if (vec.*size*() == (cur\_indx + 1))//已经取到vec中的最后一个数据了，需要把前面遍历的结果保存起来。

ret.*push\_back*(t);

else

do\_digui(vec, cur\_indx + 1, t, ret);

}

}

int *main*() {

*vector*<*vector*<*string*> > vec\_data;

*vector*<*string* > vec1;

*vector*<*string* > vec2;

*vector*<*string* > vec3;

vec1.*push\_back*("a");

vec1.*push\_back*("b");

vec1.*push\_back*("c");

vec2.*push\_back*("1");

vec2.*push\_back*("2");

// vec2.push\_back("3");

vec3.*push\_back*("A");

vec3.*push\_back*("B");

vec3.*push\_back*("C");

vec\_data.*push\_back*(vec1);

vec\_data.*push\_back*(vec2);

vec\_data.*push\_back*(vec3);

*vector*<*string* > tmp;

*vector*<*vector*<*string*> > ret;

do\_digui(vec\_data, 0, tmp, ret);

for (int i = 0; i < ret.*size*(); i++)

{

*vector*<*string* > tm = ret[i];

for (int j = 0; j < tm.*size*(); j++)

{

*cout* << tm[j] << " ";

}

*cout* << *endl*;

}

return 0;

}

这是一个排列组合算法的递归。可以总结到算法题里面去，排列组合方式！！

**第七内联函数**

**第八回调函数**

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这个函数是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

**回调函数机制：  
（1）定义一个函数（普通函数即可）；  
（2）将此函数的地址注册给调用者；  
（3）特定的事件或条件发生时，调用者使用函数指针调用回调函数。**

**回调函数的好处：**

**解耦，更改函数地址就可以实现不同的功能**

#include <iostream>

#include <stdio.h>

using namespace std;

typedef int (\*callback)(int, int);//int类型的函数指针，callback被声明成了一种类型

//2、将此函数的地址注册给调用者；

int getValue(int a, int b, callback p) //模拟API函数或DLL函数

{

return (\*p)(a, b);

}

int add(int a, int b) //自己的实现函数要和函数指针形式相一致

{

return a + b;

}

int minus1(int a, int b)

{

return a - b;

}

int main()

{

int res1 = getValue(4, 2, add);

std::cout << " res1 = " << res1<<"\n";

int res2 = getValue(10, 8, minus1);

std::cout << " res2 = " << res2<<"\n";

return 0;

}

如果是两个类之间如何使用回调函数？

PlayerRelay.h

#pragma once

#include <istream>

using namespace *std*;

//声明函数指针，它也是一个指针，

typedef void(\*ts\_data\_input\_t)(void\* user, const char\* s,int length);

class PlayerRelay

{

public:

PlayerRelay();

~PlayerRelay();

public:

//注册回调函数

void SetCbk(ts\_data\_input\_t ts\_data\_input,void\*user);

//数据输出函数,这里会调用Player中的回调函数

//ts\_data\_input(void\* user,char\* s,int length);

void transport\_data();

public:

ts\_data\_input\_t m\_ts\_data\_input;

void\* user;

protected:

private:

};

playerRelay.cpp

#include "playerRelay.h"

PlayerRelay::PlayerRelay()

{

}

PlayerRelay::~PlayerRelay()

{

}

void PlayerRelay::SetCbk(ts\_data\_input\_t ts\_data\_input, void\* user)

{

*printf\_s*("注册回调函数\n");

this->m\_ts\_data\_input = ts\_data\_input;

this->user = user;

}

void PlayerRelay::transport\_data()

{

*printf\_s*("PalyerRelay::transport\_data\n");

const char\* s = "afddg";

if (m\_ts\_data\_input)

{

m\_ts\_data\_input(user,s,5);

}

}

Player.h

#pragma once

#include "playerRelay.h"

#include <iostream>

class Player

{

public:

Player();

~Player();

public:

static void ts\_data\_input(void \*user,const char \*s,int length);//回调函数一般设置为static

void ts\_data\_input\_in(const char\* s,int length);//该函数处理PlayerRelay中的数据

protected:

private:

};

#include "player.h"

Player::Player()

{

PlayerRelay playerRelay;

//注册回调函数，将ts\_data\_input和当前类注册到类PlayerRekay中

playerRelay.SetCbk(ts\_data\_input, this);

//在该函数中将数据作为参数传入ts\_data\_input(...)中，

//最终在当前类中实现数据处理

playerRelay.transport\_data();

}

Player::~Player()

{

}

//回调函数

//传过来的参数user其实就是player

//PlayerRelay在通过ts\_data\_input传过来

void Player::ts\_data\_input(void\* user, const char\* s, int length)

{

*std*::*cout* << "回调函数：--->Player::ts\_data\_input\n" << *std*::*endl*;

Player\* pthis = (Player\*)user;

pthis->ts\_data\_input\_in(s, length);

}

void Player::ts\_data\_input\_in(const char\* s, int length)

{

*std*::*cout* << "处理数据Player::ts\_data\_input\_in\n" << *std*::*endl*;

*std*::*cout* << s << *std*::*endl*;

*std*::*cout* << length<< *std*::*endl*;

*std*::*cout* << "do with data!" << *std*::*endl*;

}

int main()

{

Player player;

*getchar*();

return 0;

}

这个小程序的目的是在Player类中调用PlayerRely类中的方法。在这里playerRely中的SetCbk()是回调函数，SetCbk里面输入的是Player中的一个函数，先进去SetCbk函数，之后将Player类中的ts\_data\_input的输入，从而赋值给PlayerRely的成员变量m\_ts\_data\_input，然后在playerRelay.transport\_data();

中就会使用到那个成员变量，进行的是ts\_data\_input，之后因为在ts\_data\_input中有pthis->ts\_data\_input\_in(s, length);所以才会到后面的void Player::ts\_data\_input\_in(const char\* s, int length)

的运算。

也就是说setCbk（）只是将Player类中的函数指针赋给了它而已，没有对void Player::ts\_data\_input(void\* user, const char\* s, int length)

进行计算。

这个的确是解耦了。没有那么大的耦合性。

1. **十一章，类**

**第一 类定义，对象和访问数据成员**

**第二 类成员函数**

**第三 类访问修饰符（public，private，protected）**

**第四 构造函数和析构函数**

**第五 c++拷贝构造函数**

**第六 友元函数和友元类**

**第七 c++内联函数（略）**

**第八 c++中的this指针**

**第九 智能指针**

**第十 c++中指向类的指针**

**第十一 c++类的静态成员**

**第十二 重载函数和运算符重载**

**第一 类定义，对象和访问数据成员**

类用于指定对象的形式，它包含了**数据表示法**和用于**处理数据的方法**。**类中的数据和方法称为类的成员**。**函数在一个类中被称为类的成员**。

**一 C++ 类定义**

定义一个类，本质上是定义一个数据类型的蓝图。这实际上并没有定义任何数据，**但它定义了类的名称意味着什么，也就是说，它定义了类的对象包括了什么，以及可以在这个对象上执行哪些操作。**

类定义是以关键字 class 开头，后跟类的名称。类的主体是包含在一对花括号中。类定义后必须跟着一个分号或一个声明列表。例如，我们使用关键字 class 定义 Box 数据类型，如下所示：**一般写数据成员时候加m\_**

class Box

{

**private:**

double m\_length; // 盒子的长度

double m\_breadth; // 盒子的宽度

double m\_height; // 盒子的高度

**protected:**

**public:**

};

**二 访问数据成员**

**（1）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。**

**（2）需要注意的是，私有的成员和受保护的成员不能使用直接成员访问运算符 (.) 来直接访问。我们将在后续的教程中学习如何访问私有成员和受保护的成员。**

**第二 类成员函数**

类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。

成员函数可以定义在A**类定义内部**，或者B**单独使用范围解析运算符 :: 来定义**。

A**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。**

**一般内联函数放在头文件中，并且代码量很少。inline 函数名（），立马就是定义。**

**而在类中，不需要inline 。**

**B 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：**

double Box::getVolume(void)

{

return length \* breadth \* height;

}

**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，如下所示：**

Box myBox; // 创建一个对象

myBox.getVolume(); // 调用该对象的成员函数

**其实就是public的内容，可以这样调用。**

**第三 类访问修饰符（public，private，protected）**

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。

**（一）公有（public）成员**

公有成员在程序中类的外部是可访问的。您可以不使用任何成员函数来设置和获取公有变量的值。

**（二）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。

默认情况下，类的所有成员都是私有的。

**（三）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，有两点都没有改变：**

**（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。**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 继承方式 | 基类的public成员 | 基类的protected成员 | 基类的private成员 | 继承引起的访问控制关系变化概括 |
| public继承 | 仍为public成员 | 仍为protected成员 | 不可见 | 基类的非私有成员在子类的访问属性不变 |
| protected继承 | 变为protected成员 | 变为protected成员 | 不可见 | 基类的非私有成员都为子类的保护成员 |
| private继承 | 变为private成员 | 变为private成员 | 不可见 | 基类中的非私有成员都称为子类的私有成员 |

**第四 构造函数和析构函数**

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。**

**一 默认构造函数**

**默认构造函数是未提供显式初始值时，用来创建对象的构造函数，如**

#include <iostream>

using namespace *std*;

class Line

{

public:

void setLength(double len);

double getLength(void);

**Line(); // 这是构造函数**

private:

double length;

};

// 成员函数定义，包括构造函数

**Line::Line(void)**

**{**

***cout* << "Object is being created" << *endl*;**

**}**

void Line::setLength(double len)

{

length = len;

}

double Line::getLength(void)

{

return length;

}

// 程序的主函数

int *main*()

{

**Line line;**

// 设置长度

line.setLength(6.0);

*cout* << "Length of line : " << line.getLength() << *endl*;

return 0;

}

**注意：在这里的构造函数是没有初始化的。**

**二 带参数的构造函数**

默认的构造函数没有任何参数，但如果需要，构造函数也可以带有参数。这样在创建对象时就会给对象赋初始值，如下面的例子所示：

#include "pch.h"

#include <iostream>

using namespace *std*;

class Line

{

public:

void setLength(double len);

double getLength(void);

**Line(double len); // 这是构造函数**

private:

double m\_length;

};

// 成员函数定义，包括构造函数

**Line::Line(double len)**

**{**

***cout* << "Object is being created, length = " << len << *endl*;**

**m\_length = len;**

**}**

void Line::setLength(double len)

{

m\_length = len;

}

double Line::getLength(void)

{

return m\_length;

}

// 程序的主函数

int *main*()

{

**Line line(10.0);//初始化参数**

// 获取默认设置的长度

*cout* << "Length of line : " << line.getLength() << *endl*;

// 再次设置长度

line.setLength(6.0);

*cout* << "Length of line : " << line.getLength() << *endl*;

return 0;

}

**注意：带参数的构造函数可以初始化。**

**使用初始化列表来初始化字段**

Line::Line(double len) : length(len)

{

*cout* << "Object is being created, length = " << len << *endl*;

}

**与下面的语法同一意义**

Line::Line(double len)

{

length = len;

*cout* << "Object is being created, length = " << len << *endl*;

}

**注意：**

**假设有一个类 C，具有多个字段 X、Y、Z 等需要进行初始化，同理地，您可以使用上面的语法，只需要在不同的字段使用逗号进行分隔，如下所示：**

*C*::*C*(double a, double b, double c) : *X*(a), *Y*(b), *Z*(c)

{

....

}

**三 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。

下面的实例有助于更好地理解析构函数的概念：

#include <iostream>

using namespace *std*;

class Line

{

public:

void setLength(double len);

double getLength(void);

Line(); // 这是构造函数声明

~Line(); // 这是析构函数声明

private:

double length;

};

// 成员函数定义，包括构造函数

Line::Line(void)

{

*cout* << "Object is being created" << *endl*;

}

Line::~Line(void)

{

*cout* << "Object is being deleted" << *endl*;

}

void Line::setLength(double len)

{

length = len;

}

double Line::getLength(void)

{

return length;

}

// 程序的主函数

int *main*()

{

Line line;

// 设置长度

line.setLength(6.0);

*cout* << "Length of line : " << line.getLength() << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

Object is being created

Length of line : 6

Object is being deleted

**当创建对象的时候会使用构造函数，当return 0时候会使用构造函数。**

**第五 c++拷贝构造函数**

拷贝构造函数是一种特殊的构造函数**，它在创建对象时**，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

**（1）通过使用另一个同类型的对象来初始化新创建的对象。**

**（2）复制对象把它作为参数传递给函数。**

**（3）复制对象，并从函数返回这个对象。**

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。拷贝构造函数的最常见形式如下：

**classname**(const **classname** &obj) {

// 构造函数的主体

}

**在这里，obj 是一个对象引用，该对象是用于初始化另一个对象的。**

**拷贝构造函数的调用时机**

**什么情况使用拷贝构造函数：**

**类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：**

**（1）一个对象以值传递的方式传入函数体**

**（2）一个对象以值传递的方式从函数返回**

**（3）一个对象需要通过另外一个对象进行初始化。**

#include "pch.h"

#include <iostream>

using namespace *std*;

class Line

{

public:

int getLength(void);

Line(int len); // 简单的构造函数

Line(const Line &obj); // 拷贝构造函数

~Line(); // 析构函数

private:

int \*ptr;

};

// 成员函数定义，包括构造函数

Line::Line(int len)

{

*cout* << "调用构造函数" << *endl*;

// 为指针分配内存

ptr = new int;

\*ptr = len;

}

Line::Line(const Line &obj)

{

*cout* << "调用拷贝构造函数并为指针 ptr 分配内存" << *endl*;

ptr = new int;

\*ptr = \*obj.ptr; // 拷贝值

}

Line::~Line(void)

{

*cout* << "释放内存" << *endl*;

delete ptr;

}

int Line::getLength(void)

{

return \*ptr;

}

void display(Line obj)

{

*cout* << "line 大小 : " << obj.getLength() << *endl*;

}

// 程序的主函数

int *main*()

{

Line line1(10);

Line line2 = line1; // 这里也调用了拷贝构造函数

display(line1);

display(line2);

return 0;

}

结果是

调用构造函数

调用拷贝构造函数并为指针 ptr 分配内存

**调用拷贝构造函数并为指针 ptr 分配内存**

**line 大小 : 10**

**释放内存**

**调用拷贝构造函数并为指针 ptr 分配内存**

**line 大小 : 10**

**释放内存**

释放内存

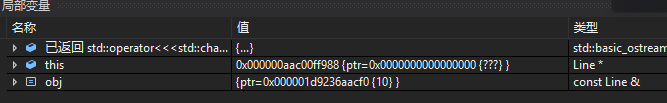
释放内存

**单步调试步骤**

先调用了构造函数Line::Line(int len)，初始化；

之后就到了Line line2 = line1;

先调用了拷贝构造函数Line::Line(const Line &obj)，这里有个this，后面会讨论



之后就到了display(line1);

调用拷贝构造函数Line::Line(const Line &obj)

再调用void display(Line obj);

之后析构函数Line::~Line(void)

之后再来一次

调用拷贝构造函数Line::Line(const Line &obj)

再调用void display(Line obj);

之后析构函数Line::~Line(void)

然后析构函数

C++ primer p406 ：拷贝构造函数是一种特殊的构造函数，具**有单个形参**，该形参（常用const修饰）是对该类类型的引用。当定义一个新对象并用一个同类型的对象对它进行初始化时，**将显示使用拷贝构造函数。当该类型的对象传递给函数或从函数返回该类型的对象时，将隐式调用拷贝构造函数。**

C++支持两种初始化形式：

拷贝初始化 int a = 5; 和直接初始化 int a(5); 对于其他类型没有什么区别，对于类类型直接初始化直接调用实参匹配的构造函数，拷贝初始化总是调用拷贝构造函数，也就是说：

A x(2);　　//直接初始化，调用构造函数

A y = x;　　//拷贝初始化，调用拷贝构造函数

**第六 友元函数和友元类**

**一 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：

class Box

{

double width;

public:

double length;

**friend void printWidth(Box box);**

void setWidth(double wid);

};

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;**

#include "pch.h"

#include <iostream>

using namespace *std*;

**class Box**

{

double width;

public:

**friend void printWidth(Box box);**

void setWidth(double wid);

};

// 成员函数定义

void Box::setWidth(double wid)

{

width = wid;

}

// 请注意：printWidth() 不是任何类的成员函数

**void printWidth(Box box)**

{

/\* 因为 printWidth() 是 Box 的友元**，它可以直接访问该类的任何成员** \*/

*cout* << "Width of box : " << box.width << *endl*;

}

// 程序的主函数

int *main*()

{

Box box ;//没有设置构造函数，所以使用的是默认构造函数

// 使用成员函数设置宽度

box.setWidth(10.0);

// 使用友元函数输出宽度

printWidth(box);//看到这个表达，不是box.printWidth(),因为setWidth()返回的是Box类型。

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

Width of box : 10

**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：**

（1）要访问非static成员时，需要对象做参数；

（2）要访问static成员或全局变量时，则不需要对象做参数；

（3）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

**二 友元类**

#include "pch.h"

#include <iostream>

using namespace *std*;

class Box

{

double width;

public:

**friend void printWidth(Box box);**

**friend class BigBox;**

void setWidth(double wid);

};

**class BigBox**

{

public:

**void Print(int width, Box &box)**

{

**// BigBox是Box的友元类，它可以直接访问Box类的任何成员**

**box.setWidth(width);**

*cout* << "Width of box : " << box.width << *endl*;

}

};

// 成员函数定义

void Box::setWidth(double wid)

{

width = wid;

}

// 请注意：printWidth() 不是任何类的成员函数

void printWidth(Box box)

{

/\* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 \*/

*cout* << "Width of box : " << box.width << *endl*;

}

// 程序的主函数

int *main*()

{

Box box;

BigBox big;

// 使用成员函数设置宽度

box.setWidth(10.0);

// 使用友元函数输出宽度

printWidth(box);

// 使用友元类中的方法设置宽度

big.Print(20, box);

*getchar*();

return 0;

}

**第七 c++内联函数（略）**

**第八 c++中的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。

下面的实例有助于更好地理解 this 指针的概念：

#include "pch.h"

#include <iostream>

using namespace *std*;

class Box

{

public:

// 构造函数定义

Box(double l = 2.0, double b = 2.0, double h = 2.0)

{

*cout* << "Constructor called." << *endl*;

length = l;

breadth = b;

height = h;

}

double Volume()

{

return length \* breadth \* height;

}

int compare(Box box)

{

return this->Volume() > box.Volume();

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int *main*(void)

{

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

if (Box1.compare(Box2))

{

*cout* << "Box2 is smaller than Box1" << *endl*;

}

else

{

*cout* << "Box2 is equal to or larger than Box1" << *endl*;

}

return 0;

}

C++ Primer Page 258

**引入 this：**

当我们调用成员函数时，实际上是替某个对象调用它。成员函数通过一个名为 this 的额外隐式参数来访问调用它的那个对象，当我们调用一个成员函数时，用请求该函数的对象地址初始化 this。

例如，如果调用 total.isbn()则编译器负责把 total 的地址传递给 isbn 的隐式形参 this，可以等价地认为编译器将该调用重写成了以下形式：

**//伪代码，用于说明调用成员函数的实际执行过程Sales\_data::isbn(&total)**

其中，调用 Sales\_data 的 isbn 成员时传入了 total 的地址。

在成员函数内部，**我们可以直接使用调用该函数的对象的成员，而无须通过成员访问运算符来做到这一点，因为 this 所指的正是这个对象。**任何对类成员的直接访问都被看作是对 this 的隐式引用**，也就是说，当 isbn 使用 bookNo 时，它隐式地使用 this 指向的成员，就像我们书写了 this->bookNo 一样。**

对于我们来说，this 形参是隐式定义的。实际上，任何自定义名为 this 的参数或变量的行为都是非法的。我们可以在成员函数体内部使用 this，因此尽管没有必要，我们还是能把 isbn 定义成如下形式：

std::string isbn() const { return this->bookNo; }

因为 this 的目的总是指向“这个”对象，所以 this 是一个常量指针（参见2.4.2节，第56页），我们不允许改变 this 中保存的地址。

#include <iostream>using namespace std;

class Box{

public:

Box(){;}

~Box(){;}

Box\* get\_address() //得到this的地址

{

return this;

}};

int main(){

Box box1;

Box box2;

// Box\* 定义指针p接受对象box的get\_address()成员函数的返回值，并打印

Box\* p = box1.get\_address();

cout << p << endl;

p = box2.get\_address();

cout << p << endl;

return 0;}

**this 指针的类型可理解为 Box\*。**

**此时得到两个地址分别为 box1 和 box2 对象的地址。**

**第九 智能指针**

**一 什么是智能指针**

C++程序设计中使用堆内存是非常频繁的操作，堆内存的申请和释放都由程序员自己管理。程序员自己管理堆内存可以提高了程序的效率，但是整体来说堆内存的管理是麻烦的，C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

**理解智能指针需要从下面三个层次：**

**（1）**从较浅的层面看，智能指针是利用了一种叫做RAII（资源获取即初始化）的技术对普通的指针进行封装，这使得智能指针实质是一个对象，行为表现的却像一个指针。

**（2）**智能指针的作用是防止忘记调用delete释放内存和程序异常的进入catch块忘记释放内存。另外指针的释放时机也是非常有考究的，多次释放同一个指针会造成程序崩溃，这些都可以通过智能指针来解决。

**（3）**智能指针还有一个作用是把值语义转换成引用语义。C++和Java有一处最大的区别在于语义不同，在Java里面下列代码：

**二 智能指针的使用**

**（一） shared\_ptr的使用**

shared\_ptr多个指针指向相同的对象。shared\_ptr使用引用计数，每一个shared\_ptr的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除所指向的堆内存。shared\_ptr内部的引用计数是线程安全的，但是对象的读取需要加锁。

1. **初始化**

**智能指针是个模板类，**可以指定类型，传入指针通过构造函数初始化。也可以使用make\_shared函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如std::shared\_ptr<int> p4 = new int(1);的写法是错误的

1. **拷贝和赋值**

拷贝使得对象的引用计数增加1，赋值使得原对象引用计数减1，当计数为0时，自动释放内存。后来指向的对象引用计数加1，指向后来的对象。

**（3）get函数获取原始指针**

**（4）注意不要用一个原始指针初始化多个shared\_ptr，否则会造成二次释放同一内存**

**（5）**注意避免循环引用，shared\_ptr的一个最大的陷阱是循环引用，循环，循环引用会导致堆内存无法正确释放，导致内存泄漏。循环引用在weak\_ptr中介绍。

#include <iostream>

#include <memory>

int main()

{

int a = 10;

*std*::*shared\_ptr*<int> ptra = *std*::*make\_shared*<int>(a);

*std*::*shared\_ptr*<int> ptra2(ptra); //copy

*std*::*cout* << ptra.*use\_count*() << *std*::*endl*;

int b = 20;

int\* pb = &a;

//std::shared\_ptr<int> ptrb = pb; //error

*std*::*shared\_ptr*<int> ptrb = *std*::*make\_shared*<int>(b);

ptra2 = ptrb; //assign

pb = ptrb.*get*(); //获取原始指针

*std*::*cout* << ptra.*use\_count*() << *std*::*endl*;

*std*::*cout* << ptrb.*use\_count*() << *std*::*endl*;

}

1. **unique\_ptr的使用**

unique\_ptr“唯一”拥有其所指对象，同一时刻只能有一个unique\_ptr指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。相比与原始指针unique\_ptr用于其RAII的特性，使得在出现异常的情况下，动态资源能得到释放。unique\_ptr指针本身的生命周期：从unique\_ptr指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他操作)。unique\_ptr指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过reset方法重新指定、通过release方法释放所有权、通过移动语义转移所有权。

#include <iostream>

#include <memory>

int main() {

{

std::unique\_ptr<int> uptr(new int(10)); //绑定动态对象

//std::unique\_ptr<int> uptr2 = uptr; //不能賦值

//std::unique\_ptr<int> uptr2(uptr); //不能拷貝

std::unique\_ptr<int> uptr2 = std::move(uptr); //轉換所有權

uptr2.release(); //释放所有权 }

//超過uptr的作用域，內存釋放

}

**（三）weak\_ptr的使用**

weak\_ptr是为了配合shared\_ptr而引入的一种智能指针，因为它不具有普通指针的行为，没有重载operator\*和->,它的最大作用在于协助shared\_ptr工作，像旁观者那样观测资源的使用情况。weak\_ptr可以从一个shared\_ptr或者另一个weak\_ptr对象构造，获得资源的观测权。但weak\_ptr没有共享资源，它的构造不会引起指针引用计数的增加。使用weak\_ptr的成员函数use\_count()可以观测资源的引用计数，另一个成员函数expired()的功能等价于use\_count()==0,但更快，表示被观测的资源(也就是shared\_ptr的管理的资源)已经不复存在。weak\_ptr可以使用一个非常重要的成员函数lock()从被观测的shared\_ptr获得一个可用的shared\_ptr对象， 从而操作资源。但当expired()==true的时候，lock()函数将返回一个存储空指针的shared\_ptr。

#include <iostream>

#include <memory>

int main() {

{

std::shared\_ptr<int> sh\_ptr = std::make\_shared<int>(10);

std::cout << sh\_ptr.use\_count() << std::endl;

std::weak\_ptr<int> wp(sh\_ptr);

std::cout << wp.use\_count() << std::endl;

if(!wp.expired()){

std::shared\_ptr<int> sh\_ptr2 = wp.lock(); //get another shared\_ptr

\*sh\_ptr = 100;

std::cout << wp.use\_count() << std::endl;

}

}

//delete memory

}

**（四）循环引用**

考虑一个简单的对象建模——家长与子女：a Parent has a Child, a Child knowshis/her Parent。在Java 里边很好写，不用担心内存泄漏，也不用担心空悬指针，只要正确初始化myChild 和myParent，那么Java 程序员就不用担心出现访问错误。一个handle 是否有效，只需要判断其是否non null。

public class Parent  
{  
　　private Child myChild;  
}  
public class Child  
{  
　　private Parent myParent;  
}  
 在C++ 里边就要为资源管理费一番脑筋。如果使用原始指针作为成员，Child和Parent由谁释放？那么如何保证指针的有效性？如何防止出现空悬指针？这些问题是C++面向对象编程麻烦的问题，现在可以借助smart pointer把对象语义（pointer）转变为值（value）语义，shared\_ptr轻松解决生命周期的问题，不必担心空悬指针。但是这个模型存在循环引用的问题，注意其中一个指针应该为weak\_ptr。

**原始指针的做法，容易出错**

**#include <iostream>**

**#include <memory>**

**class Child;**

**class Parent;**

**class Parent {**

**private:**

**Child\* myChild;**

**public:**

**void setChild(Child\* ch) {**

**this->myChild = ch;**

**}**

**void doSomething() {**

**if (this->myChild) {**

**}**

**}**

**~Parent() {**

**delete myChild;**

**}**

**};**

**class Child {**

**private:**

**Parent\* myParent;**

**public:**

**void setPartent(Parent\* p) {**

**this->myParent = p;**

**}**

**void doSomething() {**

**if (this->myParent) {**

**}**

**}**

**~Child() {**

**delete myParent;**

**}**

**};**

**int main() {**

**{**

**Parent\* p = new Parent;**

**Child\* c = new Child;**

**p->setChild(c);**

**c->setPartent(p);**

**delete c; //only delete one**

**}**

**return 0;**

**}**

循环引用内存泄露的问题

#include <iostream>

#include <memory>

class Child;class Parent;

class Parent {private:

std::shared\_ptr<Child> ChildPtr;public:

void setChild(std::shared\_ptr<Child> child) {

this->ChildPtr = child;

}

void doSomething() {

if (this->ChildPtr.use\_count()) {

}

}

~Parent() {

}

};

class Child {private:

std::shared\_ptr<Parent> ParentPtr;public:

void setPartent(std::shared\_ptr<Parent> parent) {

this->ParentPtr = parent;

}

void doSomething() {

if (this->ParentPtr.use\_count()) {

}

}

~Child() {

}

};

int main() {

std::weak\_ptr<Parent> wpp;

std::weak\_ptr<Child> wpc;

{

std::shared\_ptr<Parent> p(new Parent);

std::shared\_ptr<Child> c(new Child);

p->setChild(c);

c->setPartent(p);

wpp = p;

wpc = c;

std::cout << p.use\_count() << std::endl; // 2

std::cout << c.use\_count() << std::endl; // 2 }

std::cout << wpp.use\_count() << std::endl; // 1

std::cout << wpc.use\_count() << std::endl; // 1

return 0;

}

正确做法

#include <iostream>

#include <memory>

class Child;class Parent;

class Parent {private:

//std::shared\_ptr<Child> ChildPtr;

std::weak\_ptr<Child> ChildPtr;public:

void setChild(std::shared\_ptr<Child> child) {

this->ChildPtr = child;

}

void doSomething() {

//new shared\_ptr

if (this->ChildPtr.lock()) {

}

}

~Parent() {

}

};

class Child {private:

std::shared\_ptr<Parent> ParentPtr;public:

void setPartent(std::shared\_ptr<Parent> parent) {

this->ParentPtr = parent;

}

void doSomething() {

if (this->ParentPtr.use\_count()) {

}

}

~Child() {

}

};

int main() {

std::weak\_ptr<Parent> wpp;

std::weak\_ptr<Child> wpc;

{

std::shared\_ptr<Parent> p(new Parent);

std::shared\_ptr<Child> c(new Child);

p->setChild(c);

c->setPartent(p);

wpp = p;

wpc = c;

std::cout << p.use\_count() << std::endl; // 2

std::cout << c.use\_count() << std::endl; // 1 }

std::cout << wpp.use\_count() << std::endl; // 0

std::cout << wpc.use\_count() << std::endl; // 0

return 0;

}

**三智能指针的设计和实现**

下面是一个简单智能指针的demo。智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）。智能指针就是模拟指针动作的类。所有的智能指针都会重载 -> 和 \* 操作符。智能指针还有许多其他功能，比较有用的是自动销毁。这主要是利用栈对象的有限作用域以及临时对象（有限作用域实现）析构函数释放内存。

#include <iostream>

#include <memory>

template <typename T>

class SmartPointer {

private:

T\* \_ptr;

*size\_t*\* \_count;

public:

SmartPointer(T\* ptr = nullptr) :

\_ptr(ptr) {

if (\_ptr) {

\_count = new *size\_t*(1);

}

else {

\_count = new *size\_t*(0);

}

}

SmartPointer(const SmartPointer& ptr) {

if (this != &ptr) {

this->\_ptr = ptr.\_ptr;

this->\_count = ptr.\_count;

(\*this->\_count)++;

}

}

SmartPointer& operator=(const SmartPointer& ptr) {

if (this->\_ptr == ptr.\_ptr) {

return \*this;

}

if (this->\_ptr) {

(\*this->\_count)--;

if (this->\_count == 0) {

delete this->\_ptr;

delete this->\_count;

}

}

this->\_ptr = ptr.\_ptr;

this->\_count = ptr.\_count;

(\*this->\_count)++;

return \*this;

}

T& operator\*() {

*assert*(this->\_ptr == nullptr);

return \*(this->\_ptr);

}

T\* operator->() {

*assert*(this->\_ptr == nullptr);

return this->\_ptr;

}

~SmartPointer() {

(\*this->\_count)--;

if (\*this->\_count == 0) {

delete this->\_ptr;

delete this->\_count;

}

}

*size\_t* use\_count() {

return \*this->\_count;

}

};

int main() {

{

SmartPointer<int> sp(new int(10));

SmartPointer<int> sp2(sp);

SmartPointer<int> sp3(new int(20));

sp2 = sp3;

*std*::*cout* << sp.use\_count() << *std*::*endl*;

*std*::*cout* << sp3.use\_count() << *std*::*endl*;

}

//delete operator

}

完全是大佬！！！有点不明白！！模板类！！

**第十c++中指向类的指针**

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。**

下面的实例有助于更好地理解指向类的指针的概念：

#include <iostream>

using namespace std;

class Box{

public:

// 构造函数定义

Box(double l=2.0, double b=2.0, double h=2.0)

{

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

}

double Volume()

{

return length \* breadth \* height;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box};

int main(void){

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

Box \*ptrBox; // Declare pointer to a class.

// 保存第一个对象的地址

ptrBox = &Box1;

// 现在尝试使用成员访问运算符来访问成员

cout << "Volume of Box1: " << ptrBox->Volume() << endl;

// 保存第二个对象的地址

ptrBox = &Box2;

// 现在尝试使用成员访问运算符来访问成员

cout << "Volume of Box2: " << ptrBox->Volume() << endl;

return 0;}

**结果**

**Constructor called.**

**Constructor called.**

**Volume of Box1: 5.94**

**Volume of Box2: 102**

**第十一 c++类的静态成员**

**一静态成员变量**

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，如下面的实例所示。**

**下面的实例有助于更好地理解静态成员数据的概念：**

#include "pch.h"

#include <iostream>

using namespace *std*;

class Box

{

public:

static int objectCount;

// 构造函数定义

Box(double l = 2.0, double b = 2.0, double h = 2.0)

{

*cout* << "Constructor called." << *endl*;

length = l;

breadth = b;

height = h;

// 每次创建对象时增加 1

objectCount++;

}

double Volume()

{

return length \* breadth \* height;

}

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

// 初始化类 Box 的静态成员

**int Box::objectCount = 0;**

int *main*(void)

{

Box Box1(3.3, 1.2, 1.5); // 声明 box1

Box Box2(8.5, 6.0, 2.0); // 声明 box2

// 输出对象的总数

*cout* << "Total objects: " << **Box::objectCount** << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Constructor called.**

**Constructor called.**

**Total objects: 2**

**二 静态成员函数**

**（一）静态成员函数的作用**

**（1）**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。**

**（2）**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。

**（3）**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。

**（二）静态成员函数与普通成员函数的区别：**

（1）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）。

（2）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。

下面的实例有助于更好地理解静态成员函数的概念：

#include "pch.h"

#include <iostream>

using namespace *std*;

class Box

{

public:

**static int objectCount;**

// 构造函数定义

Box(double l = 2.0, double b = 2.0, double h = 2.0)

{

*cout* << "Constructor called." << *endl*;

length = l;

breadth = b;

height = h;

// 每次创建对象时增加 1

objectCount++;

}

double Volume()

{

return length \* breadth \* height;

}

**static int getCount()**

**{**

**return objectCount;**

**}**

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

// 初始化类 Box 的静态成员

**int Box::objectCount = 0;**

int *main*(void)

{

// 在创建对象之前输出对象的总数

*cout* << "Inital Stage Count: " << **Box::getCount()** << *endl*;

Box Box1(3.3, 1.2, 1.5); // 声明 box1

Box Box2(8.5, 6.0, 2.0); // 声明 box2

// 在创建对象之后输出对象的总数

*cout* << "Final Stage Count: " << **Box::getCount()** << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Inital Stage Count: 0**

**Constructor called.**

**Constructor called.**

**Final Stage Count: 2**

**第十二 重载函数和运算符重载**

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。

**一 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。

下面的实例中，同名函数 print() 被用于输出不同的数据类型：

#include <iostream>

using namespace *std*;

class printData

{

public:

void print(int i) {

*cout* << "整数为: " << i << *endl*;

}

void print(double f) {

*cout* << "浮点数为: " << f << *endl*;

}

void print(char c[]) {

*cout* << "字符串为: " << c << *endl*;

}

};

int main(void)

{

printData pd;

// 输出整数

pd.print(5);

// 输出浮点数

pd.print(500.263);

// 输出字符串

char c[] = "Hello C++";

pd.print(c);

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

整数为 : 5

浮点数为 : 500.263

字符串为 : Hello *C*++

**二 运算符重载**

您可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

Box operator+(const Box&);

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

Box operator+(const Box&, const Box&);

下面的实例使用成员函数演示了运算符重载的概念。在这里，对象作为参数进行传递，对象的属性使用 this 运算符进行访问，如下所示：

#include <iostream>

using namespace *std*;

class Box

{

public:

double getVolume(void)

{

return length \* breadth \* height;

}

void setLength(double len)

{

length = len;

}

void setBreadth(double bre)

{

breadth = bre;

}

void setHeight(double hei)

{

height = hei;

}

// 重载 + 运算符，用于把两个 Box 对象相加

Box operator+(const Box& b)

{

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

// 程序的主函数

int main()

{

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

Box Box3; // 声明 Box3，类型为 Box

double volume = 0.0; // 把体积存储在该变量中

// Box1 详述

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// Box2 详述

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// Box1 的体积

volume = Box1.getVolume();

*cout* << "Volume of Box1 : " << volume << *endl*;

// Box2 的体积

volume = Box2.getVolume();

*cout* << "Volume of Box2 : " << volume << *endl*;

// 把两个对象相加，得到 Box3

Box3 = Box1 + Box2;

// Box3 的体积

volume = Box3.getVolume();

*cout* << "Volume of Box3 : " << volume << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Volume of Box1 : 210**

**Volume of Box2 : 1560**

**Volume of Box3 : 5400**

1. **十四章，继承**

**第一 继承**

**第二 多态**

**第三 数据抽象**

**第四 接口（抽象类）**

**第一 继承**

面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行效率的效果。

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为基类，新建的类称为派生类。

继承代表了 is a 关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

**一 基类 & 派生类**

一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。定义一个派生类，我们使用一个类派生列表来指定基类。类派生列表以一个或多个基类命名，形式如下：

class derived-class: access-specifier base-class

其中，访问修饰符 access-specifier 是 public、protected 或 private 其中的一个，base-class 是之前定义过的某个类的名称。如果未使用访问修饰符 access-specifier，则默认为 private。

假设有一个基类 Shape，Rectangle 是它的派生类，如下所示：

#include "pch.h"

#include <iostream>

using namespace *std*;

// 基类

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

**protected:**

**int width;**

**int height;**

};

// 派生类

**class Rectangle : public Shape**

{

**public:**

**int getArea()**

**{**

**return (width \* height);**

**}**

};

int *main*(void)

{

**Rectangle Rect;**

**Rect.setWidth(5);**

**Rect.setHeight(7);**

// 输出对象的面积

*cout* << "Total area: " << **Rect.getArea()** << *endl*;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Total area: 35

**二 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：**

**（1）基类的构造函数、析构函数和拷贝构造函数。**

**（2）基类的重载运算符。**

**（3）基类的友元函数。**

**三 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：

**（1）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。

**（2）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。

**（3）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。

**四 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。现在让我们一起看看下面的实例：

#include "pch.h"

#include <iostream>

using namespace *std*;

// 基类 Shape

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// 基类 PaintCost

class PaintCost

{

public:

int getCost(int area)

{

return area \* 70;

}

};

// 派生类

class Rectangle : public Shape, public PaintCost

{

public:

int getArea()

{

return (width \* height);

}

};

int *main*(void)

{

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// 输出对象的面积

*cout* << "Total area: " << Rect.getArea() << *endl*;

// 输出总花费

*cout* << "Total paint cost: $" << Rect.getCost(area) << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Total area: 35**

**Total paint cost: $2450**

**第二 多态**

多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

下面的实例中，基类 Shape 被派生为两个类，如下所示：

#include "pch.h"

#include <iostream>

using namespace *std*;

class Shape

{

protected:

int width, height;

public:

Shape(int a = 0, int b = 0)

{

width = a;

height = b;

}

**int area()**

**{**

***cout* << "Parent class area :" << *endl*;**

**return 0;**

**}**

};

class Rectangle : public Shape

{

public:

Rectangle(int a = 0, int b = 0) :Shape(a, b) { }

**int area()**

{

*cout* << "Rectangle class area :" << width \* height<< *endl*;

return (width \* height);

}

};

class Triangle : public Shape

{

public:

Triangle(int a = 0, int b = 0) :Shape(a, b) { }

**int area()**

{

*cout* << "Triangle class area :"<< width \* height / 2<< *endl*;

return (width \* height / 2);

}

};

// 程序的主函数

int *main*()

{

Shape \*shape;

Rectangle rec(10, 7);

Triangle tri(10, 5);

// 存储矩形的地址

**shape = &rec;**

// 调用矩形的求面积函数 area

shape->area();

// 存储三角形的地址

**shape = &tri;**

// 调用三角形的求面积函数 area

shape->area();

return 0;

}

**得到的结果是**

**Parent class area**

**Parent class area**

用到的是基类！！

#include "pch.h"

#include <iostream>

using namespace *std*;

class Shape

{

protected:

int width, height;

public:

Shape(int a = 0, int b = 0)

{

width = a;

height = b;

}

**virtual int area()**

**{**

***cout* << "Parent class area :" << *endl*;**

**return 0;**

**}**

};

class Rectangle : public Shape

{

public:

Rectangle(int a = 0, int b = 0) :Shape(a, b) { }

**int area()**

{

*cout* << "Rectangle class area :" << width \* height<< *endl*;

return (width \* height);

}

};

class Triangle : public Shape

{

public:

Triangle(int a = 0, int b = 0) :Shape(a, b) { }

**int area()**

{

*cout* << "Triangle class area :"<< width \* height / 2<< *endl*;

return (width \* height / 2);

}

};

// 程序的主函数

int *main*()

{

Shape A;

Shape \*shape = &A;

Shape->area();

Rectangle rec(10, 7);

Triangle tri(10, 5);

// 存储矩形的地址

**shape = &rec;**

// 调用矩形的求面积函数 area

**shape->area();**

// 存储三角形的地址

**shape = &tri;**

// 调用三角形的求面积函数 area

**shape->area();**

return 0;

}

**得到的结果**

**Parent class area:**

**Rectangle class area：70**

**Triangle class area:25**

对比两者的区别就是在基类的**virtual int area(),加了一个virtual**

此时，**编译器看的是指针的内容，而不是它的类型。**因此，由于 tri 和 rec 类的对象的地址存储在 \*shape 中，所以会调用各自的 area() 函数。

正如您所看到的，每个子类都有一个函数 area() 的独立实现。这就是多态的一般使用方式。有了多态，您可以有多个不同的类，都带有同一个名称但具有不同实现的函数，函数的参数甚至可以是相同的。

**（一） 虚函数**

虚函数 是在基类中使用关键字 virtual 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要**静态链接到该函**数,要动态链接。

**我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或后期绑定。**

**（二）纯虚函数**

您可能想要在基类中定义虚函数，**以便在派生类中重新定义该函数更好地适用于对象，但是您在类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。**

我们可以把基类中的虚函数 area() 改写如下：

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0)

{

width = a;

height = b;

}

// pure virtual function

**virtual int area() = 0;**

};

**virtual int area() = 0;**

**这个=0 告诉编译器，函数没有主体，上面的虚函数是纯虚函数。**

**第三 数据抽象**

数据抽象是指，只向外界提供关键信息，并隐藏其后台的实现细节，即只表现必要的信息而不呈现细节。**数据抽象是一种依赖于接口和实现分离的编程（设计）技术。**

让我们举一个现实生活中的真实例子，比如一台电视机，您可以打开和关闭、切换频道、调整音量、添加外部组件（如喇叭、录像机、DVD 播放器），但是您不知道它的内部实现细节，也就是说，您并不知道它是如何通过缆线接收信号，如何转换信号，并最终显示在屏幕上。

因此，我们可以说电视把它的内部实现和外部接口分离开了，您无需知道它的内部实现原理，直接通过它的外部接口（比如电源按钮、遥控器、声量控制器）就可以操控电视。

现在，让我们言归正传，就 C++ 编程而言，C++ 类为数据抽象提供了可能。它们向外界提供了大量用于操作对象数据的公共方法，也就是说，外界实际上并不清楚类的内部实现。

例如，您的程序可以调用 sort() 函数，而不需要知道函数中排序数据所用到的算法。实际上，函数排序的底层实现会因库的版本不同而有所差异，只要接口不变，函数调用就可以照常工作。

在 C++ 中，我们使用类来定义我们自己的抽象数据类型（ADT）。您可以使用类 iostream 的 cout 对象来输出数据到标准输出，如下所示：

#include <iostream>

using namespace *std*;

int main()

{

*cout* << "Hello C++" << *endl*;

return 0;

}

在这里我们不是需要理解 cout 是如何在用户的屏幕上显示文本。您只需要知道公共接口即可，cout 的底层实现可以自由改变。

**一 访问标签强制抽象**

在 C++ 中，我们使用访问标签来定义**类的抽象接口**。一个类可以包含零个或多个访问标签：

**（1）**使用公共标签定义的成员都可以访问该程序的所有部分。一个类型的数据抽象视图是由它的公共成员来定义的。

**（2）**使用私有标签定义的成员无法访问到使用类的代码。私有部分对使用类型的代码隐藏了实现细节。

访问标签出现的频率没有限制。每个访问标签指定了紧随其后的成员定义的访问级别。指定的访问级别会一直有效，直到遇到下一个访问标签或者遇到类主体的关闭右括号为止。

**二 数据抽象的好处**

数据抽象有两个重要的优势：

**（1）类的内部受到保护，不会因无意的用户级错误导致对象状态受损。**

**（2）类实现可能随着时间的推移而发生变化，以便应对不断变化的需求，或者应对那些要求不改变用户级代码的错误报告。**

如果只在类的私有部分定义数据成员，编写该类的作者就可以随意更改数据。如果实现发生改变，则只需要检查类的代码，看看这个改变会导致哪些影响。如果数据是公有的，则任何直接访问旧表示形式的数据成员的函数都可能受到影响。

**三 数据抽象的实例**

C++ 程序中，任何带有公有和私有成员的类都可以作为数据抽象的实例。请看下面的实例：

#include "pch.h"

#include <iostream>

using namespace *std*;

class Adder {

public:

// 构造函数

Adder(int i = 0)

{

total = i;

}

// 对外的接口

void addNum(int number)

{

total += number;

}

// 对外的接口

int getTotal()

{

return total;

};

private:

// 对外隐藏的数据

int total;

};

int *main*()

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

*cout* << "Total " << a.getTotal() << *endl*;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

**Total 60**

**上面的类把数字相加，并返回总和。公有成员 addNum 和 getTotal 是对外的接口，用户需要知道它们以便使用类。私有成员 total 是用户不需要了解的，但又是类能正常工作所必需的。**

**第四 接口（抽象类）**

接口描述了类的行为和功能，而不需要完成类的特定实现。

C++ 接口是使用抽象类来实现的，抽象类与数据抽象互不混淆，数据抽象是一个把实现细节与相关的数据分离开的概念。

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "= 0" 来指定的，如下所示：

class Box

{

public:

// 纯虚函数

virtual double getVolume() = 0;

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

设计**抽象类**（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为**接口**使用。如果试图实例化一个抽象类的对象，会导致编译错误。

因此，如果一个 ABC 的子类需要被实例化，则必须实现每个虚函数，这也意味着 C++ 支持使用 ABC 声明接口。如果没有在派生类中重写纯虚函数，就尝试实例化该类的对象，会导致编译错误。

**可用于实例化对象的类被称为具体类**。

抽象类的实例

请看下面的实例，基类 Shape 提供了一个接口 getArea()，在两个派生类 Rectangle 和 Triangle 中分别实现了 getArea()：

#include "pch.h"

#include <iostream>

using namespace *std*;

// 基类

class Shape

{

public:

// 提供接口框架的纯虚函数

**virtual int getArea() = 0;**

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// 派生类

class Rectangle : public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

class Triangle : public Shape

{

public:

int getArea()

{

return (width \* height) / 2;

}

};

int *main*(void)

{

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

// 输出对象的面积

*cout* << "Total Rectangle area: " << Rect.getArea() << *endl*;

Tri.setWidth(5);

Tri.setHeight(7);

// 输出对象的面积

*cout* << "Total Triangle area: " << Tri.getArea() << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Total Rectangle area: 35**

**Total Triangle area: 17**

从上面的实例中，我们可以看到一个抽象类是如何定义一个接口 getArea()，两个派生类是如何通过不同的计算面积的算法来实现这个相同的函数。

**对比一下第2个点的多态区别！！！**

1. **十六章 异常，模板，和其他**

**第一 异常**

**第二 模板**

**第三 其他（在stl和数据结构里面）**

**第一 异常**

异常是程序在执行期间产生的问题。C++ 异常是指在程序运行时发生的特殊情况，比如尝试除以零的操作。异常提供了一种转移程序控制权的方式。C++ 异常处理涉及到三个**关键字：try、catch、throw。**

**（1）throw: 当问题出现时，程序会抛出一个异常。这是通过使用 throw 关键字来完成的。**

**（2）catch: 在您想要处理问题的地方，通过异常处理程序捕获异常。catch 关键字用于捕获异常。**

**（3）try: try 块中的代码标识将被激活的特定异常。它后面通常跟着一个或多个 catch 块。**

如果有一个块抛出一个异常，捕获异常的方法会使用 try 和 catch 关键字。try 块中放置可能抛出异常的代码，try 块中的代码被称为保护代码。使用 try/catch 语句的语法如下所示：

try

{

// 保护代码

}

catch (ExceptionName e1)

{

// catch 块

}

catch (ExceptionName e2)

{

// catch 块

}

catch (ExceptionName eN)

{

// catch 块

}

如果 try 块在不同的情境下会抛出不同的异常，这个时候可以尝试罗列多个 catch 语句，用于捕获不同类型的异常。

**一 抛出异常**

您可以使用 throw 语句在代码块中的任何地方抛出异常。throw 语句的操作数可以是任意的表达式，表达式的结果的类型决定了抛出的异常的类型。

以下是尝试除以零时抛出异常的实例：

double division(int a, int b)

{

if (b == 0)

{

throw "Division by zero condition!";

}

return (a / b);

}

**二 捕获异常**

catch 块跟在 try 块后面，用于捕获异常。您可以指定想要捕捉的异常类型，这是由 catch 关键字后的括号内的异常声明决定的。

try

{

// 保护代码

}

catch (ExceptionName e)

{

// 处理 ExceptionName 异常的代码

}

上面的代码会捕获一个类型为 ExceptionName 的异常。如果您想让 catch 块能够处理 try 块抛出的任何类型的异常，则必须在异常声明的括号内使用省略号 ...，如下所示：

try

{

// 保护代码

}

catch (...)

{

// 能处理任何异常的代码

}

下面是一个实例，抛出一个除以零的异常，并在 catch 块中捕获该异常。

#include "pch.h"

#include <iostream>

using namespace *std*;

double division(int a, int b)

{

if (b == 0)

{

throw "Division by zero condition!";

}

return (a / b);

}

int *main*()

{

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

*cout* << z << *endl*;

}

catch (const char\* msg) {

*cerr* << msg << *endl*;

}

return 0;

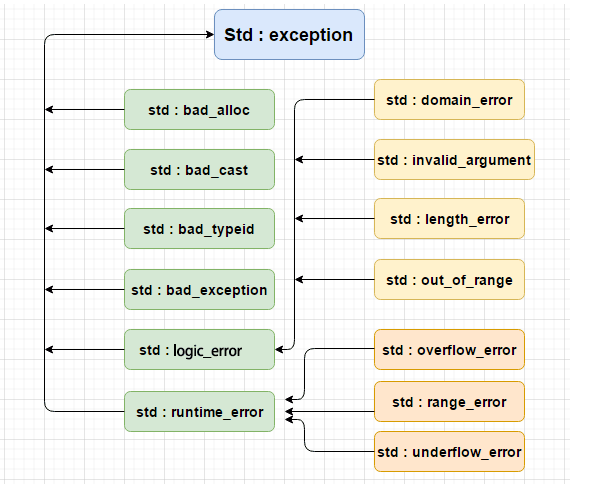
}

由于我们抛出了一个类型为 const char\* 的异常，因此，当捕获该异常时，我们必须在 catch 块中使用 const char\*。当上面的代码被编译和执行时，它会产生下列结果：

**Division by zero condition!**

**三 C++ 标准的异常**

C++ 提供了一系列标准的异常，定义在 <exception> 中，我们可以在程序中使用这些标准的异常。它们是以父子类层次结构组织起来的，如下所示：



下表是对上面层次结构中出现的每个异常的说明：

|  |  |
| --- | --- |
| **异常** | **描述** |
| **std::exception** | 该异常是所有标准 C++ 异常的父类。 |
| std::bad\_alloc | 该异常可以通过 **new** 抛出。 |
| std::bad\_cast | 该异常可以通过 **dynamic\_cast** 抛出。 |
| std::bad\_exception | 这在处理 C++ 程序中无法预期的异常时非常有用。 |
| std::bad\_typeid | 该异常可以通过 **typeid** 抛出。 |
| **std::logic\_error** | 理论上可以通过读取代码来检测到的异常。 |
| std::domain\_error | 当使用了一个无效的数学域时，会抛出该异常。 |
| std::invalid\_argument | 当使用了无效的参数时，会抛出该异常。 |
| std::length\_error | 当创建了太长的 std::string 时，会抛出该异常。 |
| std::out\_of\_range | 该异常可以通过方法抛出，例如 std::vector 和 std::bitset<>::operator[]()。 |
| **std::runtime\_error** | 理论上不可以通过读取代码来检测到的异常。 |
| std::overflow\_error | 当发生数学上溢时，会抛出该异常。 |
| std::range\_error | 当尝试存储超出范围的值时，会抛出该异常。 |
| std::underflow\_error | 当发生数学下溢时，会抛出该异常。 |

**二 定义新的异常**

您可以通过继承和重载 exception 类来定义新的异常。下面的实例演示了如何使用 std::exception 类来实现自己的异常：

#include "pch.h"

#include <iostream>

#include <exception>

using namespace *std*;

struct MyException : public *exception*

{

const char \* what() const throw ()

{

return "C++ Exception";

}

};

int *main*()

{

try

{

throw MyException();

}

catch (MyException& e)

{

*std*::*cout* << "MyException caught" << *std*::*endl*;

*std*::*cout* << e.what() << *std*::*endl*;

}

catch (*std*::*exception*& e)

{

//其他的错误

}

}

这将产生以下结果：

**MyException caught**

**C++ Exception**

**第二 模板**

**一函数模板**

模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码。

模板是创建泛型类或函数的蓝图或公式。库容器，比如迭代器和算法，都是泛型编程的例子，它们都使用了模板的概念。

每个容器都有一个单一的定义，比如 向量，我们可以定义许多不同类型的向量，比如 vector <int> 或 vector <string>。

您可以使用模板来定义函数和类，接下来让我们一起来看看如何使用。

**函数模板的格式：**

**template <class 形参名，class 形参名，......> 返回类型 函数名(参数列表)**

**{**

**函数体**

**}**

　　其中template和class是关键字**，class可以用typename 关见字代替，**在这里typename 和class没区别，<>括号中的参数叫模板形参，模板形参和函数形参很相像，模板形参不能为空。一但声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，即可以在该函数中使用内置类型的地方都可以使用模板形参名。模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际的模板实参类型就称他实例化了函数模板的一个实例。比如swap的模板函数形式为

**template <class T> void swap(T& a, T& b){}，**

**当调用这样的模板函数时类型T就会被被调用时的类型所代替**，比如swap(a,b)其中a和b是int 型，这时模板函数swap中的形参T就会被int 所代替，模板函数就变为swap(int &a, int &b)。而当swap(c,d)其中c和d是double类型时，模板函数会被替换为swap(double &a, double &b)，这样就实现了函数的实现与类型无关的代码。

**注意：对于函数模板而言不存在 h(int,int) 这样的调用，不能在函数调用的参数中指定模板形参的类型，对函数模板的调用应使用实参推演来进行，即只能进行 h(2,3) 这样的调用，或者int a, b; h(a,b)。**

#include <iostream>

#include <string>

using namespace *std*;

**template <typename T>**

**inline T const& Max(T const& a, T const& b)**

**{**

**return a < b ? b : a;**

**}**

int *main*()

{

**int i** = 39;

int j = 20;

***cout* << "Max(i, j): " << Max(i, j) << *endl*;**

**double f1 =** 13.5;

double f2 = 20.7;

*cout* << "Max(f1, f2): " << Max(f1, f2) << *endl*;

***string* s1** = "Hello";

*string* s2 = "World";

*cout* << "Max(s1, s2): " << Max(s1, s2) << *endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Max(i, j): 39**

**Max(f1, f2): 20.7**

**Max(s1, s2): World**

**二 类模板**

**正如我们定义函数模板一样，我们也可以定义类模板。泛型类声明的一般形式如下所示：**

**template <class type> class class-name {...}**

在这里，type 是占位符类型名称，可以在类被实例化的时候进行指定。您可以使用一个逗号分隔的列表来定义多个泛型数据类型。

下面的实例定义了类 Stack<>，并实现了泛型方法来对元素进行入栈出栈操作：

#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>

using namespace *std*;

template <class T>

class Stack {

private:

*vector*<T> elems; // 元素

public:

void push(T const&); // 入栈

void pop(); // 出栈

T top() const; // 返回栈顶元素

bool empty() const { // 如果为空则返回真。

return elems.*empty*();

}

};

template <class T>

void Stack<T>::push(T const& elem)

{

// 追加传入元素的副本

elems.*push\_back*(elem);

}

template <class T>

void Stack<T>::pop()

{

if (elems.*empty*()) {

throw *out\_of\_range*("Stack<>::pop(): empty stack");

}

// 删除最后一个元素

elems.*pop\_back*();

}

template <class T>

T Stack<T>::top() const

{

if (elems.*empty*()) {

throw *out\_of\_range*("Stack<>::top(): empty stack");

}

// 返回最后一个元素的副本

return elems.*back*();

}

int *main*()

{

try {

Stack<int> intStack; // int 类型的栈

Stack<*string*> stringStack; // string 类型的栈

// 操作 int 类型的栈

intStack.push(7);

*cout* << intStack.top() << *endl*;

// 操作 string 类型的栈

stringStack.push("hello");

*cout* << stringStack.top() << *std*::*endl*;

stringStack.pop();

stringStack.pop();

}

catch (*exception* const& ex) {

*cerr* << "Exception: " << ex.*what*() << *endl*;

return -1;

}

}

当上面的代码被编译和执行时，它会产生下列结果：

7

Hello

Exception: Stack<>::pop(): empty stack

**第三 其他（在stl和数据结构里面）**

**第十七章输入输出，文件（基本不用一般是使用Qt/MFC）**

**第一 基本的输入输出**

**第二 文件和流**

**第一 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。

**一 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&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。 |

**（一）标准输出流（cout）**

预定义的对象 cout 是 iostream 类的一个实例。cout 对象"连接"到标准输出设备，通常是显示屏。cout 是与流插入运算符 << 结合使用的，如下所示

#include <iostream>

using namespace *std*;

int main()

{

char str[] = "Hello C++";

*cout* << "Value of str is : " << str << *endl*;

}

C++ 编译器根据要输出变量的数据类型，选择合适的流插入运算符来显示值。<< 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。流插入运算符 << 在一个语句中可以多次使用，如上面实例中所示，endl 用于在行末添加一个换行符。

**（二）标准输入流（cin）**

预定义的对象 cin 是 iostream 类的一个实例。cin 对象附属到标准输入设备，通常是键盘。cin 是与流提取运算符 >> 结合使用的，如下所示：

#include <iostream>

using namespace *std*;

int main()

{

char name[50];

*cout* << "请输入您的名称： ";

*cin* >> name;

*cout* << "您的名称是： " << name << *endl*;

}

**（三）标准错误流（cerr）**

预定义的对象 cerr 是 iostream 类的一个实例。cerr 对象附属到标准错误设备，通常也是显示屏，但是 cerr 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

cerr 也是与流插入运算符 << 结合使用的，如下所示：

#include <iostream>

using namespace *std*;

int main()

{

char str[] = "Unable to read....";

*cerr* << "Error message : " << str << *endl*;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Error message : Unable to read....**

**（四）标准日志流（clog）**

预定义的对象 clog 是 iostream 类的一个实例。clog 对象附属到标准错误设备，通常也是显示屏，但是 clog 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲在，直到缓冲填满或者缓冲区刷新时才会输出。

clog 也是与流插入运算符 << 结合使用的，如下所示：

#include <iostream>

using namespace *std*;

int main()

{

char str[] = "Unable to read....";

*clog* << "Error message : " << str << *endl*;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Error message : Unable to read....**

**通过这些小实例，我们无法区分 cout、cerr 和 clog 的差异，但在编写和执行大型程序时，它们之间的差异就变得非常明显。所以良好的编程实践告诉我们，使用 cerr 流来显示错误消息，而其他的日志消息则使用 clog 流来输出。**

**第二 文件和流**

到目前为止，我们已经使用了 iostream 标准库，它提供了 cin 和 cout 方法分别用于从标准输入读取流和向标准输出写入流。

本教程介绍如何从文件读取流和向文件写入流。这就需要用到 C++ 中另一个标准库 fstream，它定义了三个新的数据类型：

|  |  |
| --- | --- |
| **数据类型** | **描述** |
| ofstream | 该数据类型表示输出文件流，用于创建文件并向文件写入信息。 |
| ifstream | 该数据类型表示输入文件流，用于从文件读取信息。 |
| fstream | 该数据类型通常表示文件流，且同时具有 ofstream 和 ifstream 两种功能，这意味着它可以创建文件，向文件写入信息，从文件读取信息。 |

**要在 C++ 中进行文件处理，必须在 C++ 源代码文件中包含头文件<iostream> 和 <fstream>。**

**一 打开文件**

在从文件读取信息或者向文件写入信息之前，必须先打开文件。ofstream 和 fstream 对象都可以用来打开文件进行写操作，如果只需要打开文件进行读操作，则使用 ifstream 对象。

下面是 open() 函数的标准语法，open() 函数是 fstream、ifstream 和 ofstream 对象的一个成员。

**void open(const char \*filename, ios::openmode mode);**

在这里，open() 成员函数的第一参数指定要打开的文件的名称和位置，第二个参数定义文件被打开的模式。

|  |  |
| --- | --- |
| **模式标志** | **描述** |
| ios::app | 追加模式。所有写入都追加到文件末尾。 |
| ios::ate | 文件打开后定位到文件末尾。 |
| ios::in | 打开文件用于读取。 |
| ios::out | 打开文件用于写入。 |
| ios::trunc | 如果该文件已经存在，其内容将在打开文件之前被截断，即把文件长度设为 0。 |

您可以把以上两种或两种以上的模式结合使用。例如，如果您想要以写入模式打开文件，并希望截断文件，以防文件已存在，那么您可以使用下面的语法：

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );

类似地，您如果想要打开一个文件用于读写，可以使用下面的语法：

ifstream afile;

afile.open("file.dat", ios::out | ios::in );

**二 关闭文件**

当 C++ 程序终止时，它会自动关闭刷新所有流，释放所有分配的内存，并关闭所有打开的文件。但程序员应该养成一个好习惯，在程序终止前关闭所有打开的文件。

下面是 close() 函数的标准语法，close() 函数是 fstream、ifstream 和 ofstream 对象的一个成员。

**void close();**

**三 写入文件**

在 C++ 编程中，我们使用流插入运算符（ << ）向文件写入信息，就像使用该运算符输出信息到屏幕上一样。唯一不同的是，在这里您使用的是 ofstream 或 fstream 对象，而不是 cout 对象。

**四 读取文件**

在 C++ 编程中，我们使用流提取运算符（ >> ）从文件读取信息，就像使用该运算符从键盘输入信息一样。唯一不同的是，在这里您使用的是 ifstream 或 fstream 对象，而不是 cin 对象。

**五 读取 & 写入实例**

下面的 C++ 程序以读写模式打开一个文件。在向文件 afile.dat 写入用户输入的信息之后，程序从文件读取信息，并将其输出到屏幕上：

#include "pch.h"

#include <fstream>

#include <iostream>

using namespace *std*;

int *main*()

{

char data[100];

// 以写模式打开文件

*ofstream* outfile;

outfile.*open*("afile.dat");

*cout* << "Writing to the file" << *endl*;

*cout* << "Enter your name: ";

*cin*.*getline*(data, 100);

// 向文件写入用户输入的数据

outfile << data << *endl*;

*cout* << "Enter your age: ";

*cin* >> data;

*cin*.*ignore*();

// 再次向文件写入用户输入的数据

outfile << data << *endl*;

// 关闭打开的文件

outfile.*close*();

// 以读模式打开文件

*ifstream* infile;

infile.*open*("afile.dat");

*cout* << "Reading from the file" << *endl*;

infile >> data;

// 在屏幕上写入数据

*cout* << data << *endl*;

// 再次从文件读取数据，并显示它

infile >> data;

*cout* << data << *endl*;

// 关闭打开的文件

infile.*close*();

return 0;

}

**Writing to the file**

**Enter your name: Zara**

**Enter your age: 9**

**Reading from the file**

**Zara**

**9**

**在项目下有一个afile.dat文件。**

**六 文件位置指针**

**istream** 和**ostream**都提供了用于重新定位文件位置指针的成员函数。这些成员函数包括关于 **istream** 的**seekg（"seek get"**）和关于 **ostream** 的**seekp（"seek put"）**。

seekg 和 seekp 的参数通常是一个长整型。第二个参数可以用于指定查找方向。查找方向可以是 ios::beg（默认的，从流的开头开始定位），也可以是 ios::cur（从流的当前位置开始定位），也可以是 ios::end（从流的末尾开始定位）。

文件位置指针是一个整数值，指定了从文件的起始位置到指针所在位置的字节数。下面是关于定位 "get" 文件位置指针的实例：

// 定位到 fileObject 的第 n 个字节（假设是 ios::beg）

**fileObject.seekg( n );**

// 把文件的读指针从 fileObject 当前位置向后移 n 个字节

**fileObject.seekg( n, ios::cur );**

// 把文件的读指针从 fileObject 末尾往回移 n 个字节

**fileObject.seekg( n, ios::end );**

// 定位到 fileObject 的末尾

**fileObject.seekg( 0, ios::end );**

1. **十二章内存模型和名称空间**

**第一 命名空间**

**第二 内存模型**

**第三 几个关键词（static,extern,auto,const,new,delete，\_beginthreadex）**

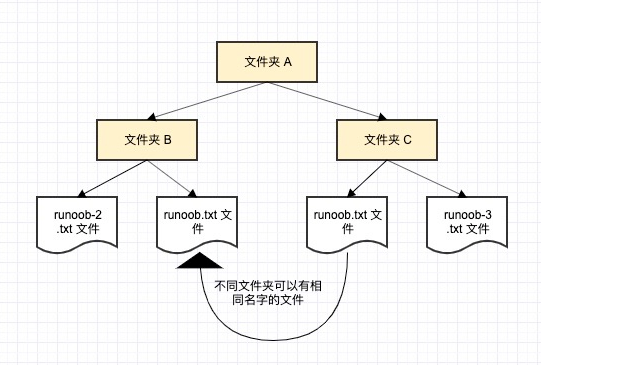
**第一 命名空间**

假设这样一种情况，当一个班上有两个名叫 Zara 的学生时，为了明确区分它们，我们在使用名字之外，不得不使用一些额外的信息，比如他们的家庭住址，或者他们父母的名字等等。

同样的情况也出现在 C++ 应用程序中。例如，您可能会写一个名为 xyz() 的函数，在另一个可用的库中也存在一个相同的函数 xyz()。这样，编译器就无法判断您所使用的是哪一个 xyz() 函数。

因此，引入了命名空间这个概念，专门用于解决上面的问题，它可作为附加信息来区分不同库中相同名称的函数、类、变量等。使用了命名空间即定义了上下文。本质上，命名空间就是定义了一个范围。

我们举一个计算机系统中的例子，一个文件夹(目录)中可以包含多个文件夹，每个文件夹中不能有相同的文件名，但不同文件夹中的文件可以重名。



**一 定义命名空间**

命名空间的定义使用关键字 namespace，后跟命名空间的名称，如下所示：

namespace namespace\_name {

// 代码声明

}

为了调用带有命名空间的函数或变量，需要在前面加上命名空间的名称，如下所示：

**name::code; // code 可以是变量或函数**

让我们来看看命名空间如何为变量或函数等实体定义范围：

#include "pch.h"

#include <iostream>

using namespace *std*;

// 第一个命名空间

namespace first\_space {

void func() {

*cout* << "Inside first\_space" << *endl*;

}

}

// 第二个命名空间

namespace second\_space {

void func() {

*cout* << "Inside second\_space" << *endl*;

}

}

int *main*()

{

// 调用第一个命名空间中的函数

first\_space::func();

// 调用第二个命名空间中的函数

second\_space::func();

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**Inside first\_space**

**Inside second\_space**

**二 using 指令**

您可以使用 using namespace 指令，这样在使用命名空间时就可以不用在前面加上命名空间的名称。这个指令会告诉编译器，后续的代码将使用指定的命名空间中的名称。

#include "pch.h"

#include <iostream>

using namespace *std*;

// 第一个命名空间

namespace first\_space {

void func() {

*cout* << "Inside first\_space" << *endl*;

}

}

// 第二个命名空间

namespace second\_space {

void func() {

*cout* << "Inside second\_space" << *endl*;

}

}

using namespace first\_space;

int *main*()

{

// 调用第一个命名空间中的函数

func();

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Inside first\_space

using 指令也可以用来指定命名空间中的特定项目。例如，如果您只打算使用 std 命名空间中的 cout 部分，您可以使用如下的语句：

using std::cout;

随后的代码中，在使用 cout 时就可以不用加上命名空间名称作为前缀，但是 std 命名空间中的其他项目仍然需要加上命名空间名称作为前缀，如下所示：

#include "pch.h"

#include <iostream>

using *std*::*cout*;

int *main*()

{

*cout* << "std::endl is used with std!" << *std*::*endl*;

return 0;

}

**当上面的代码被编译和执行时，它会产生下列结果：**

**std::endl is used with std!**

**using 指令引入的名称遵循正常的范围规则。名称从使用 using 指令开始是可见的，直到该范围结束。此时，在范围以外定义的同名实体是隐藏的。**

**三 嵌套的命名空间**

命名空间可以嵌套，您可以在一个命名空间中定义另一个命名空间，如下所示：

**namespace namespace\_name1 { // 代码声明 namespace namespace\_name2 { // 代码声明 } }**

您可以通过使用 :: 运算符来访问嵌套的命名空间中的成员：

// 访问 namespace\_name2 中的成员 using namespace namespace\_name1::namespace\_name2;

// 访问 namespace:name1 中的成员

using namespace namespace\_name1;

在上面的语句中，如果使用的是 namespace\_name1，那么在该范围内 namespace\_name2 中的元素也是可用的，如下所示：

#include "pch.h"

#include <iostream>

using namespace *std*;

// 第一个命名空间

namespace first\_space {

void func() {

*cout* << "Inside first\_space" << *endl*;

}

// 第二个命名空间

namespace second\_space {

void func() {

*cout* << "Inside second\_space" << *endl*;

}

}

}

using namespace first\_space::second\_space;

int *main*()

{

// 调用第二个命名空间中的函数

func();

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Inside second\_space

**关于命名空间内变量和函数及全局变量的使用和作用域:**

#include "pch.h"

#include <iostream>

using namespace *std*;

namespace A

{

int a = 100;

namespace B //嵌套一个命名空间B

{

int a = 20;

}

}

int a = 200;//定义一个全局变量

int *main*(int argc, char \*argv[])

{

*cout* << "A::a =" << *A*::a << *endl*;

*cout* << "a =" << a << *endl*;

*cout* << "::a =" << ::a << *endl*;

int a = 30;

*cout* << "a =" << a << *endl*;

*cout* << "::a =" << ::a << *endl*;

return 0;

}

结果：

A::a =100

A::B::a =20

a =200 //全局变量a::a =200

a =30 //局部变量a::a =200

**即：全局变量 a 表达为 ::a，用于当有同名的局部变量时来区别两者。**

补充关于 using 的错误事例：

#include "pch.h"

#include <iostream>

using namespace *std*;

namespace A

{

int a = 100;

int fun()

{

*cout* << "a = " << a << *endl*;

}

namespace B //嵌套一个命名空间B

{

int a = 20;

int fun()

{

*cout* << "a = " << a << *endl*;

}

}

}

int *main*(int argc, char \*argv[])

{

*cout* << a << *endl*;

fun();

return 0;

}

**这样会出错：**会显示 a 变量和 fun 函数 “was not declared in this scope”，即找不到这个 a 和 fun 函数。

**解决办法：** 用 using 来告诉编译器用到的是哪个命名空间内的内容。在 main() 上面加**using namespace A; 或者 using namespace A::B;**。这样就可以使用其中的 a 和 fun()。但是不能同时使用，因为这样也会导致编译出错，编译器器不知道要去使用哪个 a 和 fun()。

**第二 内存模型**

**一 内存管理**

1. **内存分配方式**

**（1）栈：**在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。**在c++ primer plus中，说的是自动变量一般储存在栈中。自动变量就是局部变量。**

**（2）堆：**就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。这在c++ primer plus中是动态存储包括自由存储空间和堆的。

**（3）自由储存区：**就是那些由malloc等分配的内存块，他和堆是十分相似的，不过它是用free来结束自己的生命的。

**（4）全局/静态存储区：**全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区

**（5）常量存储区：**这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。

**总结：内存有以上5种方式存储，有以下三种方式分配**

**（1）从静态存储区分配：**内存在程序编译的时候已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

**（2）在栈上创建：**在执行函数时，函数内局部变量的存储单元可以在栈上创建，函数执行结束时，这些内存单元会自动被释放。 栈内存分配运算内置于处理器的指令集，效率高，但是分配的内存容量有限。

**（3）在堆上分配：亦称为动态内存分配。** 程序在运行的时候使用malloc或者new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。 动态内存的生命周期有程序员决定，使用非常灵活，但如果在堆上分配了空间，既有责任回收它，否则运行的程序会出现内存泄漏，频繁的分配和释放不同大小的堆空间将会产生内存碎片。

1. **栈和堆的区别**

**（1）管理方式不同：**对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生内存泄露（memory leak）。

**（2）空间大小不同：**栈的空间是有限的，在32位平台下，VC6下默认为1M，堆最大可以到4G。**（当然可以变）**

**（3）碎片问题：**对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了。

**（4）生长方向：**对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

**（5）分配方式：堆都是动态分配的，没有静态分配的堆。**栈有2种分配方式：**静态分配和动态分配。静态分配**是编译器完成的，比如局部变量的分配。**动态分配**由malloc函数进行分配，**但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。因为是在c中实现的。**

**（6）分配效率**：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

**总结：**堆和栈相比，由于大量new/delete的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候debug可是相当困难的：）

例子：

//main.cpp   
int a = 0; **全局初始化区**char \*p1; **全局未初始化区**main()   
{   
int b; **栈**   
char s[] = "abc"; **栈**   
char \*p2; **栈**   
char \*p3 = "123456"; **123456在常量区，p3在栈上。**   
static int c =0； **全局（静态）初始化区**p1 = (char \*)malloc(10);   
p2 = (char \*)malloc(20);   
**分配得来得10和20字节的区域就在堆区**。   
strcpy(p1, "123456"); **123456放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。**}

**二 内存泄露**

**三 内存回收**

**第三 几个关键词（static,extern,auto,const,new,delete，\_beginthreadex）**

**一 New,delete**

了解动态内存在 C++ 中是如何工作的是成为一名合格的 C++ 程序员必不可少的。C++ 程序中的内存分为两个部分：

**栈：在函数内部声明的所有变量都将占用栈内存。**

**堆：这是程序中未使用的内存，在程序运行时可用于动态分配内存。**

很多时候，您无法提前预知需要多少内存来存储某个定义变量中的特定信息，所需内存的大小需要在运行时才能确定。

在 C++ 中，您可以使用特殊的运算符为给定类型的变量在运行时分配堆内的内存，这会返回所分配的空间地址。这种运算符即 new 运算符。

如果您不再需要动态分配的内存空间，可以使用 delete 运算符，删除之前由 new 运算符分配的内存。

**new 和 delete 运算符**

下面是使用 new 运算符来为任意的数据类型动态分配内存的通用语法：

**new data-type;**

在这里，data-type 可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。让我们先来看下内置的数据类型。例如，我们可以定义一个指向 double 类型的指针，然后请求内存，该内存在执行时被分配。我们可以按照下面的语句使用 new 运算符来完成这点：

double\* pvalue = NULL; // 初始化为 null 的指针

pvalue = new double; // 为变量请求内存

如果自由存储区已被用完，可能无法成功分配内存。所以建议检查 new 运算符是否返回 NULL 指针，并采取以下适当的操作：

double\* pvalue = NULL;

if( !(pvalue = new double ))

{

cout << "Error: out of memory." <<endl;

exit(1);

}

**malloc() 函数在 C 语言中就出现了，在 C++ 中仍然存在，但建议尽量不要使用 malloc() 函数。new 与 malloc() 函数相比，其主要的优点是，new 不只是分配了内存，它还创建了对象。**

在任何时候，当您觉得某个已经动态分配内存的变量不再需要使用时，您可以使用 delete 操作符释放它所占用的内存，如下所示：

**delete pvalue; // 释放 pvalue 所指向的内存**

#include "pch.h"

#include <iostream>

using namespace *std*;

int *main*()

{

double\* pvalue = *NULL*; // 初始化为 null 的指针

pvalue = new double; // 为变量请求内存

\*pvalue = 29494.99; // 在分配的地址存储值

*cout* << "Value of pvalue : " << \*pvalue << *endl*;

delete pvalue; // 释放内存

return 0;

}

**结果**

**Value of pvalue : 29495**

**二 数组的动态内存分配**

假设我们要为一个字符数组（一个有 20 个字符的字符串）分配内存，我们可以使用上面实例中的语法来为数组动态地分配内存，如下所示：

**char\* pvalue = NULL; // 初始化为 null 的指针**

**pvalue = new char[20]; // 为变量请求内存**

要删除我们刚才创建的数组，语句如下：

**delete [] pvalue; // 删除 pvalue 所指向的数组**

下面是 new 操作符的通用语法，可以为多维数组分配内存，如下所示：

**（一）一维数组**

// 动态分配,数组长度为

m int \*array=new int [m]；

//释放内存

delete [] array;

**（二）二维数组**

int \*\*array

// 假定数组第一维长度为 m， 第二维长度为 n

// 动态分配空间

array = new int \*[m];

for( int i=0; i<m; i++ )

{

**array[i] = new int [n]** ;

}

//释放

for( int i=0; i<m; i++ )

{

**delete [] arrary[i];**

}

**delete [] array;**

#include "pch.h"

#include <iostream>

using namespace *std*;

int *main*()

{

int \*\*p;

int i, j; //p[4][8]

//开始分配4行8列的二维数据

**p = new int \*[4];**

**for (i = 0; i < 4; i++) {**

**p[i] = new int[8];**

**}**

for (i = 0; i < 4; i++) {

for (j = 0; j < 8; j++) {

p[i][j] = j \* i;

}

}

//打印数据

for (i = 0; i < 4; i++) {

for (j = 0; j < 8; j++)

{

if (j == 0) *cout* << *endl*;

*cout* << p[i][j] << "\t";

}

}

//开始释放申请的堆

for (i = 0; i < 4; i++) {

delete[] p[i];

}

delete[] p;

return 0;

}

**（三）三维数组**

int \*\*\*array;

// 假定数组第一维为 m， 第二维为 n， 第三维为h

// 动态分配空间

array = new int \*\*[*m*];

for (int i = 0; i < *m*; i++)

{

array[i] = new int \*[*n*];

for (int j = 0; j < *n*; j++)

{

array[i][j] = new int[*h*];

}

}

//释放

for (int i = 0; i < *m*; i++)

{

for (int j = 0; j < *n*; j++)

{

delete[] array[i][j];

}

delete[] array[i];

}

delete[] array;

**实例**

#include "pch.h"

#include <iostream>

using namespace *std*;

int *main*()

{

int i, j, k; // p[2][3][4]

int \*\*\*p;

p = new int \*\*[2];

for (i = 0; i < 2; i++)

{

p[i] = new int \*[3];

for (j = 0; j < 3; j++)

p[i][j] = new int[4];

}

//输出 p[i][j][k] 三维数据

for (i = 0; i < 2; i++)

{

for (j = 0; j < 3; j++)

{

for (k = 0; k < 4; k++)

{

p[i][j][k] = i + j + k;

*cout* << p[i][j][k] << " ";

}

*cout* << *endl*;

}

*cout* << *endl*;

}

// 释放内存

for (i = 0; i < 2; i++)

{

for (j = 0; j < 3; j++)

{

delete[] p[i][j];

}

}

for (i = 0; i < 2; i++)

{

delete[] p[i];

}

delete[] p;

return 0;

}

**对象的动态内存分配**

对象与简单的数据类型没有什么不同。例如，请看下面的代码，我们将使用一个对象数组来理清这一概念：

#include "pch.h"

#include <iostream>

using namespace *std*;

class Box

{

public:

Box() {

*cout* << "调用构造函数！" << *endl*;

}

~Box() {

*cout* << "调用析构函数！" << *endl*;

}

};

int *main*()

{

Box\* myBoxArray = new Box[4];

delete[] myBoxArray; // 删除数组

return 0;

}

如果要为一个包含四个 Box 对象的数组分配内存，构造函数将被调用 4 次，同样地，当删除这些对象时，析构函数也将被调用相同的次数（4次）。

当上面的代码被编译和执行时，它会产生下列结果：

调用构造函数！

调用构造函数！

调用构造函数！

调用构造函数！

**调用析构函数！**

**调用析构函数！**

**调用析构函数！**

**调用析构函数！**

**（四）delete 与 delete[] 区别：**

**（1）针对简单类型 使用 new 分配后的不管是数组还是非数组形式内存空间用两种方式均可 如：**

int \*a = new int[10]; delete a; delete [] a;

此种情况中的释放效果相同**，原因在于：**分配简单类型内存时，内存大小已经确定，系统可以记忆并且进行管理，在析构时，系统并不会调用析构函数， 它直接通过指针可以获取实际分配的内存空间，哪怕是一个数组内存空间(在分配过程中 系统会记录分配内存的大小等信息，此信息保存在结构体\_CrtMemBlockHeader中， 具体情况可参看VC安装目录下CRT\SRC\DBGDEL.cpp)

**（2）针对类Class，两种方式体现出具体差异**

当你通过下列方式分配一个类对象数组：

class A

{

private:

char \*m\_cBuffer;

int m\_nLen;

public:

A() { m\_cBuffer = new char[m\_nLen]; }

~A() { delete[] m\_cBuffer; }

};

A \*a = new A[10];

**// 仅释放了a指针指向的全部内存空间 但是只调用了a[0]对象的析构函数 剩下的从a[1]到a[9]这9个用户自行分配的m\_cBuffer对应内存空间将不能释放 从而造成内存泄漏**

**delete a;**

**// 调用使用类对象的析构函数释放用户自己分配内存空间并且 释放了a指针指向的全部内存空间**

**delete[] a;**

**所以总结下就是，如果ptr代表一个用new申请的内存返回的内存空间地址，即所谓的指针，那么：**

**1）delete ptr -- 代表用来释放内存，且只用来释放ptr指向的内存。**

**2）delete[] rg -- 用来释放rg指向的内存，！！还逐一调用数组中每个对象的 destructor！！**

**对于像 int/char/long/int\*/struct 等等简单数据类型，由于对象没有 destructor，所以用 delete 和 delete [] 是一样的！但是如果是C++ 对象数组就不同了！**

**三 Static**

**（一）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）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。**

**（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。**

**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。**

**（二）在 C/C++ 中static的作用**

**（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）。

**（三） 静态变量与普通变量**

**（1）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）；

**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。**

**（2）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。

**（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。

**四static 用法**

**（一）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>。**

**（二） 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();//是个错误的，非静态成员引用必须与特定对象相对。

Point::output();

}

**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();

}

**（三）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。**

**（四）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。

**（五）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。

**五 Extern**

1. **基本解释**

**extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。此外extern也可用来进行链接指定。**

      也就是说extern有两个作用，第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b);则告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的，C++的规则在翻译这个函数名时会把fun这个名字变得面目全非，可能是fun@aBc\_int\_int#%$也可能是别的，这要看编译器的"脾气"了(不同的编译器采用的方法不一样)，为什么这么做呢，因为C++支持函数的重载啊，在这里不去过多的论述这个问题，如果你有兴趣可以去网上搜索，相信你可以得到满意的解释!  
 **（ 二）当extern不与"C"在一起修饰变量或函数时**，如在头文件中: **extern int g\_Int**; **它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!**也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时**，它只要包含A模块的头文件即可,**在编译阶段，**模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。**

**（三）问题：extern 变量**在一个源文件里定义了一个数组：char a[6];  
　　在另外一个文件里用下列语句进行了声明：extern char \*a；  
　　请问，这样可以吗？   
　　**答案与分析：**  
　　（1）不可以，程序运行时会告诉你非法访问。原因在于，指向类型T的指针并不等价于类型T的数组。extern char \*a声明的是一个指针变量而不是字符数组，因此与实际的定义不同，从而造成运行时非法访问。应该将声明改为extern char a[ ]。  
　　**（2）例子分析如下，如果a[] = "abcd",则外部变量a=0x61626364 (abcd的ASCII码值)，\*a显然没有意义  
　　显然a指向的空间（0x61626364）没有意义，易出现非法内存访问。**　 （3）这提示我们，在使用extern时候要严格对应声明时的格式，在实际编程中，这样的错误屡见不鲜。  
　 （4）**extern用在变量声明中常常有这样一个作用，你在\*.cpp文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在\*.h中并用extern来声明。使用方法**

1. **在第一个源文件.cpp中声明，初始化，在另外的源文件中extern 它，就可以了**
2. **包含第一个的头文件，在头文件中extern int i，在cpp文件中i=10；在使用的cpp文件中包含第一个个头文件.h就可以使用它了。**

**（四）问题：当方面修改extern 函数原型**当函数提供方单方面修改函数原型时，如果使用方不知情继续沿用原来的extern申明，这样编译时编译器不会报错。但是在运行过程中，因为少了或者多了输入参数，往往会照成系统错误，这种情况应该如何解决？  
　　答案与分析：  
　　目前业界针对这种情况的处理没有一个很完美的方案，通常的做法是提供方在自己的xxx\_pub.h中提供对外部接口的声明，然后调用方include该头文件，从而省去extern这一步。以避免这种错误。  
　　宝剑有双锋，对extern的应用，不同的场合应该选择不同的做法。

**（五）问题：extern “C”**  
　　在C++环境下使用C函数的时候，常常会出现编译器无法找到obj模块中的C函数定义，从而导致链接失败的情况，应该如何解决这种情况呢？

　　答案与分析：  
　　C++语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而C语言则不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern “C”进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。  
　　下面是一个标准的写法：  
//在.h文件的头上  
#ifdef \_\_cplusplus  
#if \_\_cplusplus  
extern "C"{  
　#endif  
　#endif /\* \_\_cplusplus \*/   
　…  
　…  
　//.h文件结束的地方  
　#ifdef \_\_cplusplus  
　#if \_\_cplusplus  
}  
#endif  
#endif /\* \_\_cplusplus \*/

**（6）问题：extern 函数声明  
　　常常见extern放在函数的前面成为函数声明的一部分，那么，C语言的关键字extern在函数的声明中起什么作用？  
　　答案与分析：  
　　如果函数的声明中带有关键字extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用。即下述两个函数声明没有明显的区别：  
extern int f(); 和int f();  
　　当然，这样的用处还是有的，就是在程序中取代include “\*.h”来声明函数，在一些复杂的项目中，我比较习惯在所有的函数声明前添加extern修饰。关于这样做的原因和利弊可见下面的这个例子：“用extern修饰的全局变量”**

   (1) 在test1.h中有下列声明:  
    #ifndef TEST1H  
    #define TEST1H  
    extern char g\_str[]; // 声明全局变量g\_str  
    void fun1();  
    #endif  
    (2) 在test1.cpp中  
    #include "test1.h"  
        char g\_str[] = "123456"; // 定义全局变量g\_str  
        void fun1() { cout << g\_str << endl; }  
    (3) 以上是test1模块， 它的编译和连接都可以通过,如果我们还有test2模块也想使用g\_str,只需要在原文件中引用就可以了  
    #include "test1.h"

     void fun2()    { cout << g\_str << endl;    }  
**以上test1和test2可以同时编译连接通过，如果你感兴趣的话可以用ultraEdit打开test1.obj,你可以在里面找到"123456"这个字符串,但是你却不能在test2.obj里面找到，这是因为g\_str是整个工程的全局变量，在内存中只存在一份,test2.obj这个编译单元不需要再有一份了，不然会在连接时报告重复定义这个错误!  
    (4) 有些人喜欢把全局变量的声明和定义放在一起，这样可以防止忘记了定义，如把上面test1.h改为  
    extern char g\_str[] = "123456"; // 这个时候相当于没有extern  
    然后把test1.cpp中的g\_str的定义去掉,这个时候再编译连接test1和test2两个模块时，会报连接错误，这是因为你把全局变量g\_str的定义放在了头文件之后，test1.cpp这个模块包含了test1.h所以定义了一次g\_str,而test2.cpp也包含了test1.h所以再一次定义了g\_str,这个时候连接器在连接test1和test2时发现两个g\_str。如果你非要把g\_str的定义放在test1.h中的话，那么就把test2的代码中#include "test1.h"去掉 换成:  
    extern char g\_str[];  
    void fun2()   {  cout << g\_str << endl;   }  
   这个时候编译器就知道g\_str是引自于外部的一个编译模块了，不会在本模块中再重复定义一个出来，但是我想说这样做非常糟糕，因为你由于无法在test2.cpp中使用#include "test1.h",那么test1.h中声明的其他函数你也无法使用了，除非也用都用extern修饰，这样的话你光声明的函数就要一大串，而且头文件的作用就是要给外部提供接口使用的，所以 请记住， 只在头文件中做声明，真理总是这么简单。**

**（7）extern 和 static**

 (1) extern 表明该变量在别的地方已经定义过了,在这里要使用那个变量.  
 (2) static 表示静态的变量，分配内存的时候, 存储在静态区,不存储在栈上面.

    static 作用范围是内部连接的关系, 和extern有点相反.它和对象本身是分开存储的,extern也是分开存储的,但是extern可以被其他的对象用extern 引用,而static 不可以,只允许对象本身用它. **具体差别首先，static与extern是一对“水火不容”的家伙，也就是说extern和static不能同时修饰一个变量；其次，static修饰的全局变量声明与定义同时进行，也就是说当你在头文件中使用static声明了全局变量后，它也同时被定义了；最后，static修饰全局变量的作用域只能是本身的编译单元，也就是说它的“全局”只对本编译单元有效，其他编译单元则看不到它**,如:  
    (1) test1.h:  
    #ifndef TEST1H  
    #define TEST1H  
    static char g\_str[] = "123456";   
    void fun1();  
    #endif

    (2) test1.cpp:  
    #include "test1.h"  
    void fun1()  {   cout << g\_str << endl;  }  
    (3) test2.cpp  
    #include "test1.h"  
    void fun2()  {   cout << g\_str << endl;  }  
    以上两个编译单元可以连接成功, 当你打开test1.obj时，你可以在它里面找到字符串"123456",同时你也可以在test2.obj中找到它们，**它们之所以可以连接成功而没有报重复定义的错误是因为虽然它们有相同的内容，但是存储的物理地址并不一样，就像是两个不同变量赋了相同的值一样，而这两个变量分别作用于它们各自的编译单元。** 也许你比较较真，自己偷偷的跟踪调试上面的代码,结果你发现两个编译单元（test1,test2）的g\_str的内存地址相同，于是你下结论static修饰的变量也可以作用于其他模块，但是我要告诉你，那是你的编译器在欺骗你，大多数编译器都对代码都有优化功能，以达到生成的目标程序更节省内存，执行效率更高，当编译器在连接各个编译单元的时候，它会把相同内容的内存只拷贝一份，比如上面的"123456", 位于两个编译单元中的变量都是同样的内容，那么在连接的时候它在内存中就只会存在一份了，如果你把上面的代码改成下面的样子，你马上就可以拆穿编译器的谎言:  
    (1) test1.cpp:  
    #include "test1.h"  
    void fun1()  
    {  
        g\_str[0] = ''a'';  
        cout << g\_str << endl;  
    }

    (2) test2.cpp  
    #include "test1.h"  
    void fun2()  {  cout << g\_str << endl;  }  
    (3) void main()     {  
        fun1(); // a23456  
        fun2(); // 123456  
    }  
    这个时候你在跟踪代码时，就会发现两个编译单元中的g\_str地址并不相同，因为你在一处修改了它，所以编译器被强行的恢复内存的原貌，在内存中存在了两份拷贝给两个模块中的变量使用。**正是因为static有以上的特性，所以一般定义static全局变量时，都把它放在源文件中而不是头文件，这样就不会给其他模块造成不必要的信息污染，同样记住这个原则吧！**

**（8）extern 和const**

   C++中const修饰的全局常量据有跟static相同的特性，即它们只能作用于本编译模块中，但是const可以与extern连用来声明该常量可以作用于其他编译模块中, 如extern const char g\_str[];  
    然后在原文件中别忘了定义:     const char g\_str[] = "123456";

    所以当const单独使用时它就与static相同，而当与extern一起合作的时候，它的特性就跟extern的一样了！所以对const我没有什么可以过多的描述，我只是想提醒你，const char\* g\_str = "123456" 与 const char g\_str[] ="123465"是不同的， 前面那个const 修饰的是char \*而不是g\_str,它的g\_str并不是常量，它被看做是一个定义了的全局变量（可以被其他编译单元使用）， 所以如果你像让char\*g\_str遵守const的全局常量的规则，最好这么定义const char\* const g\_str="123456".

**六Auto的用法**

auto可以在声明变量的时候根据变量初始值的类型自动为此变量选择匹配的类型，类似的关键字还有decltype。举个例子：

**int a = 10;**

**auto au\_a = a;//自动类型推断，au\_a为int类型**

cout << typeid(au\_a).name() << endl;

typeid运算符可以输出变量的类型。程序的运行结果输出了

int

这种用法就类似于C#中的var关键字。auto的自动类型推断发生在编译期，所以使用auto并不会造成程序运行时效率的降低。而是否会造成编译期的时间消耗，我认为是不会的，在未使用auto时，编译器也需要得知右操作数的类型，再与左操作数的类型进行比较，检查是否可以发生相应的转化，是否需要进行隐式类型转换。

**（一）auto的用法**

上面举的这个例子很简单，在真正编程的时候也不建议这样来使用auto，直接写出变量的类型更加清晰易懂。下面列举auto关键字的正确用法。

用于代替冗长复杂、变量使用范围专一的变量声明。

想象一下在没有auto的时候，我们操作标准库时经常需要这样：

#include<string>#include<vector>int main(){

std::vector<std::string> vs;

for (std::vector<std::string>::iterator i = vs.begin(); i != vs.end(); i++)

{

//...

}

}

这样看代码写代码实在烦得很。有人可能会说为何不直接使用using namespace std，这样代码可以短一点。实际上这不是该建议的方法（C++Primer对此有相关叙述）。使用auto能简化代码：

#include<string>#include<vector>int main(){

std::vector<std::string> vs;

for (auto i = vs.begin(); i != vs.end(); i++)

{

//..

}

}

for循环中的i将在编译时自动推导其类型，而不用我们显式去定义那长长的一串。

**在定义模板函数时，用于声明依赖模板参数的变量类型。**

template <typename \_Tx,typename \_Ty>void Multiply(\_Tx x, \_Ty y){

auto v = x\*y;

std::cout << v;

}

若不使用auto变量来声明v，那这个函数就难定义啦，不到编译的时候，谁知道x\*y的真正类型是什么呢？

**模板函数依赖于模板参数的返回值**

template <typename \_Tx, typename \_Ty>auto multiply(\_Tx x, \_Ty y)->decltype(x\*y){

return x\*y;

}

当模板函数的返回值依赖于模板的参数时，我们依旧无法在编译代码前确定模板参数的类型，故也无从知道返回值的类型，这时我们可以使用auto。格式如上所示。  
decltype操作符用于查询表达式的数据类型，也是C++11标准引入的新的运算符，其目的也是解决泛型编程中有些类型由模板参数决定，而难以表示它的问题。  
 auto在这里的作用也称为返回值占位，它只是为函数返回值占了一个位置，真正的返回值是后面的decltype(\_Tx\*\_Ty)。为何要将返回值后置呢？如果没有后置，则函数声明时为：

decltype(x\*y)multiply(\_Tx x, \_Ty y)

而此时x,y还没声明呢，编译无法通过。

**（二）注意事项**

**（1）auto 变量必须在定义时初始化，这类似于const关键字。**

**（2）定义在一个auto序列的变量必须始终推导成同一类型。**例如：

auto a4 = 10, a5 = 20, a6 = 30;//正确auto b4 = 10, b5 = 20.0, b6 = 'a';//错误,没有推导为同一类型

使用auto关键字做类型自动推导时，依次施加一下规则：

**（3）如果初始化表达式是引用，则去除引用语义。**

int a = 10;int &b = a;

auto c = b;//c的类型为int而非int&（去除引用）auto &d = b;//此时c的类型才为int&

c = 100;//a =10;

d = 100;//a =100;

**（4）如果初始化表达式为const或volatile（或者两者兼有），则除去const/volatile语义。**

const int a1 = 10;auto b1= a1; //b1的类型为int而非const int（去除const）const auto c1 = a1;//此时c1的类型为const int

b1 = 100;//合法

c1 = 100;//非法

**（5）如果auto关键字带上&号，则不去除const语意。**

const int a2 = 10;auto &b2 = a2;//因为auto带上&，故不去除const，b2类型为const int

b2 = 10; //非法

这是因为如何去掉了const，则b2为a2的非const引用，通过b2可以改变a2的值，则显然是不合理的。

**（6）初始化表达式为数组时，auto关键字推导类型为指针。**

int a3[3] = { 1, 2, 3 };auto b3 = a3;cout << typeid(b3).name() << endl;

程序将输出

int \*

若表达式为数组且auto带上&，则推导类型为数组类型。

int a7[3] = { 1, 2, 3 };auto & b7 = a7;cout << typeid(b7).name() << endl;

程序输出

int [3]

函数或者模板参数不能被声明为auto

void func(auto a) //错误{//...

}

**时刻要注意auto并不是一个真正的类型。  
auto仅仅是一个占位符，它并不是一个真正的类型，不能使用一些以类型为操作数的操作符，如sizeof或者typeid。**

**cout << sizeof(auto) << endl;//错误cout << typeid(auto).name() << endl;//错误**

**七\_beiginthreadex()的用法**

**（一）需要的头文件支持**

**#include <process.h>         // for \_beginthread()**

**需要的设置:ProjectàSetting-->C/C++-->User run-time library 选择Debug Multithreaded 或者Multithreaded。即使用: MT或MTD。**

**源码如下：**

#include <stdio.h>

#include <string> // for STL string class

#include <windows.h> // for HANDLE

#include <process.h> // for \_beginthread()

using namespace *std*;

class ThreadX

{

private:

int loopStart;

int loopEnd;

int dispFrequency;

public:

*string* threadName;

ThreadX(int startValue, int endValue, int frequency)

{

loopStart = startValue;

loopEnd = endValue;

dispFrequency = frequency;

}

static unsigned \_\_stdcall ThreadStaticEntryPoint(void\* pThis)

{

ThreadX\* pthX = (ThreadX\*)pThis; // the tricky cast

pthX->ThreadEntryPoint(); // now call the true entry-point-function

return 1; // the thread exit code

}

void ThreadEntryPoint()

{

for (int i = loopStart; i <= loopEnd; ++i)

{

if (i % dispFrequency == 0)

{

*printf*("%s: i = %d\n", threadName.*c\_str*(), i);

}

}

*printf*("%s thread terminating\n", threadName.*c\_str*());

}

};

int main()

{

ThreadX\* o1 = new ThreadX(0, 1, 2000);

*HANDLE* hth1;

unsigned uiThread1ID;

hth1 = (*HANDLE*)*\_beginthreadex*(*NULL*, // security

0, // stack size

ThreadX::ThreadStaticEntryPoint,

o1, // arg list

*CREATE\_SUSPENDED*, // so we can later call ResumeThread()

&uiThread1ID);

if (hth1 == 0)

*printf*("Failed to create thread 1\n");

*DWORD* dwExitCode;

*GetExitCodeThread*(hth1, &dwExitCode); // should be STILL\_ACTIVE = 0x00000103 = 259

*printf*("initial thread 1 exit code = %u\n", dwExitCode);

o1->threadName = "t1";

ThreadX\* o2 = new ThreadX(-100000, 0, 2000);

*HANDLE* hth2;

unsigned uiThread2ID;

hth2 = (*HANDLE*)*\_beginthreadex*(*NULL*, // security

0, // stack size

ThreadX::ThreadStaticEntryPoint,

o2, // arg list

*CREATE\_SUSPENDED*, // so we can later call ResumeThread()

&uiThread2ID);

if (hth2 == 0)

*printf*("Failed to create thread 2\n");

*GetExitCodeThread*(hth2, &dwExitCode); // should be STILL\_ACTIVE = 0x00000103 = 259

*printf*("initial thread 2 exit code = %u\n", dwExitCode);

o2->threadName = "t2";

*ResumeThread*(hth1); // serves the purpose of Jaeschke's t1->Start()

*ResumeThread*(hth2);

*WaitForSingleObject*(hth1, *INFINITE*);

*WaitForSingleObject*(hth2, *INFINITE*);

*GetExitCodeThread*(hth1, &dwExitCode);

*printf*("thread 1 exited with code %u\n", dwExitCode);

*GetExitCodeThread*(hth2, &dwExitCode);

*printf*("thread 2 exited with code %u\n", dwExitCode);

*CloseHandle*(hth1);

*CloseHandle*(hth2);

delete o1;

o1 = *NULL*;

delete o2;

o2 = *NULL*;

*printf*("Primary thread terminating.\n");

return 0;

}

**（二）解释**

**（1）如果你正在编写C/C++代码，决不应该调用CreateThread。相反，应该使用VisualC++运行期库函数\_beginthreadex，退出也应该使用\_endthreadex。如果不使用Microsoft的VisualC++编译器，你的编译器供应商有它自己的CreateThread替代函数。不管这个替代函数是什么，你都必须使用。**

**（2）因为\_beginthreadex和\_endthreadex是CRT线程函数，所以必须注意编译选项runtimelibaray的选择，使用MT或MTD。[MultiThreaded , Debug MultiThreaded]。**

**（3）\_beginthreadex函数的参数列表与CreateThread函数的参数列表是相同的，但是参数名和类型并不完全相同。这是因为Microsoft的C/C++运行期库的开发小组认为，C/C++运行期函数不应该对Windows数据类型有任何依赖。\_beginthreadex函数也像CreateThread那样，返回新创建的线程的句柄。**

**下面是关于\_beginthreadex的一些要点：**

**1）每个线程均获得由C/C++运行期库的堆栈分配的自己的tiddata内存结构。（tiddata结构位于Mtdll.h文件中的VisualC++源代码中）。**

**2）传递给\_beginthreadex的线程函数的地址保存在tiddata内存块中。传递给该函数的参数也保存在该数据块中。**

**3）\_beginthreadex确实从内部调用CreateThread，因为这是操作系统了解如何创建新线程的唯一方法。**

**4）当调用CreatetThread时，它被告知通过调用\_threadstartex而不是pfnStartAddr来启动执行新线程。还有，传递给线程函数的参数是tiddata结构而不是pvParam的地址。**

**5）如果一切顺利，就会像CreateThread那样返回线程句柄。如果任何操作失败了，便返回NULL。**

**（4）\_endthreadex的一些要点：**

**C运行期库的\_getptd函数内部调用操作系统的TlsGetValue函数，该函数负责检索调用线程的tiddata内存块的地址。**

**然后该数据块被释放，而操作系统的ExitThread函数被调用，以便真正撤消该线程。当然，退出代码要正确地设置和传递。**

**（5）虽然也提供了简化版的的\_beginthread和\_endthread，但是可控制性太差，所以一般不使用。**

**（6）线程handle因为是内核对象，所以需要在最后closehandle。**

**（7）更多的API：**

**HANDLE GetCurrentProcess();**

**HANDLE GetCurrentThread();**

**DWORD GetCurrentProcessId();**

**DWORD GetCurrentThreadId()。**

**DWORD SetThreadIdealProcessor(HANDLE hThread,DWORDdwIdealProcessor);**

**BOOL SetThreadPriority(HANDLE hThread,int nPriority);**

**BOOL SetPriorityClass(GetCurrentProcess(),  IDLE\_PRIORITY\_CLASS);**

**BOOL GetThreadContext(HANDLE hThread,PCONTEXTpContext);**

**BOOL SwitchToThread();**

**（三）注意**

**（1）C++主线程的终止，同时也会终止所有主线程创建的子线程，不管子线程有没有执行完毕。所以上面的代码中如果不调用WaitForSingleObject，则2个子线程t1和t2可能并没有执行完毕或根本没有执行。**

**（2）如果某线程挂起，然后有调用WaitForSingleObject等待该线程，就会导致死锁。所以上面的代码如果不调用resumethread，则会死锁。**

**四、为什么用\_beginthreadex而不是CreateThread?**

**为什么要用C运行时库的\_beginthreadex代替操作系统的CreateThread来创建线程？**

**来源自自1999年7月MSJ杂志的《Win32 Q&A》栏目**

**你也许会说我一直用CreateThread来创建线程，一直都工作得好好的，为什么要用\_beginthreadex来代替CreateThread，下面让我来告诉你为什么。**

**回答一个问题可以有两种方式，一种是简单的，一种是复杂的。**

**如果你不愿意看下面的长篇大论，那我可以告诉你简单的答案：\_beginthreadex在内部调用了CreateThread，在调用之前\_beginthreadex做了很多的工作，从而使得它比CreateThread更安全。**

**转载一部分，自己总结了一部分。**

**如果是使用Qt或者MFC那么就使用相应的框架构封装成的线程函数！！！！！**