By:Sukuna(CSXJ1902)

C++程序设计(OOP 的程序设计[C++])复习 I

Author:Sukuna

目录

C++要注意继承与虚函数(多态的实现)&以及运算符重载&多继承虚基类的定义与用法

第一部分 procedures oriented 部分

第 2 章 类型,常量,变量

2.1 C++的单词

1、char16_t 表示双字节字符类型，支持 UTF-16。
2、char32_t 表示四字节字符类型，支持 UTF-32
3、nullptr 表示空指针。

2.2 预定义类型及值域和常量

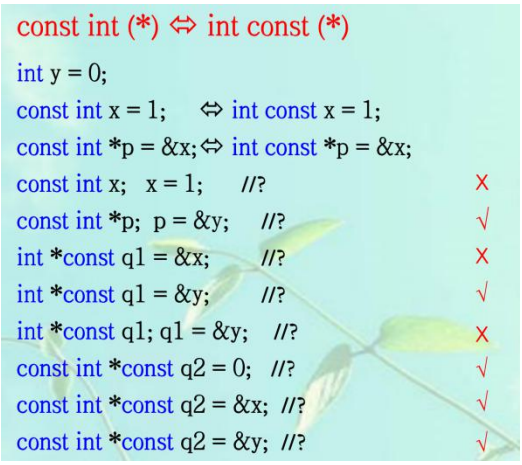
1、自动类型转换路径： char→unsigned char→ short→unsigned short→ int→unsigned int→long→unsigned long→float→double→long double。
2、数值零自动转换为布尔值 false，数值非零转换为布尔值 true。
3、强制类型转换的格式为： (类型表达式) 数值表达式
4、char、short、int、long 前可加 unsigned 表示无符号数。
5、long int 等价于 long；long long 占用八字节。
6、sizeof(short)≤sizeof(int)≤sizeof(long) ，sizeof(double)≤sizeof(long double)。(注意:小于等于)

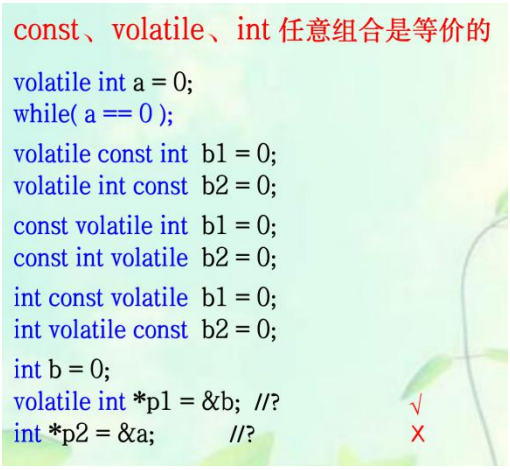
2.3 变量及其类型解析

1、变量说明：描述变量的类型及名称，但没有初始化。可以说明多次。
变量定义：描述变量的类型及名称，同时进行初始化。只能定义一次。
说明实例：extern int x; extern int x; //变量可以说明多次
定义实例：int x=3; extern int y=4; int z; //全局变量 z 的初始值为 0
2、**模块静态变量：使用 static 在函数外部定义的变量。可通过单目::访问。**
局部静态变量：使用 static 在函数内部定义的变量。

```
static int x, y; //模块静态变量 x、y 定义，默认初始值均为 0
int main( ){
    static int y;          //局部静态变量 y 定义，  初始值 y=0
    return  ::y+x+y;//分别访问模块静态变量 y,模块静态变量 x,局部静态变量
}
```

3、只读变量：使用 const 或 constexpr 说明或定义的变量，定义时必须同时初始化。**当前程序**只能读不能修改其值。
constexpr 变量必须用常量表达式初始化，编译将出现该变量的地方优化为常量。
4、易变变量：使用 volatile 说明或定义的变量，可以后初始化。当前程序没有修改其值，但是变量的值变了。不排出其它程序修改。(多线程)
const 实例：extern const int x; const int x=3; //定义必须显式初始化 x
volatile 例： extern const int y;; volatile int y; //可不显式初始化 y，全局 y=0
若 y=0，语句 if(y==2)是有意义的，因为易变变量 y 可以变为任何值。
作为类型修饰符，const 和 volatile 可以定义函数参数和返回值。
5、保留字 inline 用于定义函数外部变量或函数外部静态变量、类内部的静态数据成员。
inline 函数外部变量的作用域和 inline 函数外部静态变量一样,都是局限于当前代码文件的，相当于默认加了 static。
用 inline 定义的变量可以使用任意表达式初始化，但这样不能保证被优化。
6、VS2019 在 X86 编译模式下，使用 4 个字节表示地址
其他的 C 语言讲过
7、const 修饰的指针判断(作业一的第一题)





8、&定义的有址引用

传统左值有址引用变量必须由同类型的**传统左值表达式初始化。**

例如：在“int x=3; int &y=x;”中，x 为左值表达式，并且 x 的类型也为 int

说明：如果 x 为 char 类型，则 int&y=x 默认进行转换 int&y=(int)x，而转换后的结果(int)x 为 int 类型的右值，不符合用左值表达式初始化的要求。

传统右值有址引用变量要用的传统右值表达式初始化。由于左值同时为右值，故也可用左值表达式初始化。

```
const int u=4;   const int &v=u;   //用传统右值表达式 u 初始化 v
const int &w=4;                    //用传统右值表达式 4 初始化 w
int x=3;   const int &z=x;         //用传统左值表达式 x 初始化 z
```

函数调用的实参传递也可看作是对形参赋值，必须遵守上述类似规则

左值:可以在表达式左边的式子,右值:可以在表达式右边的式子

传统左值有址引用变量共享被引用的传统左值的内存。**理论上自己无内存**：故不能定义传统左值有址引用变量去引用传统左值有址引用变量(无内存)

例如：int & &u; //错：传统左值有址引用变量 u 去引用传统左值有址引用

```
int & *v;    //错：p 不能指向引用传统左值有址引用(无内存)
int x=3;
int &y=x; //对：y 共享 x 的内存，y=5 将使 x=5，x=6 将使 y=6
int &z=y; //对：z 引用 y 所引用的变量 x，注意 z、y 同类型。非 z 引用 y。
```

由于**引用变量无内存**，故数组元素不能为引用类型，即不能定义 int &a[2]。

由于数组有内存，故可被引用”int s[6]; int(&t)[6]=s;”。

位段无地址，不可被引用；register 可转为内存存储，故可被引用。

由于有址引用变量被编译为指针，const 和 volatile 有关指针的用法可推广至&定义的有址引用变量

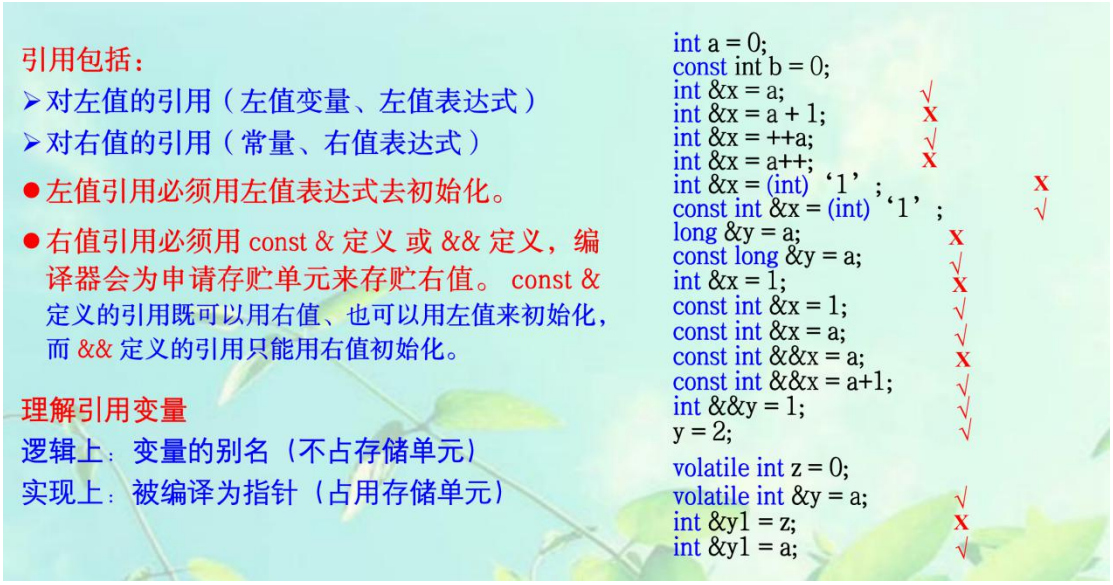
例如：“所指单元值只读的指针(地址)不能赋给所指单元值可写的指针变量”推广至引用为“所引用单元值只读的引用不能初始化所引用单元值可写的引用变量”。如前所述，反之是成立的。

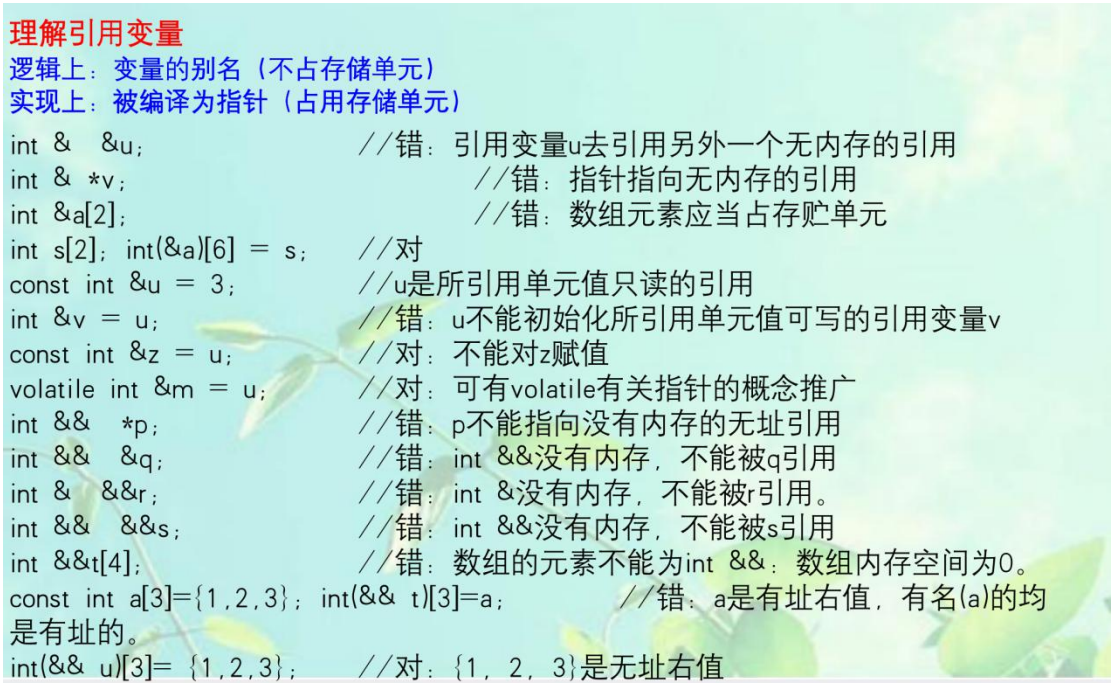
```
const int &u=3;    //u 是所引用单元值只读的引用
int &v=u;          //错：u 不能初始化所引用单元值可写的引用变量 v
int x=3; int &y=x;//对：可进行 y=4，则 x=4。
const int &z=y;    //对：不可进行 z=4。但若 y=5，则 x=5, z=5。
volatile int &m=y;//对：可有 volatile 有关指针的概念推广，m 引用 x。
```

9、&&定义的无址引用(不常用)

&&定义引用无址右值的变量。常见的无址右值为常量: int &&x=2;

注意，以上 x 是传统左值无址引用变量，即可进行赋值：x=3;





10、

```
enum struct RND{e=2, f=0, g, h}; //正确：e=2，f=0，g=1，h= 2
RND m= RND::h;              //必须用限定名 RND::h
int   n=sizeof(RND::h);     //n=4，枚举元素实现为整数
```

11、数组 a: 1, 2, 3, 4, 5, 6　　//第 1 个元素为 a[0][0]，第 2 个为 a[0][1],第 4 个为 a[1][0]

若上述 a 为全局变量，则 a 在数据段分配内存，1,2…6 等初始值存放于该内存。

若上述 a 为静态变量，则 a 的内存分配及初始化值存放情况同上。

若上述 a 为函数内定义的局部非静态变量，则 a 的内存在栈段分配，而初始化值则在数据段分配，最终函数使用栈段的内存。

C++数组并不存放每维的长度信息，因此也没有办法自动实现下标越界判断。每维下标的起始值默认为 0。
数组名 a 代表数组的首地址，其代表的类型为 int [2][3]或 int(*)[3]。

第 3 章 语句、函数及程序设计

3.1 C++的语句

1、asm 语句可在 C 或 C++程序中插入汇编代码。VS2019 编译器使用“_asm”插入汇编代码。static_assert 用于提供静态断言服务，即在编译时判定执行条件是否满足。使用格式为“static_assert(条件表达式，”输出信息“)”。不满足则编译报错

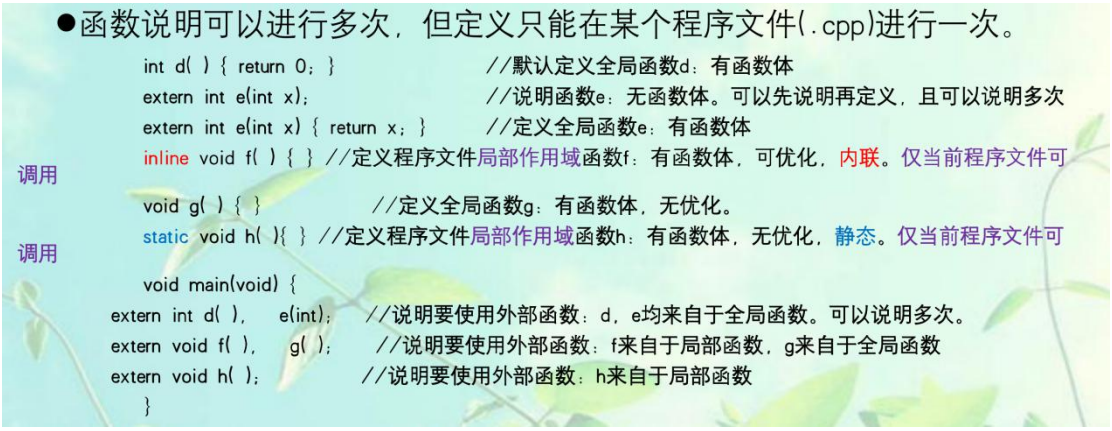
3.2 C++的函数

1、函数可说明或定义为四种作用域：（1）全局函数(默认)；（2）内联即 inline 函数；（3）外部即 extern 函数；（4）静态即 static 函数

全局函数可被任何程序文件(.cpp)的程序用，只有全局 main 函数不可被调用(新标准)。故它是全局作用域的。

内联函数可在程序文件内或类内说明或定义，只能被当前程序文件的程序调用。它是文件局部文件作用域的，可被编译优化(掉）。

静态函数可在程序文件内或类内说明或定义。类内的静态函数不是文件局部文件作用域的，程序文件内的静态函数是文件局部文件作用域的。



2、省略参数...表示可以接受 0 至任意个任意类型的参数。通常须提供一个参数表示省略了多少个实参。
3、声明或定义函数时也可定义参数默认值，调用时若未传实参则用默认值。
函数说明或者函数定义只能定义一次默认值。
默认值所用的表达式不能出现同参数表的参数。
所有默认值必须出现在参数表的右边，默认值参数中间不能出现没有默认值的参数。VS2019 的实参传递是自右至左的，即先传递最右边的实参.

```
int u=3;
int f(int x, int y=u+2, int z=3) { return x+y+z; }
int w=f(3)+f(2,6)+f(1,4,7); //等价于 w=f(3,5,3)+f(2,6,3)+f(1,4,7);
```

若同时定义有函数 int f(int m, ...); 则调用 f(3)可解释为调用 int f(int m, ...); 或调用 int f(int x, int y=u+2, int z=3)均可，故编译会报二义性错误。
4、编译会对内联 inline 函数调用进行优化，即直接将其函数体插入到调用处，而不是编译为 call 指令，这样可以减少调用开销，提高程序执行效率。
若函数为虚函数、或包含分支(if, switch,?:,循环,调用)，或取过函数地址，或调用时未见函数体，则内联失败。**失败不代表程序有错，只是被编译为函数调用指令。**

5、用 constexpr 定义的函数当其实参为常量时可以被更彻底的优化。
constexpr 函数内不能有 goto 语句或标号，也不能有 try 语句块。
constexpr 函数不能调用非 constexpr 的函数，如 printf 函数。
constexpr 函数不能定义或使用 static 变量、线程本地变量等永久期限变量。
类似 inline 函数，constexpr 函数的函数体可能被优化掉，其作用域相当于 static。
函数 main 为全局作用域，故不能定义为 constexpr 函数
constexpr 是 C++11 中新增的关键字，表示需要在编译时计算出其结果（得到一个常量）。
基本的常量表达式：字面值、全局变量/函数的地址、sizeof 等关键字返回的结果。
constexpr 用来修饰：变量、函数的返回值、构造函数。用 constexpr 定义的变量不能重新赋值（具有 const 特性）。

```
#include <iostream>
struct A {                                  Problem：(1) 将 int v = 10 改成 const int v = 10，结果会怎样？
                                                (2) 将constexpr A c(1);改成constexpr A c(0);，结
    int x;
    int y = 9;                              果会怎样？
    constexpr A(int a): x(a) { }                    //对：所有成员变量要么有缺省值、要么在初始化列
表中
    constexpr A(int a, int b): x(a) { y = b; x += y; }      //对：函数体内是常量表达式
    constexpr A(int a): x(a) { std::cout << x + y; }        //错：函数体内包含非常量表达式
    constexpr A(int a): y(a) { x = x + y; }                     //错：初始化列表中没有x
    A(): x(1),y(2) { std::cout << x+y; }                        //对：会生成普通的非constexpr对象
};
int main()
{   int v = 10;
    constexpr A a(v);  //错：非const变量的值不能在编译时确定(运行时才能确定)，因此a.x不能在编译时确
定
    constexpr A b;          //错：构造函数A()不能产生constexpr对象
    constexpr A c(1);       //对：生成constexpr对象c
    constexpr A d(1,2);     //对：生成constexpr对象d
    A e;                    //对：构造函数A()产生普通对象（非constexpr）
    A f();                      //对：声明一个函数e()，其返回值是A的对象
    int z1[c.x], z2[d.x+d.y];       //对：c、d是constexpr的，c.x、d.x+d.y的值在编译时确定，等价 z1[1]、
z2[5]
    int z3[e.x];            //错：e是普通对象（非constexpr的），e.x的值在编译时不能确定
    enum { X = c.x, Y = c.y};   //X = 1, Y = 9
    c.y = 10;               //错：c是constexpr的，不能改变其成员
    e.y = 10;               //对：e是普通对象
    f.y = 10;               //错：对象f不存在
}
```

3.3 作用域

1、程序可由若干代码文件(.cpp)构成，**整个程序为全局作用域：全局变量和函数属于此作用域。**
稍小的作用域是代码文件作用域：函数外的 static 变量和函数属此作用域。
更小的作用域是函数体：函数局部变量和函数参数属于此作用域。
在函数体内又有更小的复合语句块作用域。
最小的作用域是数值表达式：常量在此作用域。
除全局作用域外，同层作用域可以定义同名的常量、变量、函数。但他们为不同的实体。
如果**变量和常量是对象，则进入面向对象的作用域。**
同名变量、函数的作用域越小、被访问的优先级越高

```
代码文件"A.CPP"的内容如下。
extern int x;                   //B.CPP没定义全局变量x，此x即A.CPP自定义全局变量x
extern int x;                   //可以多次说明x
int x=2;                        //定义全局变量x，只能在A.cpp或B.cpp中共计定义一次
static int u=5;                 //模块静态变量u，A.cpp或B.cpp均可定义各自的同名变量
int v=3;                        //定义全局变量v，A.cpp定义后则B.cpp不能定义
static int y=3;                 //模块静态变量可在A.cpp和B.cpp中各自定义一次
int f( ) {//作用域范围越小，被访问的优先级越高：局部变量总是优先于外部变量被访问。。全局函数f只能被定义一次
    int u=4;                            //函数局部非静态变量：作用域为函数f内部
    static int v=5;             //函数局部静态变量：作用域为函数f内部
    v++;                        //优先访问自定义函数局部静态变量v，不会访问函数外部v
    return u+v+x+y;             //A.CPP自定义的u、v、y被优先访问，不会访问函数外部
}
static int g( ) {  return x+y;  }   //模块静态函数g可在A.cpp和B.cpp中各定义一次
```

```
代码文件“B.CPP”的内容如下。
extern int x;                  //欲访问模块外部变量x，即访问A.cpp定义的全局变量x
static int y=3;                //模块静态变量可在A.cpp和B.cpp中各定义一次
extern int f( );               //欲访问模块外部函数f，即A.cpp定义的全局函数f
static int g( ) {              //或模块静态函数可在A.cpp和B.cpp中各定义一次
   extern int x;               //可再次说明(本行可省)，函数g外部的变量x只有全局变量x
   extern int y;               //函数外部变量y(本行可省)，优先访问本模块静态变量y
   return x+y++;               //访问A.cpp的全局变量x、优先访问B.CPP自定义的y
}
void main( ){                  //A.CPP或者B.CPP只能有一个全局main函数定义
   int a=f( );                 //a=15：A.cpp定义的f返回后，f中的v仍然活着，v=6
   a=f( );                     //a=16：A.cpp定义的f返回后，f中的v仍然活着，v=7
   a=g( );                     //a=5
   a=g( );                     //a=6
}
```

3.4 生命期

作用域是变量等存在的空间，生命期是变量等存在的时间。

变量的生命期从其被运行到的位置开始，直到其生命结束（如被析构或函数返回等）为止。常量的生命期即其所在表达式。

函数参数或自动变量的生命期当退出其作用域时结束。

静态变量的生命期从其被运行到的位置开始，直到整个程序结束。

全局变量的生命期从其初始化位置开始，直到整个程序结束。

通过 new 产生的对象如果不 delete，则永远生存（内存泄漏）。

外层作用域变量不要引用内层作用域自动变量(包括函数参数)，否则导致变量的值不确定：因为内存变量的生命已经结束（内存已做他用）。

```
int x=2;                       //全局变量：生命期和作用域为整个程序
static int y=3;                //模块静态变量：生命期自第一次访问开始至整个程序结束
int f( )                       //全局函数f()：其作用域为整个程序，生命期从调用时开始
{
    int u=4;                   //函数自动变量：生命期和作用域为当前函数
    static int v=5;                  //函数静态变量：生命期自第一次调用开始至整个程序结束
    v++;
    return u+v+x+y;            //f()的生命期在此结束
}
static int g( ) { return x; }//静态函数g()：其作用域为“A.cpp”文件，生命期从调用时开始
```

```
extern int x;
static int y=3;                //模块静态变量：生命期自第一次访问开始至整个程序结束
extern int f( );
static int g( )                //静态函数g()：其作用域为“B.cpp”文件，生命期从调用时开始
{
    return x+y++;              //x由“A.cpp”定义，y由“B.cpp”定义
}
void main( )                   //全局函数main()：其生命期和作用域为整个程序
{
    int a=f( );                //函数自动变量a：生命期和作用域为当前函数
    const int&&b=2;            //传统右值无址引用变量b引用常量2：产生匿名变量存储2
    a=f( );                          //main()开始全局函数f()的生命期
    a=3;                             //常量3的生命期和作用域为当前赋值表达式
    a=g( );                          //main()开始“B.cpp”的静态函数g()的生命期
}//为b产生的匿名变量的生命期在main()返回时结束
```

第二部分 object oriented 部分基础

第 4 章 C++的类

4.1 类的声明与定义

1、类保留字：class、struct 或 union 可用来声明和定义类。

class 类型名;//前向声明 **class 默认 private struct 默认 public**

class 类型名{

private:

私有成员声明或定义;

protected:

保护成员声明或定义;

public:

公有成员声明或定义;

};

2、使用 private、protected 和 public 保留字标识主体中每一区间的访问权限，同一保留字可以多次出现；

同一区间内可以有数据成员、函数成员和类型成员，习惯上按类型成员、数据成员和函数成员分开；

成员在类定义体中出现的顺序可以任意，**函数成员的实现既可以放在类的外面，也可以内嵌在类定义体中**；但是数据成员的定义顺序与初始化顺序有关。

若函数成员在类定义体外实现，则在函数返回类型和函数名之间，**应使用类名和作用域运算符"::"**来指明该函数成员所属的类。防止自递归

3、在类定义体中**允许对所数据成员定义默认值**，若在构造函数的"："和函数体的"{"之间对其进行了初始化，则默认值无效，否则用默认值初始化；

类定义体的最后一个花括号后要跟有分号作为定义体结束标志。

构造函数和析构函数都不能定义返回类型。

如果类没有自定义的构造函数和析构函数，且有非公开实例数据成员等情形，则 C++为类生成默认的无参构造函数和析构函数。

构造函数的参数表可以出现参数，因此可以重载。

4、构造函数和析构函数：是类封装的两个特殊函数成员，都有固定类型的隐含参数 this。this 指针可以指向调用的类自己

构造函数：函数名和类名相同的函数成员。

析构函数：函数名和类名相同且带波浪线的参数表无参函数成员。

定义变量或其生命期开始时自动调用构造函数，生命期结束时自动调用析构函数。

同一个对象仅自动构造一次。**构造函数是唯一不能被显式（人工）调用的函数成员。**

5、构造函数用来为对象申请各种资源，并初始化对象的数据成员。构造函数有隐含参数 this，可以在参数表定义若干参数，用于初始化数据成员。

析构函数是用来毁灭对象的，析构过程是构造过程的逆过程。析构函数释放对象申请的所有资源。

析构函数既能被显式调用，也能被隐式（自动）调用。由于只有一个固定类型的 this，故不可能重载，**只能有一个析构函数。**

若实例数据成员有指针，应当防止反复析构（用指针是否为空做标志）。

```
#include <alloc.h>
struct STRING {
   typedef char * CHARPTR;        //定义类型成员
   CHARPTR s;                     //定义数据成员
   int strlen( );                 //声明函数成员，求谁的长(有this)
   STRING(CHARPTR);               //声明构造函数，有this
   ~STRING( );                    //声明析构函数，有this
};
int STRING::strlen( ){            //用运算符::在类体外定义
   int k;
   for(k=0; s[k]!=0; k++);
   return k;
}
```

```
STRING::STRING(char *t) {//用::在类体外定义构造函数,无返回类型
   int k;
   for(k =0; t[k]!=0; k++);
   s=(char *)malloc(k+1);       //s等价于this->s
   for(k=0; (s[k]=t[k])!=0; k++);
}
STRING::~STRING( ) {//用::在类体外定义析构函数,无返回类型
   free(s);
}
struct STRING x("simple");     //struct可以省略
void main( ){
   STRING y("complex"), *z=&y;
   int m=y.strlen( );          //当前对象包含的字符串的长度
   m=z->strlen( );
} //返回时自动调用y的析构函数
```

6、程序不同结束形式对对象的影响：

exit 退出：局部自动对象不能自动执行析构函数，故此类对象资源不能被释放。静态和全局对象在 exit 退出 main 时自动执行收工函数析构。

abort 退出：所有对象自动调用的析构函数都不能执行。局部和全局对象的资源都不能被释放，即 abort 退出 main 后不执行收工函数。

return 返回：**隐式调用的析构函数得以执行。局部和全局对象的资源被释放。**

int main( ){ …; if (error) return 1; …;}

提倡使用 return。如果用 abort 和 exit，则要显式调用析构函数。另外，使用异常处理时，自动调用的析构函数都会执行。

7、接受与删除编译自动生成的函数：**default=接受, delete=删除。**

若类 A 没有定义任何构造函数，则编译器会自动提供无参的构造函数 **A( )**（实际上是空操作，相当于一条 return 语句）；如果 A 定义了任何构造函数，编译器就不会自动提供无参构造函数。

不管 A 是否定义了构造函数，A 类只要没有定义带引用参数 A & 的构造函数 A(const A &a)，编译器会自动提供该构造函数 **A(const A &a)**（利用浅拷贝实现该构造函数）。

可以利用 default、delete 强制说明接受、删除编译器提供的构造函数，如 A() = default、A(const A &a) =delete。

```
例4.4使用delete禁止构造函数以及default接受构造函数。
struct A {
    int x=0;
    A(int m): x(m) { }
    A(const A &a) = default;    //接受编译生成的拷贝构造函数A(const A&)
};
void main(void) {
    A x(2);                  //正确：调用程序员自定义的单参构造函数A(int)
    A y(x);                  //正确：调用编译生成的拷贝构造函数A(const A &)
    A u;                     //错误：调用构造函数A( )，编译器不提供缺省的A( )（因为有别
的构造函数）
    A v( );                  //正确：声明一个函数v( )，其返回类型为A
}   //A v( )  《=》 extern  A v( )
em：  (1) 如果在类A中增加一条语句 A() = default，结果会怎样？
      (2) 如果删除类A中的 A(const A &a) = default，结果会怎样？
      (3) 如果将A中的 A(const A &a) = default 写成 A(const A &a) = delete ，结果会怎样？
```

4.2 成员访问权限及其访问

封装机制规定了数据成员、函数成员、类型成员 的访问权限。包括三类：

private：私有成员，本类函数成员可以访问；派生类函数成员、其他类函数成员和普通函数都不能访问。

protected：保护成员，本类和派生类的函数成员可以访问，其他类函数成员和普通函数都不能访问。

public：公有成员，任何函数均可访问。

类的友元不受这些限制，可以访问类的所有成员。另外，通过强制类型转换可突破访问权限的限制。

构造函数和析构函数可以定义为任何访问权限。不能访问构造函数则无法用其初始化对象。

```
#include <iostream>
class A {
    int x;     //私有，只能被本类的成员访问
    typedef char * POINTER1;
public:        //公有，能被任何函数使用
    int y;
protected:   //保护：只能被本类及继承类的成员访问
    int z;
public:        //公有，能被任何函数使用
    typedef char * POINTER2;
    A(int x,int y,int z): x(x),y(y),z(z) {}
    int sum() {
        return x+y+z;
    }
};
```

```
#include <iostream>
void main(void) {
    A a(1,2,3);
    std::cout << a.x;        //错：不能访问私有成员
    std::cout << a.y;        //对
    std::cout << a.z;        //错：不能访问保护成员
    std::cout << a.sum(); //对
    char s[] = "abcd";
    A::POINTER1 p1 = s;   //错：类型A::POINTER1是私有的
    A::POINTER2 p2 = s; //对
    p2 = "abcd";   //错："abcd"是const char *，p2是char *
    p2 = (char *)"abcd"; //对，等价 p2 = (A::POINTER2)"abcd"
}
```

4.3 内联、匿名类及位段

1、函数成员的内联说明：

在类体内定义的任何函数成员都会自动内联。

在类内或类外使用 inline 保留字说明函数成员。

内联失败：有分支类语句、定义在使用后，取函数地址，定义(纯)虚函数。

内联函数成员的作用域局限于当前代码文件。

2、位段成员：按位分配内存的数据成员。

class、struct 和 union 都能定义位段成员；

位段类型必须是字节数少于整数类型的类型，如：

char，short，int，long long, enum（实现为 int：简单类型）

相临位段成员分配内存时,可能出现若干位段成员共处一个字节，或一个位段成员跨越多个字节。因按字节编址，故位段无地址。

3、对于没有对象的匿名联合，C++兼容 C 的用法：

没有对象的全局匿名联合必须定义为 static，局部的匿名联合不能定义为 static ；

匿名联合内只能定义公有数据成员；

4.4 new 和 delete

1、用“new 类型表达式 {}”可使分配的内存清零，若“{}”中有数值可用于初始化。delete 为运算符，操作数为指针类型值表达式，先调用析构函数，然后底层调用 free。(调用类的时候 new 和 malloc 区别就在于构造函数有没有调用)

2、**new <类型表达式> //后接()或{}用于初始化或构造。[]可用于数组元素**

类型表达式：int *p=new int; //等价 int *p=new int(0);

数组形式仅第一维下标可为任意表达式，其它维为常量表达式：int (*q)[6][8]=new int[x+20][6][8];

为对象数组分配内存时，必须调用参数表无参构造函数,生成一个指针

没有[],生成简单指针

有[],生成指针,并开辟多少位的空间

3、delete <指针>

指针指向**非数组的单个实体**：delete p; 可能调析构函数。

4、delete [ ]<数组指针>

指针指向任意维的数组时：delete [ ] q;

如为对象数组，对所有对象(元素)调用析构函数。

若数组元素为简单类型，则可用 delete <指针>代替。

定义二维整型动态数组的类。

```
#include <alloc.h>
#include <process.h>
class   ARRAY{              //class 体的缺省访问权限为 private
    int   *a, r, c;
public:                     //访问权限改为 public
    ARRAY(int x, int y);
    ~ARRAY( );
};
ARRAY::ARRAY(int x, int y){
    a=new int[(r=x)*(c=y)];      //可用 malloc：int 为简单类型
}
ARRAY::~ARRAY( ){           //a 指向的简单类型 int 数组无析构函数
    if(a) { delete [ ]a; a=0; }   //可用 free(a)，也可用 delete a
}
ARRAY   x(3, 5);                    //开工函数构造，收工函数析构 x
void main(void){
    int error=0;
    ARRAY   y(3, 5), *p;            //退出 main 时析构 y
    p=new ARRAY(5, 7);       //不能用 malloc，ARRAY 有构造函数
    delete   p;                 //不能用 free，否则未调用析构函数
}                                                   //退出 main 时，y 被自动析构
//程序结束时，收工函数析构全局对象 x
```

4.5 隐含参数 this

this 指针是一个特殊的指针，它是**普通函数成员隐含的第一个参数**，其类型是指向要调用该函数成员的对象的 **const 指针。**

当对象调用函数成员时，对象的地址作为函数的第一个实参首先压栈，通过这种方式将对象地址传递给隐含参数 this。

构造函数和析构函数的 this 参数类型固定。例如 A::~A()的 this 参数类型为

A*const this; //析构函数的 this 指向可写对象，但 this 本身是只读的

注意：可用*this 来引用或访问调用该函数成员的普通、const 或 volatile 对象；类的静态函数成员没有隐含的 this 指针；**this 指针不允许移动**

```
#include <iostream.h>
class   TREE{
    int     value;
    TREE   *left, *right;
public:
    TREE (int);       //this 类型: TREE * const this
     ~TREE( );        //this 类型: TREE * const this，析构函数不能重载
      const TREE *find(int) const; //this 类型: const TREE * const this
};
TREE::TREE(int value){   //隐含参数 this 指向要构造的对象
    this->value=value;       //等价于 TREE::value=value
```

```
        left=right=0;       //C++提倡空指针 NULL 用 0 表示
    }
```

4.6 对象的构造与析构

1、类可能会定义只读和引用类型的非静态数据成员，**在使用它们之前必须初始化**；若无默认值，**该类必须定义构造函数初始化这类成员(参数表)**。

类 A 还可能定义类 B 类型的非静态对象成员，若对象成员必须用带参数的构造函数构造，则 **A 必须定义有初始化的构造函数(自定义的类 A 的构造函数, 传递实参初试化类 B 的非静态对象成员：缺省的无参的 A( )只调用无参的 B( ))。**

构造函数的初始化位置在参数表的":"后，所有数据成员都必须在此初始化，未列出的成员用其默认值值初始化，未列出且无默认值的非只读、非引用、非对象成员的值根据对象存储位置可取随机值（栈段）或 0 及 nullptr 值（数据段）。

按定义顺序初始化或构造数据成员（大部分编译支持）。

2、**对象数组的每个元素都必须初始化，默认采用无参构造函数初始化。**

单个参数的构造函数能自动转换单个实参值成为对象。

若类未自定义构造函数，且类包含私有实例数据成员等调价满足时，编译会自动生成构造函数。

一旦自定义构造函数，将不能接受编译生成的构造函数，除非用 default 等接受。

用常量对象做实参，**总是优先调用参数为&&类型的构造函数**；用变量等做实参，总是优先**调用参数为&类型的构造函数**。

例:包含只读、引用及对象成员的类。

```
    class A {
        int   a;
    public:
        A(int x) { a=x;}          //重载构造函数，自动内联
        A( ) { a=0; }             //重载构造函数，自动内联
    };
    class B {
        const   int   b;     //数据成员不能在定义的同时初始化
        int   c, &d, e, f;    //b, d, g, h 只能在构造函数体前初始化
        A       g, h;     //数据成员按定义顺序 b, c, d, e, f, g, h 初始化
    public:   //类 B 构造函数体前未出现 h，故 h 用 A( )初始化
    B(int y): d(c), c(y), g(y) , b(y), e(y) {   //自动内联
          c += y;
        f = y;
    }   //f 被赋值为 y
};
void main(void) {
    int x(5);                       //int x=5 等价于 int x(5)
    A a(x), y=5;                        //A y=5 等价于 A y(5)
    A *p=new A[3]{ 1, A(2) };   //初始化的元素为 A(1),  A(2), A(0)
    B b(7), z = (7, 8);             //B z=(7, 8) 等价于 B z(8), 等号右边必单值
    delete [ ] p;                   //防止内存泄漏：new 产生的所有对象必须用 delete 释放
}                                                   //故(7, 8)为 C 的扩号表达式，(7, 8) = 8
```

第 5 章 成员与成员指针

5.1 实例成员指针

1、运算符.*和->*均为双目运算符，优先级均为第 14 级，结合性自左向右。
.*的左操作数为类的实例(对象)，右操作数为指向实例成员的指针。
->*的左操作数为对象指针，右操作数为指向该对象实例成员的指针。
这个不能指向构造函数
2、实例成员指针是成员相对于**对象首地址的偏移**，不是真正的代表地址的指针。
实例成员指针不能移动：
数据成员的大小及类型不一定相同，移动后指向的内存可能是某个成员的一部分，或者跨越两个(或以上)成员的内存；
即使移动前后指向的成员的类型正好相同，这两个成员的访问权限也有可能不同，移动后可能出现越权访问问题。
实例成员指针不能转换类型，否则便可以通过类型转换，间接实现实例成员指针移动。
成员指针代表了偏移量,普通指针指向地址

```
#include <iostream.h>
    struct A{
        int   i;        //公有的成员 i
    private:
        long   j;
    public:
        int f( ) { cout << "f() ";     return 1;   }
    private:
        void   g( ) { cout << "g() ";   }
    } a;
```

```
void main(void){
   int  A::*pi = &A::i;       //普通数据成员指针pi指向public成员A::i
   int (A::*pf)( ) = &A::f;   //普通函数成员指针pf指向函数成员A::f
   int  x = a.*pi;            //等价于 x=a.*(&A::i) = a.A::i = a.i
   x = (a.*pf)( );            //.*的优先级低，故用(a.*pf)
   pi++;                      //错误， pi不能移动，否则指向私有成员j
   pf += 1;                   //错误， pf不能移动
   long y = (long) pi;        //错误，pi不能转换为长整型
   x = x+ sizeof(int)         //对
   pi = (int A::*)x;          //错误，x不能转换为成员指针
}
```

5.2 const、volatile 和 mutable

1、const 只读，volatile 易变，mutable 机动。
const 和 volatile 可以定义变量、类的数据成员、函数成员及普通函数的参数和返回类型。
mutable 只能用来定义类的数据成员。
含 const 数据成员的类必须定义构造函数，且数据成员必须在构造函数参数表之后，函数体之前初始化。
含 volatile、mutable 数据成员的类则不一定需要定义构造函数。
普通函数成员参数表后出现 const 或 volatile，修饰隐含参数 this 指向的对象。出现 const 表示 this 指向的对象(其非静态数据成员)**不能被函数修改，但可以修改 this 指向对象的非只读类型的静态数据成员**。
构造或析构函数的 this 不能被说明为 const 或 volatile 的 (**即要构造或析构的对象应该能被修改，且状态要稳定不易变**)。
对隐含参数的修饰还会会影响函数成员的重载：
普通对象应调用参数表后不带 const 和 volatile 的函数成员；
const 和 volatile 对象应分别调用参数表后出现 const 和 volatile 的函数成员，否则编译程序会对函数调用发出警告。
2、有址引用变量(&)只是被引用对象的别名，**被引用对象自己负责构造和析构**，该**引用变量(逻辑上不分配内存的实体)不必构造和析构**。

无址引用变量(&&)常用来**引用缓存中的常量对象，该引用变量(逻辑上不分配缓存的实体)不必构造和析构**。无址引用变量可为左值，但若同时用 const 定义则为传统右值。
如果 A 类型的有址引用变量 r 引用了 **new 生成**的(一定有址的）对象 x，则应使用 **delete &r** 析构 x，同时释放其所占内存。
r.~A( )仅**析构 x 而不释放其所占内存(由 new 分配)，造成内存泄漏。应该用　delete &r;**
引用变量必须在定义的同时初始化，引用参数则在调用函数时初始化。有址传统左值引用变量和参数必须用同类型的左值表达式初始化。
3、所谓机动是指在整个对象为只读状态时，其每个成员理论上都是不可写的，但若某个成员是 mutable 成员，则该成员在此状态是可写的。
例如，产品对象的信息在查询时应处于只读状态，但是其成员“查询次数”应在此状态可写，故可以定义为“机动”成员。
保留字 mutable 还可用于定义 Lambda 表达式的参数列表是否允许在 Lambda 的表达式内修改捕获的外部的参数列表的值。

```
#include <string.h>
#include <iostream.h>
class TUTOR{
   char   name[20];
   const  char  sex;       //性别为只读成员
   int    salary;
public:
   TUTOR(const char *name, const TUTOR *t);
   TUTOR(const char *name, char gender, int salary);
   const char *getname( ) { return name; }
   char *setname(const char *name);
};
```

```
TUTOR::TUTOR(const char *n, const TUTOR *t): sex(t->sex) {
   strcpy(name, n);   salary = t->salary;
} //只读成员sex必须在构造函数体之前初始化
TUTOR::TUTOR(const char *n, char g, int s): sex(g), sarlary(s) {
   strcpy(name, n);
} //非只读成员sarlary可在函数体前初始化，也可在体内再次赋值
char *TUTOR::setname(const char*n) {
   return strcpy(name, n); //strcpy的返回值为name
}
void main(void) {
   TUTOR  wang("wang", 'F', 2000);
   TUTOR  yang("yang", &wang);
   *wang.getname( ) = 'W';   //错误:不能改wang.getname( )指的字符
   *yang.setname("Zang") = 'Y';
}
```

5.3 静态数据变量

1、静态数据成员是使用 static 说明或定义的类的数据成员。
2、静态数据成员通常在类的里面说明，在**类的外面唯一定义一次**。
3、静态数据成员一般用来描述类的总体信息，例如对象总个数。
4、实例数据成员可以定义默认值，但**静态数据成员不能定义默认值**。
5、静态数据成员在类中初始化只能定义为 inline static、const static、const inline 类型(保留字顺序可变）。
6、静态数据成员不管是否用 inline、const 说明，在所有代码文件只有一个副本。
7、函数中的**局部类不能定义静态数据成员，容易造成生命期矛盾**。
8、静态数据成员不能定义为位段成员。

5.4　静态函数成员

1、静态函数成员通常在类里以 static 说明或定义，**它没有 this 参数**。
2、有 this 的构造和析构函数、虚函数及纯虚函数都不能定义为静态函数成员。
3、静态函数成员一般用来访问类的总体信息，例如对象总个数。
4、静态函数成员可以重载、内联、定义默认值参数。
5、静态函数成员同实例成员的继承、访问规则没有没有太大区别。
6、静态函数成员的参数表后不能出现 const、volatile、const volatile 等修饰符。
7、静态函数成员的返回类型可以同时使用 inline、const、volatile 等修饰。

5.5 静态成员指针

静态成员指针是指向类的静态成员的指针，包括静态数据成员指针和静态函数成员指针。
静态数据成员的存储单元为该类所有的对象共享，因此，通过该指针修改成员的值时会影响

到所有对象该成员的值。

静态数据成员除了具有访问权限外，同普通变量没有本质区别；静态成员指针则和普通指针没有任何区别。

变量、数据成员、普通函数和函数成员的参数和返回值都可以定义成静态成员指针。

必考这种玩意

```
#include <iostream>
using namespace std;
class CROWD{
   int  age;
   char  name[20];
public:
   static int number;
   static int getn( ) { return number;}
 CROWD(char *n, int a){
     strcpy(name,n);
     age=a;  number++;
   }
   ~CROWD( ) { number – –; }
};
int CROWD::number=0;
void main(void){
   int *d=&CROWD::number; //普通指针指向静态数据成员
   int (*f)( )=&CROWD::getn; //普通函数指针指向静态函数成员
   cout<<“\nCrowd number=”<<*d;
   //类CROWD无对象时访问静态成员
   CROWD zan("zan", 20);
   //d=&zan.number; 等价于如下
    d=&CROWD::number;
   cout<<"\nCrowd number="<<*d;
   CROWD tan("tan", 21);
   cout<<"\nCrowd number="<<(*f)( );
}
```

静态成员指针与普通成员指针有很大区别。静态成员指针存放成员地址，普通成员指针存放成员偏移；静态成员指针可以移动，普通成员指针不能移动；静态成员指针可以强制转换类型，普通成员指针不能强制转换类型。

```
struct A{
   int a, *b;
   int A::*u;  int A::*A::*x;
   int A::**y; int *A::*z;
   static int c, A::*d;
};z;
int A::c=0;
int A::*A::d=&A::a;
void main(void){
   int i, A::**m;
   z.a=5;   z.u=&A::a;  i=z.*z.u;
   z.x=&A::u;     i=z.*(z.*z.x);
   m=&A::d;
   m=&z.u;        i=z.**m;
   z.y=&z.u;      i=z.**z.y;
   z.b=&z.a;
   z.z=&A::b;     i=*(z.*z.z);
}
```

第 6 章 继承与构造

6.1 单继承类

1、派生类与基类：**接受成员的新类称为派生类**，如例中的 Point 类；提供成员的类称为基类，如例中的 Location 类。

基类是对若干个派生类的抽象，提取了派生类的公共特征；而派生类是基类的具体化，通过增加属性或行为变为更有用的类型。

派生类可以看作基类定义的延续，先定义一个抽象程度较高的基类，该基类中有些操作并未实现；然后定义更为具体的派生类，实现抽象基类中未实现的操作。

2、C++通过多种控制派生的方法获得新的派生类，可在定义派生类时：

添加新的数据成员和函数成员；

改变继承来的基类成员的访问权限；

修改继承来的基类成员的访问权限；

重新定义同名的数据和函数成员。

3、单继承的定义格式：

class <派生类名>:<继承方式><基类名>

{

<派生类新定义成员>
<派生类重定义基类同名的数据和函数成员>
<派生类声明修改基类成员访问权限>
};

<继承方式>指明派生类采用什么继承方式从基类获得成员，分为三种：private 表示私有基类；protected 表示保护基类；public 表示公有基类。**如果 class 默认为 private 继承 structure 默认为 public 继承**

注意区别继承方式和访问权限。

6.2 继承方式

1、公有继承：基类的公有成员和保护成员派生到派生类时，都保持原有的状态；
保护继承：基类的公有成员和保护成员派生后都成为派生类的保护成员；
私有继承：基类的公有成员和保护成员派生后都作为派生类的私有成员
基类的私有成员同样也被继承到派生类中，构成派生类的一部分，但对派生类函数成员不可见，不能被派生类函数成员访问。
若派生类函数成员要访问基类的私有成员，则必须将其声明为**基类的成员友元**。
在派生类外部，对其成员访问的权限：
对于新定义成员，按定义时的访问权限访问；
对于继承来的基类成员，取决于这些成员在派生类中的访问权限，**与其在基类中定义的访问权限无关**。

2、继承方式为 private 时，基类成员在派生类中的访问权限变为 private。不合理时可以使用“**基类名::成员**”或“**using 基类名::成员**”修改某些成员的访问权限，派生类不能再定义同名的成员。

```
class POINT:private LOCATION{ //private 可省略
    int visible;
public:
    LOCATION::getx;     //修改权限成 public
    LOCATION::gety;     //修改权限成 public
    int isvisible( ){ return visible; }
    void show( ), hide( );
    void moveto(int x,int y);
    POINT(int x,int y):LOCATION(x,y){ visible=0; }
    ~POINT( ){ hide( ); }
};
```

3、派生类中改写基类同名函数时，要注意区分这些同名函数，否则可能造成自递归调用。

6.3 成员访问

1、标识符的作用范围可分为从小到大四种级别：**①作用于函数成员内；②作用于类或者派生类内；③作用于基类内；④作用于虚基类内。**

标识符的作用范围越小，被访问到的优先级越高。如果希望访问作用范围更大的标识符，则可以用类名和作用域运算符进行限定。

```
class LIST{
   struct NODE{              //定义节点类
      int val;       NODE *next;
      NODE(int v, NODE *p){ val=v; next=p; }
      ~NODE( ){delete next;next=0;}
   }*head;                   //定义数据成员
public:
   int insert(int),  contains(int);
   LIST( ){ head=0; }        //0表示空指针
   ~LIST( ){ if(head){ delete head; head=0; //0表示空指针} }
};
int LIST::contains(int v){ //搜索链表，查询是否存在该节点
   NODE *h=head;
   while((h!=0)&&(h->val!=v))  h=h->next;
   return h!=0;              //0表示空指针
}
```

```
int LIST::insert(int v){          //在链表中插入新增节点
   head=new NODE(v,head);
   return 1;
}
class SET:protected LIST{         //采用保护继承方式
   int used;                      //集合元素的个数
public:
   LIST::contains;                //修改contains函数访问权限
   int insert(int);               //需要改变used值，因此改写insert函数
   SET( ){ };                     //等价于SET( ):LIST( ){ };
};
int SET::insert(int v){           //LIST::insert中的LIST不能省略，否则自递归
   if(!contains(v)&&LIST::insert(v))  return  ++used;
   return 0;
}
void main(void) { SET s;     s.insert(3);  s.contains(3); }
```

6.4 构造与析构

1、单继承派生类的构造顺序比较容易确定：

调用虚基类的构造函数；

调用基类的构造函数；

按照派生类中数据成员的声明顺序，依次调用数据成员的构造函数或初始化数据成员；

最后执行派生类的构造函数构造派生类。

析构是构造的逆序。

2、以下情况派生类必须定义自己的构造函数：

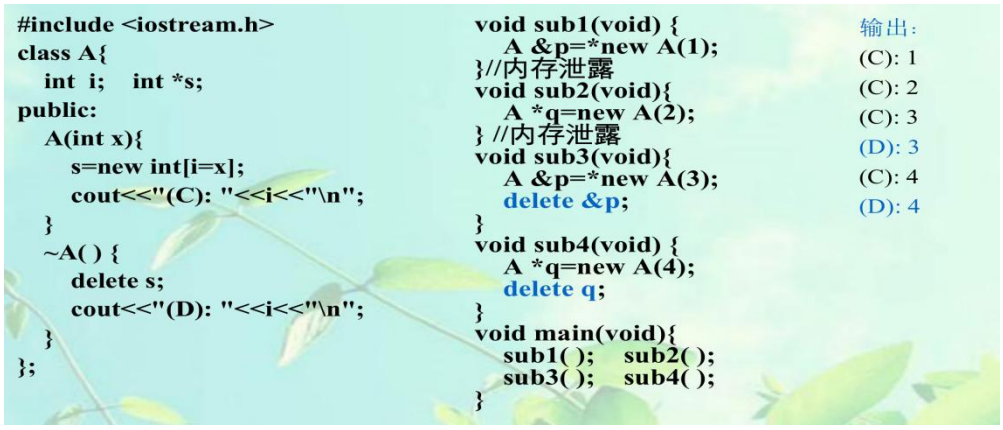
虚基类或基类只定义了带参数的构造函数；

派生类自身定义了引用成员或只读成员；

派生类需要使用带参数构造函数初始化的对象成员。

3、如果虚基类和基类的构造函数是**无参**的，则构造派生类对象时，构造函数可以不用显式调用它们的构造函数，编译程序会**自动调用**虚基类或基类的无参构造函数。

```
#include <iostream.h>
class A{
   int a;
public:
   A(int x):a(x){cout<<a;}//也可在构造函数体内再次对a赋值
   ~A( ){cout<<a;}
};
class B:A{    //私有继承，等价于class B: private A{
   int b,c;
   const int d;//B中定义有只读成员，故必须定义构造函数初始化
   A x,y;
public:
   B(int v):b(v),y(b+2),x(b+1),d(b),A(v){//注意构造与出现顺序无关
      c=v;       cout<<b<<c<<d;      cout<<"C";
   }
   ~B( ){cout<<"D";}                  //派生类成员实际构造顺序为b,d,x,y
};
void main(void){ B z(1); }            //输出结果：123111CD321
```



```
#include <iostream.h>
class A{
   int  i;   int *s;
public:
   A(int x){
      s=new int[i=x];
      cout<<"(C): "<<i<<"\n";
   }
   ~A( ) {
      delete s;
      cout<<"(D): "<<i<<"\n";
   }
};
void sub1(void) {
   A &p=*new A(1);
}//内存泄露
void sub2(void){
   A *q=new A(2);
} //内存泄露
void sub3(void){
   A &p=*new A(3);
   delete &p;
}
void sub4(void) {
   A *q=new A(4);
   delete q;
}
void main(void){
   sub1( );   sub2( );
   sub3( );   sub4( );
}
```

```
输出:
(C): 1
(C): 2
(C): 3
(D): 3
(C): 4
(D): 4
```

6.5 父类与子类

如果派生类的继承方式为 public，则这样的派生类称为基类的子类，而相应的基类则称为派生类的父类。

C++允许父类指针直接指向子类对象，也允许父类引用直接引用子类对象。

通过父类指针调用虚函数时晚期绑定，根据对象的实际类型绑定到合适的成员函数。

父类指针实际指向的对象的类型不同，虚函数绑定的函数的行为就不同，从而产生多态。

4、编译时，只能把父类指针指向的对象都当作父类对象。因此编译时：

父类指针访问对象的数据成员或函数成员时，不能超越父类为相应对象成员规定的访问权限；

也不能通过父类指针访问子类新增的成员，因为这些成员在父类中不存在，编译程序无法识别。

```
#include <iostream.h>
class POINT{
      int x,y;
public:
      int getx( ){return x; }
      int gety( ){return y; }
      void show( ){cout<<“Show a point\n”; }
      POINT(int x,int y){POINT::x=x; POINT::y=y; }
};
class CIRCLE:public POINT{   //公有继承
      int r;                 //私有成员
public:
      int getr( ){return r; }
      void show( ){cout<<“Show a circle\n”; }
      CIRCLE(int x,int y,int r):POINT(x,y){CIRCLE::r=r; }
};
void main(void){
      CIRCLE c(3,7,8);
      POINT *p=&c;//父类对象指针 p 可以直接指向子类对象，不用类型转换
      cout<<“The circle with radius ”<<c.getr();
```

```
    //不能使用 p->getr( ), 因为 getr( )函数不是父类的函数成员
    //编译程序无法通过检查
    cout<<" is at ("<<p->getx( )<<","<<p->gety( )<<")\n";
    p->show( );
    //p 虽然指向子类对象，但调用的是父类的 show 函数
    //如果定义虚函数,那就可以调用子类的 show 函数
}
```

输出结果:

The circle with radius 8 is at (3,7)

Show a point

第 7 章 可访问性

7.1 作用域

1、**单目::用于限定全局标识符**(类型名、变量名、函数名以及常量名等)。

双目::用于限定类的枚举元素、数据成员、函数成员以及类型成员等。双目运算符::还用于限定名字空间成员，以及恢复从基类继承的成员的访问权限。

在类体外定义数据和函数成员时，必须用双目::限定类的数据和函数成员，以便区分不同类之间的同名成员。

2、作用域分为面向对象的作用域、面向过程的作用域(C 传统的作用域，含被引用的名字空间及成员) 。

面向过程的：词法单位的作用范围从小到大可以分为四级：①作用于表达式内(常量)，②作用于函数内(参数和局部自动变量、局部类型)，③作用于程序文件内(static 变量、函数)，④作用于整个程序(全局变量、函数、类型) 。

面向对象的：词法单位的作用范围从小到大可以分为五级：①作用于表达式内(常量) ，②作用于函数成员内(参数和局部自动变量、局部类型) ，③作用于类或派生类内，④作用于基类内，⑤作用于虚基类内。

标识符作用域越小，被访问优先级就越高。当函数成员的参数和数据成员同名时，优先访问的是函数成员的参数。作用域层次：面向对象->面向过程。

7.2 名字空间

1、名字空间必须在全局作用域内用 namespace 定义，不能在**类、函数及函数成员内定义，最外层名字空间名称必须在全局作用域唯一。**

```
    namespace   A { int x, f( ) {return 1; }; class B {/*…*/}; };
    class B { namespase C { int y; }; int z; };     //错
    namespace B::C { int z; };                                  //错
    void f( ) { namespace E { int x}; };                  //错
```

同一名字空间内的标识符名必须唯一，不同名字空间内的标识符名可以相同。当程序引用多个名字空间的同名成员时，可以用名字空间加作用域运算符::限定

2、名字空间(包括匿名名字空间)**可以分多次定义**:

可以先在初始定义中定义一部分成员，然后在扩展定义中再定义另一部分成员；

或者先在初始定义中声明的函数原型，然后在扩展定义中再定义函数体；

初始定义和扩展定义的语法格式相同。

保留字 using 用于指示程序要引用的名字空间，或者用于声明程序要引用的名字空间内的成员。在引用名字空间的某个成员之前，该成员必须已经在名字空间中声明了原型或进行

了定义。

```
#include <iostream>
using namespace std;
namespace ALPHA {   //初始定义 ALPHA
    extern int x;                    //声明整型变量 x
    void g(int);          //声明函数原型 void g(int)
    void g(long) {//定义函数 void g(long)
        cout << "Processing a long argument.\n";
    }
}
using ALPHA::x;  //声明引用变量 x
using ALPHA::g;  //声明引用 void g(int)和 g(long)
namespace ALPHA {   //扩展定义 ALPHA
    int x=5;        //定义整型变量 x
    void g(int a)         //定义函数 void g(int)
    {cout << "Processing a int argument.\n";     }
    void g(void)          //定义新的函数 void g(void)
    {cout << "Processing a void argument.\n";    }
}
void main(void) {
    g(4);             //调用函数 void g(int)
    g(4L);                           //调用函数 void g(long)
    cout << "X= " << x;     //访问整型变量 x
    g(void);        //using 之前无该 原型，失败
}
```

3、名字空间成员三种访问方式：①直接访问成员，②引用名字空间成员，③引用名字空间。直接访问成员的形式为：**<名字空间名称>::<成员名称>**。直接访问**总能唯一的访问名字空间成员**。

引用成员的形式为：**using <名字空间名称>::<成员名称>**。如果引用时只声明或定义了一部分重载函数原型，则只引用这些函数，并且**引用时只能给出函数名**，不能带函数参数。

引用名字空间的形式为：**using namespace <名字空间名称>**，其中所有的成员可用。多个名字空间成员同名时用作用域运算符限定。

```
#include <iostream>
using namespace std;
namespace ALPHA {
    void g( ) { cout << "ALPHA\n"; }
}
namespace DELTA{
    void g( ) { cout << "DELTA\n"; }
}
using ALPHA::g ; //声明使用特定成员 ALPHA::g( )
int main(void) {
    ALPHA::g( );      //直接访问特定成员 ALPHA 的 g( )
    DELTA::g( );      //直接访问特定成员 DELTA 的 g( )
```

```
    g( );      //默认访问特定成员 ALPHA 的 g( )
        return 0;
```

4、嵌套名字空间：名字空间内可定义名字空间，形成多个层次的作用域，引用时多个作用域运算符自左向右结合。

引用名字空间后，**其内部声明或定义的成员、引用的其它名字空间单个成员、整个名字空间所有成员都能被访问。同名冲突时，用作用域运算符限定名字空间成员。**

在面向过程作用域定义标识符后，访问**名字空间同名标识符时必须用双目作用域运算符限定**，面向过程的标识符必须用单目::限定。

5、**引用名字空间特定成员时**，会将该成员定义加入当前作用域，因此就**不能再定义和该成员同名的标识符**。

引用名字空间与引用名字空间成员不同：

引用名字空间时，**不会将其成员定义加入到当前作用域**，可以在当前作用域定义和名字空间中标识符同名的标识符。

当引用的名字空间成员和函数外定义的变量同名时，用“::变量名”访问外部变量，用“名字空间名称::变量名”访问名字空间成员。

引用名字空间特定成员时，会将该成员定义加入当前作用域，因此就不能再定义和该成员同名的标识符。

```
namespace A {     // A 的初始定义
    int x = 5;
    int f( ) { return   6; }
    namespace B { int y = 8, z = 9; }
     B;
}
using A::x;          //特定名字空间成员 using 声明，不能再定义变量 x
using A::f;          //特定名字空间成员 using 声明，不能再定义函数 f
using namespace A::B; //非特定成员 using，可访问 A::B::y, A::B::z,还可重新定义
int y = 10;          //定义全局变量 y
void main(void) {
    f( );      //调用 A::f( )
    A::f( );       //调用 A::f( )
    ::A::f( );     //调用 A::f( )
    cout<<x + ::y + z + A::B::y; //同一作用域有两个 y,必须区分
//单目:限定了全局变量
}
```

6、可以为**名字空间定义别名**，以代替过长和多层的名字空间名称。对于嵌套定义的名字空间，使用别名可以大大提高程序的可读性。

匿名名字空间的作用域为当前程序文件，名字空间**被自动引用**，其成员定义不加入当前作用域（面向过程或面向名字空间），即可以在当前作用域定义同名成员。一旦同名冲突，**自动引用的匿名名字空间的成员将是不可访问**的。

匿名名字空间也可分多次定义。

程序文件 A.CPP 如下:

```
#include <iostream>
namespace {
//匿名，独立，局限于 A.CPP，
```

```
//不和 B.CPP 的合并
    void  f( ) { cout<<"A.CPP\n"; }
    //必须在名字空间内定义函数体
}
namespace A { int g( ) { return  0; } }
//名字空间 A 将和 B.CPP 合并
int m( ) {
    f( );    return A::g( );
}
```

程序文件 B.CPP 如下：

```
#include <iostream>
namespace A {
    int  g( );
    namespace B {
        namespace C {int  k=4; }
    }
}
namespace ABCD = A::B::C;
//定义别名 ABCD
using ABCD::k;
//引用成员 A::B::C::k
namespace {       //独立的，局限于 B.CPP，不和 A.CPP 的合并
    int  x = 3;    //相当于在本文件定义 static  int x=3
    void  f( ) { std::cout << "B.CPP\n"; }      //必须定义函数体
    class ANT { char c; };
}
int  x = 5;    //定义全局变量
int  z = ::x + k;          //冲突，必须使用::，匿名名字空间 x 永远不能访问
extern int m( );    //声明外部函数
int main(void) {
    ANT  a;
    m( );      f( );
    return A::g( );
}
```

7.3 成员友元

成员友元是一种将一个类的函数成员声明为其它类友元的函数。派生类函数要访问基类私有成员，必须定义为**基类的友元**。

如果某类 A 的所有函数成员都是类 B 的友元，则可以简单的在 B 的定义体内用 friend A 声明，不必列出 A 的所有函数成员。此时称**类 A 为类 B 的友元类**。

友元关系不能传递，即若类 A 是类 B 的友元类，类 B 是类 C 的友元类，此时类 A 并不是类 C 的友元类；

友元关系也不能互换，即类 A 是类 B 的友元类，类 B 并不一定是类 A 的友元类

7.4 普通友元及其注意事项

包括主函数 main 在内，任何普通函数都可以定义为一个类的普通友元。普通友元不是类的

函数成员，故普通友元可在类的任何访问权限下定义。一个普通函数可以定义为多个类的普通友元。

友元函数的**参数也可以缺省和省略。**

普通友元可以访问类的任何数据成员和函数成员。

未声明为当前类友元的函数只能访问当前类的公有成员，声明为当前类友元的函数可以访问类的所有成员。

任何函数的原型声明及其函数定义都可分开，但函数的函数体只能定义一次。在声明普通友元时，也可同时定义函数体(自动内联)。

内联的友员函数的存储类默认为 *static*，作用域局限于*当前代码文件*。全局 main 的作用域为整个程序，故不能在类中内联并定义函数体，否则便会成为局部(即 static)的 main 函数。

//static 是不允许 extern 的,内联函数不一定是 static

7.5 覆盖与隐藏

隐藏是指当基类成员和派生类成员同名时，通过**派生类对象只能访问到派生类成员**，而无法访问到其基类的同名成员。

如果通过**派生类对象还能访问到基类的同名成员**，则称派生类成员覆盖了基类成员。

在一个函数中派生类成员隐藏了基类同名成员，但在另一个函数中可能只是覆盖。

在派生类函数中，基类的**保护和公开成员会被派生类同名函数覆盖**。

```
class BAG    {       //例 7.19
      int    *const e;       //有指针成员 e，浅拷贝容易造成内存泄漏
      const    int    s;
      int p;
public:
      BAG(int m): e(new int[m]), s(e ? m : 0) { p = 0; }
      virtual ~BAG( ) { delete e;    };    //必须自定义析构函数，因为 BAG 有指针成员
      virtual int pute(int f) {
            return p < s ? (e[p++] = f, 1) : 0;
      }   //BAG 允许重复的元素
      virtual int getp( ) { return p; }
      virtual int have(int f) {
            for (int i = 0; i < p; i++)
                  if (e[i] == f) return 1; return 0;
      }
};
class SET : public BAG {    //SET 无数据成员，可直接利用编译为 SET 生成的析构函数
public:
      int pute(int f)              //不允许重复元素：故在 SET::pute()中，必须覆盖 BAG::pute()
      { return have(f) ? 1 : BAG::pute(f); }     //不能去掉 BAG::，否则自递归
      SET(int m): BAG(m) { }
};   //因为 SET 没有数据成员，可直接使用编译程序自动生成的~SET( )，它将自动调用~BAG()
void main( ) {
      SET s(10);
      s.pute(1);
      s.BAG::pute(2);    //在 main()中，BAG::pute()被覆盖，因为它还可被调用
```

```
    s.BAG::getp( );
    int x = s.getp( );    //BAG::getp()被重用，因为没有自定义 SET::getp()函数
    x = s.have(2);        //BAG::have()被重用，因为 SET 没有自定义 have()函数
}
```

在派生类中， using 特定基类数据成员后，**不允许**再在派生类中定义同名数据成员，并且可以通过前述 using 改变或指定新的访问权限。

在派生类中， using 特定基类函数成员后，**还可以**再在派生类中定义同名函数成员，并且可以通过前述 using 改变或指定基类成员继承后的访问权限。

//用 using 来改变基类访问类型

第 8 章 虚函数与多态

8.1 虚函数&虚析构函数

1、虚函数必须是类的成员函数，非成员函数不能说明为虚函数，普通函数如 main 不能说明为虚函数（与编译器有关）。

虚函数一般在基类的 public 或 protected 部分。在派生类中重新定义成员函数时，函数原型必须完全相同；

虚函数只有在具有继承关系的类层次结构中定义才有意义，否则引起额外开销 (需要通过 VFT 访问)；

一般用父类指针(或引用)访问虚函数。根据父类指针所指对象类型的不同，动态绑定相应对象的虚函数；(虚函数的动态多态性)

2、虚函数有隐含的 this 参数，参数表后可出现 const 和 volatile，静态函数成员没有 this 参数，不能定义为虚函数：即不能有 virtual static 之类的说明；

构造函数构造对象的类型是确定的，不需根据类型表现出多态性，故不能定义为虚函数；析构函数可通过父类指针(引用)或 delete 调用，父类指针指向的对象类型可能是不确定的，因此**析构函数可定义为虚函数**。

一旦父类(基类)定义了虚函数，即使没有“virtual”声明，所有派生类中原型相同的非静态成员函数自动成为虚函数；(**虚函数特性的无限传递性**)

```
using namespace std;
struct A{
    virtual void f1( ){ cout<<" A::f1\n"; };     //定义虚函数 f1()
    virtual void f2( ){ cout<<" A::f2\n"; };     //this 指向基类对象，定义虚函数 f2()
    virtual void f3( ){ cout<<" A::f3\n"; };     //定义虚函数 f3()
    virtual void f4( ){ cout<<" A::f4\n"; };     //定义虚函数 f4()
};
class B: public A{    //A 和 B 满足父子关系
    virtual void f1( ){//virtual 可省略，f1()自动成为虚函数
        cout<<" B::f1\n";
    };
    void f2( ) {          //除 this 指向派生类对象外，f2()和基类函数原型相同，自动成为虚函数
        cout<<" B::f2\n";
    };
};
class C: B{          //B 和 C 不满足父子关系，故 A 和 C 也不满足父子关系
    void f4( ) {          //f4()自动成为虚函数
```

```
        cout<<"C::f4\n";
    };
};
void main(void)
{
    C   c;
    A *p=(A *)&c;     //A 和 C 不满足父子关系，需要进行强制类型转换
    p->f1( );       //调用 B::f1( )
    p->f2( );       //调用 B::f2( )
    p->f3( );       //调用 A::f3( )
    p->f4( );       //调用 C::f4( )
    p->A::f2( );        //明确调用实函数 A::f2( )
}
```

如果去掉 A 中 f2 的 virtual,就会输出 A::f2()

2、重载函数使用静态联编（早期绑定）机制；虚函数采用动态联编（晚期绑定）机制； 早期绑定：在程序运行之前的绑定；晚期绑定：在程序运行中，由程序自己完成的绑定。

对于父类 A 中声明的虚函数 f( ), 若在子类 B 中重定义 f( ), 必须确保子类 B::f( )与父类 A::f( )具有完全相同的函数原型，才能覆盖原虚函数 f( )而产生虚特性，执行动态联编机制。否则，只要有一个参数不同，编译系统就认为它是一个全新的（函数名相同时重载）函数，而不实现动态联编。

3、如果基类的析构函数定义为虚析构函数，则派生类的析构函数就会自动成为虚析构函数（原型不同）。

8.3 类的引用

1、用父类引用实现动态多态性时需要注意，若**被(new 产生)引用对象自身不能析构**，则必须用 **delete &**析构：

```
STRING    &z=*new CLERK("zang","982021",23);
delete &z;   //析构对象 z 并释放对象 z 占用的内存
```

2、引用变量引用类的变量、函数参数或者常量，一般不需要引用变量负责构造和析构。由**被引用的类的变量、参数或常量自动完成析构**。

当用常量对象、类型为&&的返回对象作为实参调用函数时，优先调用的函数是带有&&参数的函数。

常量对象既可以被有址变量引用 (分配对象内存) , 也可以被无址变量引用 (分配对象缓存), 但优先被无址形参引用。

```
#include <iostream>
using namespace std;
class A{
    int   i;
public:
    A(int i) { A::i=i; cout<<"A: i="<<i<<"\n"; };
    ~A( ) { if(i) cout<<"~A: i="<<i<<"\n"; i=0; };
};
void g(A &a) {cout<< "g is running\n"; } //调用时初始化形参 a
void h(A &&a=A(5)) {cout<< "h is running\n"; }   //调用时初始化形参 a，A(5)为默认值
```

```
void main(void)
{
    A a(1), b(2);              //自动调用构造函数构造 a、b
    A &p=a;            //p 本身不用负责构造和析构 a
    A &q=*new A(3);        //q 有址引用 new 生成的无名对象
    A &r=p;            //r 有址引用 p 所引用的对象 a
    cout<<"CALL g(b)\n";
    g(b);                      //使用同类型的传统左值作为实参调用函数 g()
    h( );              //使用无址右值 A(5)作为实参调用 h()，初始化 h()的形参 a
    h(A(4));           //使用无址右值 A(4)作为实参调用 h()，初始化 h()的形参 a
    cout<<"main return\n";
    delete &q;                 //q 析构并释放通过 new 产生的对象 A(3)
}
```

//a 和 b 不用管

3、浅拷贝与深拷贝之不同

按照字节拷贝和直接赋地址

8.4 抽象类

1、纯虚函数：不必定义函数体的虚函数，也可以重载、缺省参数、省略参数、内联等，相当于 Java 的 interface。

定义格式：virtual 函数原型=0。 (0 即函数体为空)

纯虚函数有 this，不能同时用 static 定义(表示无 this)。

构造函数不能定义为虚函数，同样也不能定义为纯虚函数。

析构函数可以定义为虚函数，也可定义为纯虚函数。

函数体定义应在派生类中实现，成为非纯虚函数。

2、**抽象类：含有纯虚函数的类。**

如果派生类继承了抽象类的纯虚函数，却没有在派生类中重新定义该原型虚函数，或者派生类定义了基类所没有的纯虚函数，则派生类就会自动成为抽象类。

在多级派生的过程中，如果到某个派生类为止，所有纯虚函数都已在派生类中全部重新定义，则该派生类就会成为非抽象类（具体类）。

```
#include <iostream>
using namespace std;
struct A{   //A 被定义为抽象类
    virtual void f1( )=0;
    virtual void f2( )=0;
};
void A::f1( ){ cout<<“A1”; }//不是在派
void A::f2( ){ cout<<"A2"; }//生类中定义
class B: public A{
    //重新定义 f2，未定义 f1，B 为抽象类
    void f2( ) { this->A::f2( ); cout<<"B2"; }
};
class C: public B{// f1 和 f2 均重定义，C 为具体类
    void f1( ) { cout<<“C1”; }//自动成虚函数，但内联失败
};
```

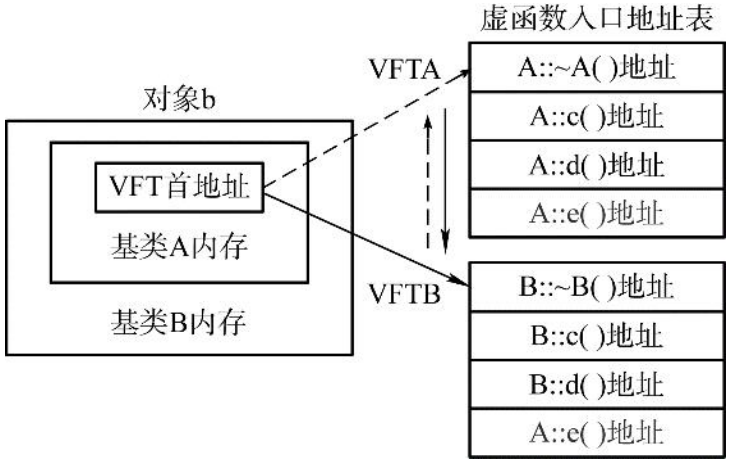
抽象类不能定义或产生任何对象，包括用 new 创建的对象，故不能用作函数参数的类型和函数的返回类型（调用前后要产生该类型的对象）。
抽象类可作派生类的基类(父类)，若定义相应的基类引用和指针，就可引用或指向非抽象派生类对象。

```
#include <iostream>
using namespace std;
struct A{
//定义类 A 为抽象类
      virtual void f1( )=0;
      void f2( ) { };
};
struct B: A{
//定义 A 的非抽象子类 B
      void f1( ){ };
};
A   f( );   //×，返回类 A 意味着抽
                  //象类要产生 A 类对象
int g(A   x); //  ×，调用时要传递
           //一个 A 类的对象
A &h(A &y); //√，可以引用非
                //抽象子类 B 的对象
void main(void)
{
      A a;        //×，抽象类不能产生
        //对象 a
      A *p;     //√，可以指向非抽象
        //子类 B 的对象
      p->f1( ); //×，运行时无 A::f1( )
                    //如 p 指向 B 类对象则正确
      p->f2( ); //√，调用 A::f2( )
}
```

8.5 虚函数友元与晚期绑定

虚函数动态绑定：
C++使用虚函数地址表(VFT)来实现虚函数的动态绑定。VFT 是一个函数指针列表，存放对象的所有虚函数的入口地址。
编译程序为有虚函数的类创建一个 VFT，其首地址通常存放在对象的起始单元中。调用虚函数的对象通过起始单元☾VFT 动态绑定相应的函数成员，从而使虚函数随调用对象的不同而表现多态特性。
虚函数动态绑定过程：设基类 A 和派生类 B 对应的虚函数表分别为 VFTA 和 VFTB。则派生类对象 b 的虚函数动态绑定过程如下：
对象构造：先将 VFTA 的首地址存放到 b 的起始单元，在 A 类构造函数的函数体执行前甚至初试化前，使 A 类对象调用的虚函数与 VFTA 绑定，可使 A 类构造函数执行 A 的虚函数；在 B 类构造函数的函数体执行前（甚至初试化前），将 VFTB 的首地址存放到 b 的起始单

元，使 B 类对象调用的虚函数与 VFTB 绑定，可使 B 类构造函数执行 B 的虚函数。
对象使用(生成期间)：b 的起始单元指向 VFTB，执行 B 的虚函数。
对象析构：由于 b 的起始单元已指向 VFTB，故析构函数调用的是 B 的虚函数；然后将 VFTA 的首地址存放到 b 的起始单元，使基类析构函数调用的虚函数与 VFTA 绑定，使基类析构函数调用基类 A 的虚函数。



8.6 有虚函数时的内存布局

派生类的存储空间由基类和派生类的非静态数据成员构成。当基类或派生类包含虚函数或纯虚函数时，派生类的存储空间还包括虚函数入口地址表首址所占存储单元。
如果基类定义了虚函数或者纯虚函数，则派生类对象将其起始单元作为共享单元，用于存放基类和派生类的虚函数地址表首址。
如果基类没有定义虚函数，而派生类定义了虚函数，则派生类的存储空间由**三部分**组成：**第一部分为基类存储空间，第二部分为派生类虚函数入口地址表首址，第三部分为该派生类新定义的数据成员**

第 9 章 多继承与虚基类

9.1 多继承类

多继承派生类有多个基类或虚基类。
派生类继承**所有**基类的数据成员和成员函数。
派生类在继承基类时，不同的基类可以采用**不同的派生控制**。
基类之间的成员可能同名，**基类与派生类的成员也可能同名**。在出现同名时，如面向对象的作用域不能解析，应该使用**作用域运算符**来指明所要访问的类的成员。
多继承派生类的定义：

```
class 派生类名:<派生方式> 基类 1,<派生方式> 基类 2,…{
    <类体>
};
```

同样存在派生类对象多次初始化同一(物理)基类对象问题。派生类对象多次初始化同一基类成员问题（多次闪烁）：假设类 Window、HScrollbar、VScrollbar 都是从基类 Port 派生，即：

```
class Port{…};
class Window:public Port{…};
class HScrollbar:public Port{…};      class VScrollbar:public Port{…};
class ScrollableWind:public Window,public HScrollbar,public VScrollbar{…};
```

这个时候我们就需要虚基类来帮忙

9.2 虚基类

同一个类不能多次作为某个派生类的直接基类，但可多次作为其间接基类，从而引起存储空间的浪费和其他问题。此时，这些间接基类可定义为**虚基类**。

同一颗派生树中的同名虚基类，共享同一个存储空间；其**构造函数和析构函数仅执行 1 次**，且**构造函数尽可能最早执行**，而析构函数尽可能最晚执行。

如果**虚基类与基类同名**，则它们将分别拥有**各自的存储空间**，只有同名虚基类才共享存储空间，而同名基类则拥有各自的存储空间。

虚基类和基类同名必然会导致二义性访问，编译程序会对这种二义性访问提出警告。当出现这种情况时，建议：要么将**基类说明为对象成员，要么将基类都说明为虚基类。可用作用域运算符限定要访问的成员**。

```
#include <iostream>
using namespace std;
struct A{
   int a;
   A(int x){ a=x; }
};
struct B: A{//等于struct B:public A
   B(int x):A(x){ }
};
struct C{
   C( ){ }
};
struct D: virtual A, C{
   D(int x):A(x)  { } //同样调用C( )
};
struct E: B, D{
   E(int x):A(x), B(x+5), D(x+10){ }
};
```

```
void main(void){
   E  e(0);
 //cout<<"a="<<e.a; //二义性访问
   cout<<"a="<<e.B::a;
   cout<<"a="<<e.D::a;
}
```

为解决e.a产生的二义性，要么将E的基类B说明为对象成员，要么将B的基类A说明为虚基类。若将B的基类A说明为虚基类，则e.a、e.B::a及e.D::a都表示虚基类A的成员a 。

9.3 派生类成员

当派生类有多个基类或虚基类时，基类或虚基类的成员之间可能出现同名；派生类和基类或虚基类的成员之间也可能出现同名。

出现上述同名问题时，必须通过**面向对象的作用域解析**，或者用作用域运算符::指定要访问的成员，否则就会引起二义性问题。

9.4 单重及多重继承的构造与析构

1、在考虑多继承派生类构造函数的执行顺序时，必须注意派生类可能有虚基类、基类、对象成员、const 成员以及引用成员。当**虚基类、基类和对象成员只有带参数的构造函数**时，派生类必须**定义自己的构造函数**，而不能利用 C++提供的缺省构造函数。类有**非静态对象成员、const 成员时，也必须定义构造函数**。

对于虚基类、基类和对象成员的**无参构造函数**，无论它们是自定义的还是由编译程序提供的，可被派生类构造函数按**定义顺序自动地调用**。

派生类对象的构造顺序描述：

按定义顺序自左至右、自上而下地构造所有虚基类；

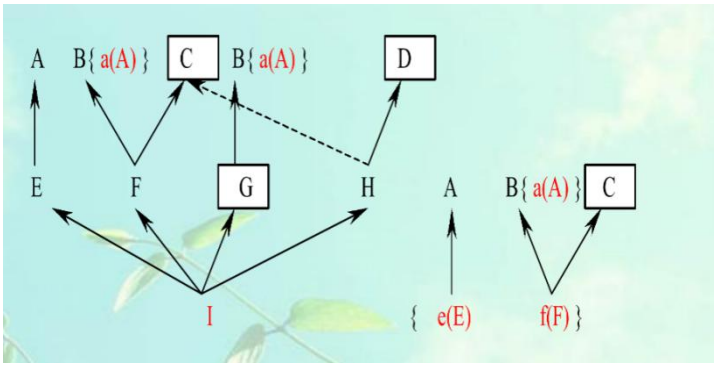
按定义顺序构造派生类的所有直接基类；

按定义顺序构造（初始化）派生类的所有数据成员，包括对象成员、const 成员和引用成员；执行派生类自身的构造函数体；

如果虚基类、基类、对象成员、const 成员以及引用成员又是派生类对象，重复上述派生类对象的构造过程，但同名虚基类对象在同一棵派生树中仅构造一次。

析构派生类对象的顺序同构造逆序。

```
#include <iostream>
using namespace std;
struct A{ A( ) { cout<<'A';} };
struct B { const A a;   B( ) { cout<<'B';} };
struct C{ C( ) { cout<<'C';} };
struct D{ D( ) { cout<<'D';} };
struct E：  A{ E( ) { cout<<'E';} };
struct F：  B, virtual C{ F( ) { cout<<'F';} };
struct G：  B{ G( ) { cout<<'G';} };
struct H：  virtual C, virtual D
{ H( ) { cout<<'H';} };
struct I：  E, F, virtual G, H{
      E   e;
      F   f;
      I( ):f( ), e( ) , F( ), E( ){ cout<<'I';}
};
void main(void) { I   i; }
```



输出:CABGDAEABFHAECABFI

```
#include <iostream>
using namespace std;
struct A{ A( ) { cout<<'A';} };
struct B { const A a;  B( ) { cout<<'B';} };
struct C{ C( ) { cout<<'C';} };
struct D{ D( ) { cout<<'D';} };
struct E:A{ E( ) { cout<<'E';} };
struct F:B, virtual C{ F( ) { cout<<'F';} };
struct G:B{ G( ) { cout<<'G';} };
struct H:virtual C, virtual D
{ H( ) { cout<<'H';} };
struct I:E, F, H,virtual G{
    E  e;
    F  f;
    I( ):f( ), e( ) , F( ), E( ){ cout<<'I';}
};
int main(void) { I  i; }
```

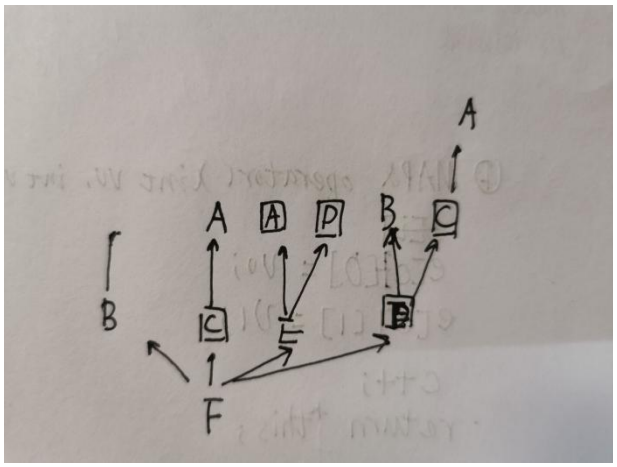
输出:**CDABGAEABFHAECABFI**

解析:先自左向右调用各虚基类的构造函数,定义多继承时,虚基类的定义顺序不同会导致其左右的顺序不同,判断左右的方式就是下决定上,比如说 C 是第二个和第四个继承类里的虚基类,D 是第三个继承类的虚基类,那就 C 在 D 左边,接着按照定义顺序构造,注意,虚基类的构造函数就不需要再次调用了!

先调用虚基类的构造,这时候构造就是嵌套的,如果虚基类里面还有虚基类,那就跳过不定义但是如果是变量内的定义就不跳过

```
struct A { A( ) { cout << 'A'; } };
```

```
struct B { B( ) { cout << 'B'; } };
struct C: A { C( ) { cout << 'C'; } };
struct D: B, virtual C { D( ) { cout << 'D'; } };
struct E: virtual A, virtual D {
D  d;
E( ): A( ) { cout << 'E'; }
};
struct F: B, virtual C, E, virtual D {
D  d;
F( ): A( ) { cout << 'F'; }
};
```



构建方式:,从左到右,从上到下对虚基类进行构造,首先是对 C 进行构造,进入 C 的构造,输出 AC

在进入 A 和 D 的构造,A 的就输出 A

D 的构造:先构造 B 和 C,因为 C 之前先构造了,那么就不管了,就直接输出 BD

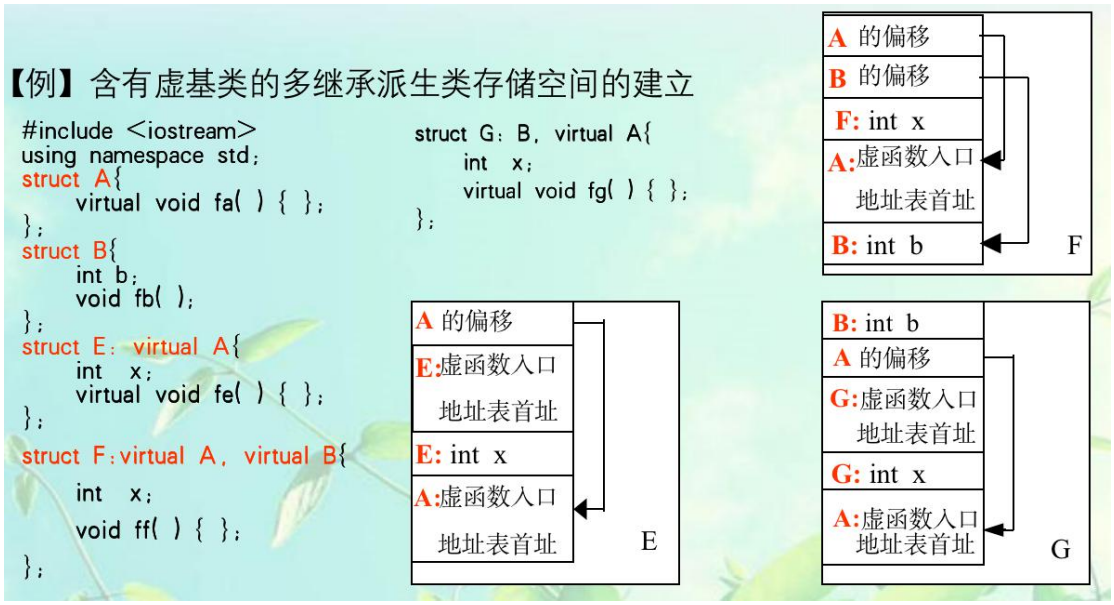
最后一个虚基类构造完了,就直接开始构造直接基类,B 和 E,B 的话就直接输出 B,E 的话,由于 A 和 D 都是虚基类,就直接构造一个新的 D,输出 ACBD,接着输出 E,最后进行 F 的构造输出 ACBD,这个和递归差不多,我们求出了每一个小类的输出顺序,就调用的时候考虑一下虚基类有没有被生成过即可

顺序:1.**多继承的类(虚基类只调用一次,从左到右,从上到下)2.成员 3.自身的构造函数**

4.类的构造是递归实现的,对于虚基类,构造了一次就不会构造第二次,包括自身和其基类,在参数表里构造的情况下构造普通类不会构造虚基类

9.5 类的存储空间

总之是虚基类在最后面就对了,不过要存一个成员指针来取地址



第 10 章 异常与断言

10.1 异常处理

1、异常：一种意外破坏程序正常处理流程的事件、由硬件或者软件触发的事件。
异常处理可以将错误处理流程同正常业务处理流程分离, 从而使程序的正常业务处理流程更加清晰顺畅。
异常发生后自动析构调用链中的所有对象，这也使程序降低了内存泄漏的风险。
由软件引发的异常用 throw 语句抛出，会在抛出点建立一个描述异常的对象，由 catch 捕获相应类型的异常。
在 catch(参数)中，**不能出现有址引用或无址引用类型的参数**。
thow：用于引发异常、继续传播异常
catch：异常捕获部分处理时，根据要捕获的异常对象类型处理相应异常事件，可以继续传播或引发新的异常。
一旦异常被某个 catch 处理过程捕获，则**其它所有处理过程都会被忽略**。
2、异常处理过程必须定义且只能定义一个参数，该参数必须是省略参数或者是类型确定的参数 ，因此，异常处理过程**不能定义 void 类型的参数**。
如果是通过 new 产生的指针类型的异常, 在 catch 处理过程捕获后, 通常应使用合适的 delete 释放内存，否则可能造成内存泄漏。
如果继续传播指针类型的异常，则可以不使用 delete。
抛出字符串常量如“abc”的异常需要用 catch(**const char *p**)捕获，处理异常完毕可以不用 delete p 释放，因为字符串常量通常存储在数据段。
没有任何实参的 throw 用于传播已经捕获的异常。
Try-------没有异常:继续,遇到 throw 就退出进入 catch,try 后面的所有语句都被忽略

10.2 捕获顺序

1、允许函数模板和模板实例函数定义异常接口
先声明的异常处理过程将先得到执行机会, 因此, 可将需要先执行的异常处理过程放在前面。
如果父类 A 的子类为 B，B 类异常也能被 catch(A)、catch(const A)、 catch(volatile A)、catch(const volatile A) 等捕获。
如果父类 A 的子类为 B, 指向可写 B 类对象的指针异常也能被 catch(A*)、catch(const A*)、catch(volatile A*)、 catch(const volatile A*)等捕获。
捕获子类对象的 catch 应放在捕获父类对象的 catch 前面。
注意 **catch(const volatile void *)能捕获任意指针类型的异常，catch(…)能捕获任意类型的异常**。
注意是退出 try 之后,catch 的进入顺序是谁先定义就先进谁的,char*和 const char*都一样的

10. 3 函数的异常接口

异常接口**不是函数原型的一部分**，不能通过异常接口来定义和区分重载函数，故其不影响函数内联、重载、缺省和省略参数
不引发任何异常的函数引发的异常、引发了未说明的异常称为不可意料的异常。
noexcept 可以表示 throw()或 throw(void)。
noexcept 一般用在移动构造函数，析构函数、移动赋值运算符函数等**肯定不会出现异常的函数后面**。
如果移动构造函数和移动赋值运算符还要申请内存之外的资源，则难免发生异常，此时不应将 noexcept 放在这些函数的参数标后面。
保留字 noexcept 和 throw( )可以出现在任何函数的后面，包括 **constexpr 函数和 Lambda 表达式**的参数表后面。但 throw(除 void 外的类型参数)不应出现在 constexpr 函数的参数表后

面，并且 constexpr 函数也不能抛出异常，否则不能优化生成常量表达式。

```
#include <iostream>
using namespace std;
//以下函数 sum()不会处理它发出的 const char *类型的异常
int sum(int a[ ], int t, int s, int c) throw (const char *)
{    //以下语句若发出 const char *类型的异常，此后的语句不执行
     if (s < 0 || s >= t || s + c<0 || s + c>t)
          throw "subscription overflow";
     int r = 0,x=0;
     for (x = 0; x < c; x++)
          r += a[s+x];
     return r;
}
void main( )
{
     int m[6]={1,2,3,4,5,6};
     int r=0;
     try{
          r=sum(m, 6, 3, 4);//发出异常后 try 中所有语句都不执行，直接到其 catch
          r=sum(m, 6, 1, 3);//不发出异常
     }
     //以下 const 去掉则不能捕获 const char *类型的异常，只读指针实参不能传递给可写指针形参 e
     catch(char *p){cout<<p; }  //不能捕获 throw "subscription overflow";
     catch(const char *e){  //还能捕获 char *类型的异常，可写指针实参可以传递给只读指针形参 e
          cout<<e;
     }//由于 throw 时未分配内存，故在 catch 中无须使用 delete  e
}
```

10.4 异常类型

1、C++提供了一个标准的异常类型 exception，以作为标准类库引发的异常类型的基类，exception 等异常由标准名字空间 std 提供。

exception 的函数成员不再引发任何异常。

函数成员 what( )返回一个只读字符串，**该字符串的值没有被构造函数初始化，因此必须在派生类中重新定义函数成员 what( )。**

异常类 exception 提供了处理异常的标准框架，**应用程序自定义的异常对象应当自 exception 继承。**

在 catch 有父子关系的多个异常对象时，应注意 catch 顺序。

10.5 异常对象的析构

从最内层被调函数抛出异常到外层调用函数的 catch 处理过程捕获异常，由此形成的函数调用链所有局部对象都会被自动析构，因此使用异常处理机制能在一定程度上防止内存泄漏。但是，调用链中的指针通过 new 分配的内存不会自动释放。

```
#include <exception>
#include <iostream>
```

```
using namespace std;
class EPISTLE : exception {    //定义异常对象的类型
public:
    EPISTLE(const char* s)  { cout<<"Construct: " << s; }
    ~EPISTLE( )noexcept { cout << "Destruct: " << exception::what( )<<endl; };
    const char* what( )const throw( ) { return exception::what( ); };
};
void h( ) {
    EPISTLE h("I am in h( )\n");
    throw new EPISTLE("I have throw an exception\n");
}
void g( ) {   EPISTLE g("I am in g( )\n");  h( );  }
void f( ) {    EPISTLE f("I am in f( )\n");      g( );  }
int main(void) {
    try {      f( );    }
    catch (const EPISTLE * m) {
        cout << m->what( )<<endl;
        delete m;
    }
}
```

```
Construct: I am in f( )
Construct: I am in g( )
Construct: I am in h( )
Construct: I have throw an exception
Destruct: std::exception
Destruct: std::exception
Destruct: std::exception
std::exception
Destruct: std::exception
```

main()->f()->g()->h()->h 抛出异常(指针)->局部对象 h、g、f 依次析构->main 捕获异常并 delete

10.6 断言

函数 assert(int)在 assert.h 中定义。

断言（assert）是一个带有整型参数的用于调试程序的函数，如果实参的值为真则程序继续执行。

否则，将输出断言表达式、断言所在代码文件名称以及断言所在程序的行号，然后调用 **abort()** 终止程序的执行。(这有可能导致内存泄漏)

断言输出的代码文件名称包含路径（编译时值），运行时程序拷到其它目录也还是按原有路径输出代码文件名称。assert()在运行时检查断言。

保留字 static_assert 定义的断言在编译时检查，为真时不终止编译运行。

第 11 章 运算符重载

11.1 运算符概述

1、C++规定运算符重载必须针对类的对象，即重载时**至少有一个参数代表对象**(类型如 A、const A、A&、const A&、volatile A 等）。

C++用 operator 加运算符进行运算符重载。对于普通运算符成员函数，**this 隐含参数代表第一个操作数对象**。
根据能否重载及重载函数的类型，运算符分为：
不能重载的：sizeof、．、.*、::、?:
只能重载为普通函数成员的：=、–>、( )、[ ]
不能重载为普通函数成员的：new、delete
其他运算符：都不能重载为静态函数成员，但可以重载为普通函数成员和普通函数。

```
class A;
int operator=(int, A&); //错误，不能重载为普通函数
A& operator +=(A&,A&);//A*和 A[ ]参数不代表对象
class A{
        friend int operator=(int,A&); //错误，不存在 operator=
        static int operator( )(A&,int); //错误，不能为静态成员
        static int operator+(A&,int);   //错误，不能为静态成员
        friend A& operator += (A&,A&);   //正确
        A& operator ++( );      //隐含参数 this 代表一个对象
};
void main(void)
{
    A a(6);                 //调用 A(int)时，实参&a 传递给隐含形参 this
    cout<<"a+7="<<a+7<<"\n";  //调用 int operator+(const A&, int)
    cout<<"a+7="<< operator+(a,7)<<"\n"; //调用 int operator+(const A&, int)
    cout<<"8+a="<< operator+(8,a)<<"\n"; //调用 int operator+(int, A)
    cout<<"8+a="<<8+a<<"\n";  //调用 int operator+(int, A)
}
```

11.2 运算符参数
1、重载函数种类不同，参数表列出的参数个数也不同。
重载为普通函数：参数个数=运算符目数
重载为普通成员：参数个数=运算符目数 - 1 (即 this 指针)
特殊运算符不满足上述关系：->双目重载为单目，前置++和--重载为单目，后置++和--重载为双目、函数( )可重载为任意目。
( )表示强制类型转换时为单参数；表示函数时可为任意个参数。
2、运算符++和--都会改变当前对象的值，重载时最好将参数定义为**非只读引用类型(左值)**，左值形参在函数返回时能使实参带出执行结果。前置运算是先运算再取值，后置运算是先取值再运算。
后置运算应重载为**返回右值的双目运算符函数**：
如果重载为类的普通函数成员，则该函数只需定义一个 int 类型的参数(已包含一个不用 const 修饰的 this 参数)；
如果重载为普通函数(C 函数)，则最好声明非 const 引用类型和 int 类型的两个参数(无 this 参数)。
前置运算应重载为返回左值的单目运算符函数：
前置运算结果应为左值，其返回类型应该定义为非只读类型的引用类型；左值运算结果可继续++或--运算。
如果重载为普通函数(C 函数)，则最好声明非 const 引用类型一个参数 (无 this 参数)。

```
class A{
    int  a;
    friend A &operator--(A&x){x.a--; return x; }//自动内联，返回左值
    friend A operator--(A&, int); //后置运算，返回右值
public:
    A &operator++( ){ a++;   return *this; }//单目，前置运算
    A operator++(int){ return  A(a++); }//双目，后置运算
    A(int x) { a=x; }
};//A m(3); (--m)--可以；因为--m 左值，其后--要求左值操作数
A operator--(A&x, int){ //x 左值引用，实参被修改
    return  A(x.a--);  //先取 x.a 返回 A(x.a)右值，再 x.a--
} //A m(3); (m--)--不可；因为 m--右值，其后--要求左值操作数
A 右 A& 左 加 const 右
int i=b->a;    //自动重载为 b->->a
```

11.3 赋值与调用

1、编译程序为每个类提供了**缺省赋值运算符函数**，对类 A 而言，其成员函数原型为 A&operator=(const A&)。

如果类自定义或重载了赋值运算函数，则优先调用类自定义或重载的赋值运算函数(不管是否取代型定义)。

编译器给定的重载函数是浅拷贝赋值的,如果类内没有指针的话就可以使用

浅拷贝：位拷贝，拷贝构造函数，赋值重载

多个对象共用同一块资源，同一块资源释放多次，崩溃或者内存泄露

深拷贝：每个对象共同拥有自己的资源，必须显式提供拷贝构造函数和赋值运算符。

（1）应定义“T(const T &)”形式的深拷贝构造函数；

（2）应定义“T(T &&) noexcept”形式的移动构造函数；

（3）应定义“virtual T &operator=(const T &)”形式的深拷贝赋值运算符；

（4）应定义“virtual T &operator=(T &&) noexcept”形式的移动赋值运算符；

（5）应定义“virtual ~T( )”形式的虚析构函数；

（6）在定义引用“T &p=*new T( )”后，要用“delete &p”删除对象；

（7）在定义指针“T *p=new T( )”后，要用“delete p”删除对象；

（8）对于形如“T a; T&&f( );”的定义，不要使用“T &&b=f( );”之类的声明和“a=f( );”

（9）不要随便使用 exit 和 abort 退出程序。

（10）最好使用异常处理机制。

11.4 强制类型转换

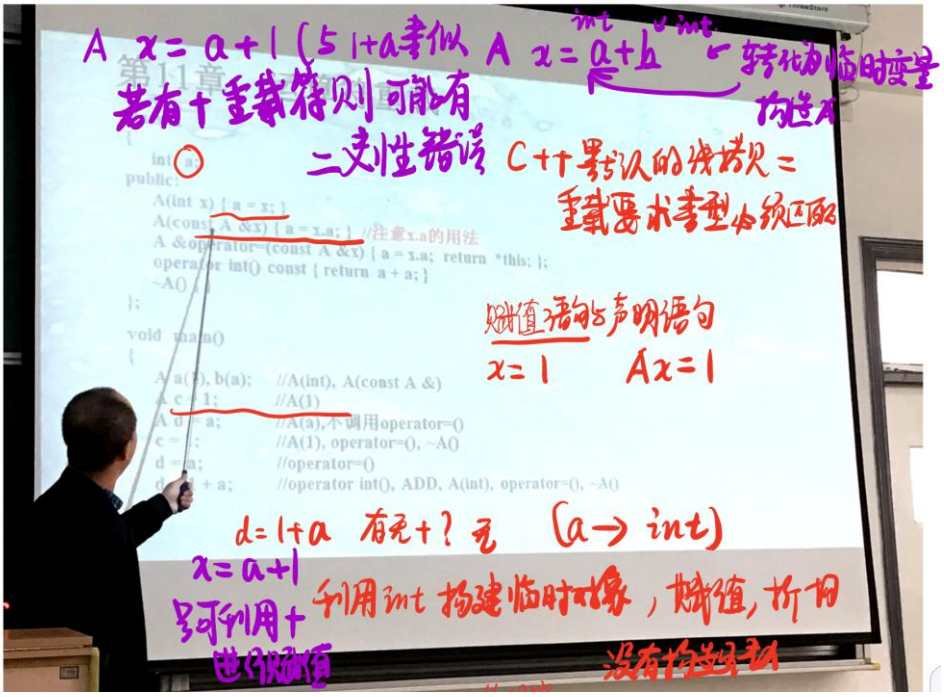
单参数的构造函数具备类型转换作用，必要时能自动将参数类型的值转换为要构造的类型。

以下通过定义单参数构造函数简化重载（同时注意 C++会自动将 int 转为 double）：

```
    class COMPLEX{
    double r, v;
    public:
    COMPLEX(double  r1);
    COMPLEX(double  r1, double v1){ r=r1; v=v1; }
    COMPLEX operator+(const COMPLEX &c)const;
    COMPLEX operator-(const COMPLEX &c)const;
```

所以说,m+2.0 可以转换成 m+COMPLEX(2.0)

所以说有两种构造函数:
单参数的构造函数相当于类型转换函数, 单参数的 T::T(const A) T::T(A&&) 、T::T(const A&)等相当于 A 类到 T 类的强制转换函数。
也可以用 operator 定义强制类型转换函数。由于转换后的类型就是函数的返回类型，所以强制类型转换函数不需要定义返回类型。
不应该同时定义 A::operator T 和 T::T(const A&)，否则容易出现二义性错误。
按照 C++约定，类型转换的结果通常为右值，故最好不要将类型转换函数的返回值定义为左值，也不应该修改当前被转换的对象 **(参数表后用 const 说明 this)。**



第 12 章 类型解析,转换与推导

12.1 隐式与显式类型转换

1、隐式转换:以前有给出
2、有关类型转换若有警告，则应修改为强制类型转换即显式类型转换。
强制类型转换引起的问题由程序员自己负责。

```
char   u = 'a';        //编译时可计算，无截断，不报警
char   v = 'a' + 1;        // 'a' + 1  没有超过 char 的范围，不报警
char   v = 'a' + 100;      // 'a' + 100  超过 char 的范围（v=-59），报警，不截断
char   w = 300;            // 300 超过 char 的范围（w=44），报警，截断
int      x = 2;       //x 占用的字节数比 char 和 short 类型多，不报警
char   y = x;         //编译时不可计算，可能截断，要报错
short z = x;          //编译时不可计算，可能截断，要报错
```

3、一般简单类型之间的强制类型转换的结果为右值。
如果对可写变量进行同类型的左值引用转换，则转换结果为左值。
只读的简单类型变量如果转换为**可写左值**，并不能修改其值 (受到页面保护机制的保护)。

编译能通过,但是不能达到改动值的要求

对于类的只读数据成员，如果转换为可写左值，可以修改其值。

目前操作系统并不支持分层保护机制，无法在对象层和数据成员层提供不同类型的保护。

```
void main(int argc, char *argv[ ]) {
    const int x = 0;
    *(int *)&x = 2;            //debug 时 x 的值变为 2
    cout << "x=" << x<< endl;   //但输出结果 x=0 不变
```

12.2 cast 类型转换

1、static_cast 静态转换

使用格式为“static_cast<T> (expr)”，用于将数值表达式 expr 的源类型转换为 T 目标类型。

目标类型**不能包含存储位置类修饰符**，如 static、extern、auto、register 等。

static_cast 仅在编译时静态检查源类型能否转换为 T 类型，运行时不做动态类型检查。

static_cast **不能去除源类型的 const 或 volatile**。即不能将指向 const 或 volatile 实体的指针(或引用)转换为指向非 const 或 volatile 实体的指针(或引用)。

要注意转换出来的是左值还是右值,这里有个顺口溜

这个和 C 里面的强制类型转换没有区别

不带&右,带&左,加了个 const 强制右,都可以读,左可以写

2、const_cast——只读转换

const_cast 的使用格式为 “const_cast<类型表达式>(数值表达式)”。

修改类型的 const 和 volatile 属性，只能转换为指针、引用或指向对象成员的指针类型。

不能用 const_cast 将无址常量、位段访问、无址返回值转换为有址引用。

```
int ww = *const_cast<int *>(&xx) = 2;
```

//不能改变受保护的 xx (只读简单类型)，ww=2， xx=0,在这个语句里面可以读成 2,但是 xx 本质上是不会改变的

其他的和 static_cast 差不多

3、dynamic_cast——动态转换

dynamic_cast 在运行时转换：**子类向父类转换，以及有虚函数的基类向派生类转换。被转换的表达式必须涉及类类型。没有虚函数的基类向派生类转换会报错**

使用格式为 “dynamic_cast<T> (expr)”, 要求类型 T 是类的引用、类的指针或者 void * 类型，而 expr 的类型 必须是 **类的对象（常量或变量：向引用类型转换）、父类或者子类的引用或指针。**

dynamic_cast 转换时不能去除数值表达式 expr 源类型中的 const 和 volitale 属性。

有址引用和无址引用之间不能相互转换。

将基类转换为派生类时，基类必须包含虚函数或纯虚函数。最好先用 typeid 检查确保基类对象实际上就是派生类对象。

```
#include <iostream>
using namespace std;
struct B {
    int m;
    B(int x): m(x) { }
    virtual void f( ) { cout << 'B'; }     //若无虚函数，“dynamic_cast<D *>(&b)”向下转换出错
};
struct D : public B {              //B 是父类，D 是子类
    int n;
```

```
    D(int x, int y): B(x), n(y) { }
    void f( ) { cout << 'D'; }        //函数 f()自动成为虚函数
};
void main( ) {
    B a(3);
    B &b = a;
    D c(5,7);
    D &d = c;
    D *pc1 = static_cast<D *>(&a);          //语法正确但为不安全的自上向下转换
    pc1->f( );                  //输出 B，若去除 A:f()前面的 virtual，结果怎样？就输出 D
    D *pc2 = static_cast<D *>(&b);          //语法正确但为不安全的自上向下转换
    pc2->f( );                  //输出 B
    D *pc3 = dynamic_cast<D *>(&a);   //若 a 无虚函数 f( )，则转换错误，a 不是 D 的对象
导致 pc3=0
    pc3->f( );                  //运行异常：pc3 为 nullptr（a 非子类对象）
    D *pc4 = dynamic_cast<D *>(&b);   //若 b 无虚函数 f( )，则自上向下转换错误
    pc4->f( );                  //运行异常：pc4 为空指针（b 非子类对象）
    B *pb1 = dynamic_cast<D *>(&c);    //语法正确且为安全的自下向上赋值
    pb1->f( );                  //输出 D：正确的多态行为
    B *pb2 = dynamic_cast<D *>(&d);   //语法正确且为安全的自下向上赋值
    pb2->f( );                  //输出 D：正确的多态行为
```

上述的内容改成 D&和 D&&都是相同的,**注意不能在&和&&之间转换,要转换请看 4!**

子类实例取地址可以赋值给基类指针

但是基类实例取地址赋值给子类指针是不允许的

4、reinterpret_cast——重释转换

reinterpret_cast <T> (expr)，将数值表达式 expr 转换成不同性质的其他类型 T。T 类型不能是实例数据成员指针。T 可以是指针、引用、或其他与 expr 完全不同的类型。

将指针转换为足够大的整数，整数类型必须够存储一个地址。X86 和 X64 的指针大小不同，X86 使用 int 类型即可。

当 T 为使用&或&&定义的引用类型时，expr 必须是一个有址表达式。

有址引用和无址引用之间可以相互转换。

甚至指针可以和 int 互相转换

12.3 类型转换实例

C++的父类指针（或引用）可以直接指向（或引用）子类对象，但是通过父类指针（或引用）只能调用父类定义的成员函数。

武断或盲目地向下转换，然后访问派生类或子类成员，会引起一系列安全问题: (1)成员访问越界(如父类无子类的成员); (2)函数不存在(如父类无子类函数) 。

关键字 typeid 可以获得对象的真实类型标识：有 ==、!=、before、raw_name、hash_code 等函数。

typeid 使用格式：（1）typeid(类型表达式)；（2）typeid(数值表达式)。

typeid 的返回结果是 **const type_info(一个类来的,用于表征类型) & 类型**，在使用 typeid 之前先#include <typeinfo>。

```
int main(int argc, char *argv[ ]) {
    B a(3);                         //定义父类对象 a
```

```
B &b = a;
D c(5, 7);                          //定义子类对象 c
D &d = c;
B *pb = &a;                         //定义父类指针 pb 指向父类对象 a
D *pc(nullptr);                     //定义子类指针 pc 并设为空指针
if (argc < 2) pb = &c;
if (typeid(*pb) == typeid(D)) {     //判断父类指针是否指向子类对象
    pc = (D *)pb;                   //C 的强制转换，子类指针指向父类-安全
    pc = static_cast<D *>(pb);      //静态强制转换，安全，因为 pb 指向 D 类
    pc = dynamic_cast<D *>(pb);     //动态强制转换：向下转换 B 须有虚函数
    pc = reinterpret_cast<D *>(pb); //重释类型转换，安全，因为 pb 指向 D 类
    pc->g( );                       //输出 G，不转换 pb 无法调用 g( )
}
cout << typeid(pc).name( )<<endl;   //输出 struct D *
cout << typeid(*pc).name( ) << endl;//输出 struct D
cout << typeid(B).before(typeid(D))<<endl; //输出 1 即布尔值真：B 是 D 的基类
```

保留字 explicit 只能用于定义构造函数或类型转换实例函数， explicit 定义的实例函数成员必须显式调用。

未用 explicit 定义前:

(1) double d = m 等价于 d = m.operator double( )

(2) m+2.0 等价于 m+ COMPLEX(2.0, 0.0)

使用 explicit 定义后:

(1) 不能定义 d = m;

(2) 不能用 m + 2.0 相加。

只能：

double d = m.operator double( );

COMPLEX a = m + COMPLEX(2.0, 0.0);

12.4 自动类型推导

保留字 auto 在 C++中用于类型推导。

可用于推导变量、各种函数的返回值、以及类中用 const 定义的静态数据成员的类型。

使用 auto 推导时, 被推导实体不能出现类型说明, 但是可以出现存储可变特性 const、voilatile 和存储位置特性如 static、register。

保留字 auto 可以推导与数组和函数相关的类型。

使用数组名代表整个数组类型，使用函数名代表该函数的指针。

使用“数组名[表达式]”表示除第一维外的数组类型，依此类推。

当被推导变量前面有“*”时，数组类型的第 1 维（仅用数组名）或者剩下维（使用“数组名[表达式]”）的第 1 维优先被解释为指针。

无论被推导变量前面有无“*”，函数名总是解释为指针。

```
int       a[10][20];                //a 的类型为 int [10][20]，可理解为“int (*a)[20]; ”
auto   b(int x) { return x; };  //b 的类型为 int b(int)
auto   c = a;                     //优先选择 int (*c)[20]
```

第二个维度以后全部保留

如果是 int a[10][20][30][40];

Auto c=a **//优先选择 int (*c)[20][30][40]**

Auto c=a[1] //int (*c)[30][40]

关键字 decltype 用来提取表达式的类型。

凡是需要类型的地方均可出现 decltype。

可用于变量、成员、参数、返回类型的定义以及 new、sizeof、异常列表、强制类型转换。

可用于构成新的类型表达式。

int a[10][20];

decltype(a) *p = &a; //等价 int (*p)[10][20]

12.5 Lambda 表达式

Lambda 表达式的调用机制：

定义 Lambda 表达式时，将创建一个**匿名类**，同时创建一个有名字的对象。

每次调用 Lambda 表达式，都是利用该对象去调用（）运算符重载函数 **operator( )**，即：对象.**operator()(…)**。

operator()的属性是 const 的，因此 operator()不能修改**匿名类中的实例成员变量**。但通过 将 Lambda 表达式修改为 **mutable** 属性，使得匿名类中的所有实例成员变量都具备 mutable 属性，这样 operator( ) 可以**修改匿名类**中所有的实例成员变量。

lambda 表达式形式：

[capture list] (parameter list) -> return type { function body }

capture list：捕获列表，用于获得 lambda 函数体外变量的值（lambda 表达式根据这些捕获到的变量创建匿名类的实例成员变量）。捕获可以分为按值捕获和按引用捕获。非局部变量，如静态变量、全局变量等不需要捕获，直接使用。

parameter list：参数列表（调用时传入的参数），可以省略。从 C++14 开始，支持默认参数。

return type：返回值类型。可以省略，这种情况下根据 lambda 函数体中的 return 语句推断出返回类型，如果函数体中没有 return，则返回类型为 void。

function body：函数体（即 ()运算符的重载函数 operator( ) 的函数体）。

Lambda 表达式的调用方式：

auto f = [](int x)->int { return x * x; }; //创建一个匿名类，同时创建对象 f.

int x = f(10); //等价：int x = f.operator()(10) （ x = 100 ）

捕捉变量：

(1) 捕捉 lambda 函数体外变量的值（lambda 表达式根据这些捕获到的变量创建匿名类的实例成员变量）。

(2) 类中实例函数成员定义的 Lambda 表达式，自动捕获类的 this 指针。

[ ] 不捕获任何变量。

[&] 以引用方式捕获所有变量。

[=] 用值的方式捕获所有变量。

[varName] 以值方式捕获变量 varName。

[this] 捕获所在类的 this 指针。

Lambda 表达式的本质：

```
int a = 1;
int main() {
        static int b = 2;
        int   m = 3, n = 4;
        char *s = new char [10] {'a', 'b', 'c', 0};
        auto f = [m, &n, s](int x)->char * {
```

```
 s[0] += m+n+x+a+b;
                    m++;   //错：匿名类的实例数据成员 m 是 const 的
                    n++;    //对：匿名类的实例数据成员 n 是引用变量,
                   //            可以修改所指向的内存单元
                    a++;    //对：全局变量 a 不是匿名类的数据成员
                    b++;    //对：静态变量 b 不是匿名类的数据成员
                    return s;
        };  //创建匿名类及其对象 f
        f(5)[0] = '1';            //等价：f.operator()(5)[0] = '1'
        std::cout << s;        //1bc
        f.operator()(6);      //等价：f(6)
        std::cout << s;        //Dbc
}
```

Lambda 表达式的解释：

为了方便解释，下面用 A 表示匿名类的名称。

调用 f(…) ➹ f.operator()(…)

```
class {
        int   m, &n;
        char *s;
        A(int m, int &n, char *s): m(m), n(n), s(s) { }
        char *operator()(int x) const {
                (A::s)[0] += A::m + A::n + x + a + b;
                A::m++;    //错，不能改变 A::m
                A::n++;
                ::a++;
                b++;      //main::b++
//b 是和 lambda 表达式定义的作用域内的元素,如果是静态的就可以访问,不是静态的不行
                return A::s;
        }
} f;
```

Lambda 表达式的调用机制解释：

```
int main() {
        static int a = 1;
        int   m = 2;
        auto f = [m](int x) mutable -> int {
                    m += a + x;
                    return m;
        };  //创建匿名类及其对象 f
        int   i = f(0);       //i = 3 （ f.operator()(0) ）
        int   j = f(0);       //j = 4
        printf("%d, %d, %d \n", m,i,j);   //2, 3, 4
}
```

Lambda 表达式的调用机制：

定义 Lambda 表达式时，将创建一个匿名类，同时创建一个有名字的对象。

每次调用 Lambda 表达式，都是利用**该对象去调用（）运算符重载函数 operator()**，即：对象.operator()(…)。

operator()的属性是 const 的，因此 **operator()不能修改匿名类中的实例成员变量**。但通过 将 Lambda 表达式修改为 mutable 属性，使得匿名类中的所有实例成员变量都具备 mutable 属性，这样 operator( ) 可以修改匿名类中所有的实例成员变量。

对 Lambda 表达式的解释：

为了方便解释，下面用 A 表示匿名类的名称。

f (…) 则有 f.operator()(…)

```
class {
        mutable int    m;
        A(int m): m(m) { }
        int    operator()(int x) const {
                A::m += a + x;      // A::m += main::a + x
                return A::m;
        }
} f;
```

[]里是表达式的类的内容,()是传参,-》是()重载函数的返回值

Lambda 表达式的匿名类与普通匿名类的区别：

普通的匿名类可以生成多个有名字的对象，**Lambda 表达式的匿名类只能产生一个有名字的对象（这个对象是在定义 Lambda 表达式时创建的）**；

普通匿名类的实例成员函数有对象的 this 指针，Lambda 表达式匿名类的实例成员函数没有对象的 this 指针；

如果在类 A 的实例成员函数中定义了 Lambda 表达式，则该 Lambda 表达式自动捕获类 A 的 this 指针。因此，在 Lambda 表达式的函数体中可以直接访问类 A 的任何成员。

构建 lambda 表达式的时候,先把数据存进匿名表达式里面,也就是说:调用 f 的时候里面的 xyz 之类的元素是定义的时候的 xyz 等元素

```
int a = 1;
int main( ) {
    static int x = 3;
    int y = 4;
    int z = 5;
    auto f = [y, &z](int v) -> int {
//        y++;    //错：f是const，不能修改匿名类的成员变量 y
          z++;    //对：z是引用，可以修改引用所指的变量
          return a+x+y+z+v;
    };  //创建一个匿名类及其对象f
    auto g = [=](int v) -> int {
//        z++;   //错：  g是const，不能修改匿名类的成员变量 z
          return a+x+y+z+v;
    };  //创建一个匿名类及其对象g
    auto h = [=](int v) mutable -> int {
          y++;    //对：mutable int y
          z++;    //对：mutable int z
          return a+x+y+z+v;
    };  //创建一个匿名类及其对象h
    int z1 = f(100);    // z1 = ?
    int z2 = g(100);   // z2 = ?
    int z3 = h(100);   // z3 = ?
    std::cout << z1 <<std::endl;
    std::cout << z2 <<std::endl;
    std::cout << z3 <<std::endl;
}
```

```
114
113
115
```

```
int a = 1;
int main( ) {
    static int x = 3;
    int y = 4;
    int z = 5;
    auto f = [y, &z](int v) -> int {
//        y++;    //错：f是const，不能修改匿名类的成员变量 y
          z++;    //对：z是引用，可以修改引用所指的变量
          return a+x+y+z+v;
    };  //创建一个匿名类及其对象f
    int z1 = f(100);    // z1 = ?
    auto g = [=](int v) -> int {
//        z++;   //错：  g是const，不能修改匿名类的成员变量 z
          return a+x+y+z+v;
    };  //创建一个匿名类及其对象g
    auto h = [=](int v) mutable -> int {
          y++;    //对：mutable int y
          z++;    //对：mutable int z
          return a+x+y+z+v;
    };  //创建一个匿名类及其对象h

    int z2 = g(100);   // z2 = ?
    int z3 = h(100);   // z3 = ?
    std::cout << z1 <<std::endl;
    std::cout << z2 <<std::endl;
    std::cout << z3 <<std::endl;
```

```
114
114
116
```

第 13 章 模版与内存回收

这个关系和类是一样的,都是构建一个模版,然后引用来构造类的实例

13.1 变量模版与实例

变量模板使用类型形参定义变量的类型，可**根据类型实参生成变量模板的实例变量**。

生成实例变量的途径有两种：一种是从变量模板隐式地或显式地生成模板实例变量；另一种是通过函数模板（见 13.2 节）和类模板（见 13.4 节）生成。

在定义变量模板时，类型形参的名称可以使用关键字 **class 或者 typename** 定义，即可以使用 “template<class T>” 或者 “template<typename T>”。

生成模板实例变量时，将使用实际类型名、类名或类模板实例代替 T。

```
#include<stdio.h>
template<typename    T>
constexpr T pi = T(3.14159265358979323 85L);     //定义变量模板 pi，其类型形参为 T
T 都是临时传入的
      const float &d1 = pi<float>;                //引用变量模板生成的模板实例变量 pi< float>
      printf("%p\n", &d1);
      const double &d2 = pi<double>;          //引用变量模板生成的模板实例变量 pi<double>
      printf("%p\n", &d2);
```

调用模版名《类型》构建一个变量传给左值

变量模板不能在函数内部声明。

显式或隐式实例化生成的**模板实例变量和变量模板的作用域相同**。因此，变量模板生成的模板实例变量只能为**全局变量或者模块静态变量**。

模板的参数列表除了类型形参外，还可以有非类型的形参 ,非类型形参可以定义默认值。实例化变量模板时，非类型形参需要传递常量作为实参。

可使用“template 类型 模板名<类型实参>”，显式生成匿名的实例变量。

```
emplate<class T, int x=0>
T g = T(10 + x);
//生成匿名实例变量  g<float, 0>
template float g<float>;
//将匿名实例变量  g<float, 0>的值拷贝给 g1
float    g1 = g<float>;
float    &g2 = g<float>;
const float &g3 = g<float>;
const float &g4 = g<float, 0>;
const float &g5 = g<float, 4>;
const float &g6 = g<float, sizeof(printf(“A”))>;   //去掉 sizeof？
int main(void)
{     //将匿名实例变量  g<float, 0>的值拷贝给 a1
        float a1 = g<float>;
        float &a2 = g<float>;
        const float &a3 = g<float>;
        const float &a4 = g<float, 0>;
        const float &a5 = g<float, 4>;
        //&g2=&g3=&g4=&a2=&a3=&a4,
        //&g5=&g6=&a5,
```

```
    //&g1!=&g2, &a1!=&a2
    //g5=g6=a5=14, other variables = 10
    a2 = 1;     //g2=g3=g4=a2=a3=a4=1
    //a3 = 100;   //error
}
```

1.对于传引用,改变一个会改变许多个

2.对于模版的内部可以加上默认值,在模版的构造过程中可以调用,也可以传参数改变

13.2 函数模版

函数模板不能在非成员函数的内部声明。

根据函数模板生成的模板实例函数也和函数模板的作用域相同。

在函数模板中，可以使用类型形参和非类型形参。

非类型形参需要传递**常量**作为实参。

1.函数模版中的《》和()都可以定义默认参数,传参方式根据括号的位置确定

```
template <class T, int m=0>  //class 可用 typename 代替
void swap(T &x, T &y=m)
{
    T   temp = x;
    x = y;
    y = temp;
}
```

2.函数参数中可以显示制定 class 的类别,也可以隐式指定 class 的类别(自动配对)

```
long x = 123, y = 456;
    char a = 'A', b = 'B';
    A   c(1, 2, 3), d(4, 5, 6);
    swap<long, 0>(x, y);     //必须用常量传给非类型实参 m
    swap(x, y);          //自动生成实例函数 void swap(long &x, long &y)
    swap(a, b);          //自动生成实例函数 void swap(char &x, char &y)
    swap(c, d);          //自动生成实例函数 void swap(A &x, A &y)
    convert(a, y);       //自动生成实例函数 char convert (char &x, long &y)
```

3.也可以单独定义函数成员作为模版

```
template <typename T> ANY(T x) {   //单独定义构造函数模板
        p = new T(x);
        t = typeid(T).name();
    }
```

4.函数模版可以加入可变形参:函数模板的类型形参允许参数个数可变 。“…” 表示任意个类型形参，并且各形参的类型可以不同。

```
template <class H, class...T>   //递归下降展开 println()的参数表
int println(H h, T...t) {
    cout << h << "*";
    return 1 + println(t...);   //递归下降调用
}
```

5.可以对某个类型构造特化函数,优先调用特化函数

```
template <typename T>
T max(T a, T b)
```

```
{
    return a>b? a : b;
}
template < >     //此行可省，特化函数将被优先调用
const char *max(const char *x, const char *y)     //特化函数：用于隐藏模板实例函数
{
    return strcmp(x, y)>0? x : y;     //进行字符串内容比较
}
const char *p = “ABCD”;          //字符串常量“ABCD”的默认类型为 const char *
    const char *q = "EFGH";
    p = max(p, q);              //调用特化定义的实例函数，进行字符串内容比较
```

13.3 类模版

类模板也称为类属类或参数化的类，用于为相似的类定义一种通用模式。

编译程序根据类型实参生成相应的类模板实例类，也可称为模板类或类模板实例。

类模板既可包含类型参数，也可包括非类型参数。

类型参数可以包含任意个类型形参。

非类型形参在实例化是必须使用**常量**做为实参。

1.在外部用::引用时注意:

```
VECTOR <T, v>::VECTOR(int n)     //必须用 VECTOR <T, v>作为类名
```

2.

```
template <class T, int v=20>
class VECTOR
{
    T    *data;
    int    size;
public:
    VECTOR(int n = v+5);
    ~VECTOR( ) noexcept;
    T &operator[ ](int);
};
    VECTOR<int>    LI(10);          //定义包含 10 个元素的整型向量 LI
    VECTOR<short> LS;               //定义包含 25 个元素的短整型向量 LS
    VECTOR<int>    LL(30);          //定义包含 30 个元素的长整型向量 LL
    VECTOR<char *> LC(40);          //定义包含 40 个元素的 char *  向量 LC
    VECTOR<double, 10(这个 10 没用)> LD(40);     //非类型形参必须使用常量作为实参,
40 个元素
```

3.用类模版定义基类和派生类

```
template <class T>     //定义基类的类模板
class VECTOR
{
    T    *data;
    int    size;
public:
    int getsize( ) { return size; };
    VECTOR(int n) { data = new T[size = n]; };
```

```
    ~VECTOR( ) noexcept { if(data) {delete[ ]data; data=nullptr; size=0; } };
    T &operator[ ](int i) { return data[i]; };
};
template <class T>     //定义派生类的类模板
class STACK : public VECTOR<T> {     //派生类类型形参 T 作为实参, 实例化 VECTOR<T>
    int top;
public:
    int full( ) { return top==VECTOR<T>::getsize( ); }
    int null( ) { return top==0; }
    int push(T t);
    int pop(T &t);
    STACK(int s): VECTOR<T>(s) { top = 0; };
    ~STACK( ) noexcept { };
};
template <class T>
int STACK<T>::push(T    t)
{
    if ( full( ) ) return 0;
    (*this)[top++] = t;
    return 1;
 }
```

注意派生的形式

4.注意一下多个类型的形参的顺序变化对类模版的定义没什么影响

```
template<class T2, class T1> void A<T2, T1>::f1( ) { }       //正确：A<T2, T1>同类型形参一致
template<class T1, class T2> void A<T1, T2>::f2( ) { }       //正确：A<T1, T2>同类型形参一致
//template<class T2, class T1>void A<T1, T2>::f2( ) { }     //错误：A<T1, T2>与类型形参不同
```

注意类模版的定义的顺序与函数模版定义的顺序

5.用……表示类型形参的时候,表示类型形参有任意个类型参数

6.当然还可以用 auto 来定义模版

```
template <class T> auto n = new VECTOR<T>[10] { };
```

7.也可在变量、函数参数、返回类型等定义时实例化类模板。

实例化生成的类同类模板的作用域相同。实例化包含非类型形参的模板必须用**常量**作为非类型形参的实参。

当实例化生成的类实例、函数成员实例不合适时，可以自定义类、函数成员隐藏编译自动生成的类实例或函数成员。

当然还可以显式定义

8.

```
template < >       //定义特化的字符指针向量类
class VECTOR <char *>
{
    char    **data;
    int    size;
public:
    VECTOR(int);                                //特化后其所属类名为 VECTOR <char *>
    ~VECTOR( ) noexcept;     //特化后其所属类名为 VECTOR <char *>，不是虚函数
```

```
    virtual char*& operator[ ](int i) { return data[i]; };    //特化后为虚函数
};
```

可以定义特化的类

9.类模板中可以定义实例成员指针。见如下例 13.18。

实例化类模板时，类模板中的成员指针类型随之实例化。

使用类模板的实例化类作为类模板实例化的形参时，会出现嵌套的实例化现象。原本没有问题的类型形参 T，用实参 int 实例化 new T[10]时没有问题；但在嵌套实例化时，若用 A<int>实例化 new T[10]时，则会要求类模板 A 定义定义无参构造函数 A<int>::A()。例 13.18

若类模板中使用非类型形参，实例化时使用表达式很可能出现“>”，导致编译误认为模板参数列表已经结束，此时可用“（）”如 List<int, (3>2)> L1(8);。

类模板常用在 STL 标准类库等需要泛型定义的场合。见如下例 13.20：注意其中类型转换 static_cast 等的用法。

```
template <class T, int n=10>
struct A {
    static   T   t;
    T   u;
    T   *v;
    T   A::*w;
    T   A::*A::* x;
    T   A::**y;
    T   *A::*z;
    A(T k=0, int h=n);    //因 A( )被调用，故必须定义 A( )，等价于调用 A(0, n)
    ~A( ) { delete [ ] v; }
};
template <class T, int n>
T   A<T, n>::t = 0;    //类模板静态成员的初始化
main 函数: A<int> a(5);                          //等价于“A<int, 10> a(5);”
    int u = 10, *v = &u;
    int A<int>::*w = &A<int>::u;    //等价于“int A<int, 10>::* w;”
    int A<int>::*A<int>::*x = &A<int>::w;
    int A<int>::**y = &w;
    int *A<int>::*z = &A<int>::v;
    v = &A<int>::t;
    v = &a.u;
    y = &a.w;
    A<A<int>> b(a);    //等价: A<A<int,10>, 10> b(a)，构造 b 时调用 A<int>::A( )
    A<int>   A<A<int>>::*c= &A<A<int>>::u;
    a = b.*c;
    A<int> A<A<int>>::* A<A<int>>::*d = &A<A<int>>::w;
```

Tips:

为解决内存泄漏问题，可像 Java 那样定义一个始祖基类 Object。

所有其他类都从 Object 继承，比如 Name。参见例 13.21。

定义一个 Type 类模板，用于管理类 Name 的对象引用计数，若对象被引用次数为 0，则可析构该对象。Type 类模板的构造函数使用 Name *作为参数, 所有 Name 的对象都是通过 new

产生的。 Type 类模板的赋值运算符重载函数负责对象的引用计数。
当要使用 Name 产生对象时，可用 Name 作为类模板 Type 的类型实参，产生实例化类，然后使用该实例化类。
用两个栈模拟一个队列:

```
//以下初始化一定要用 std::move，否则 QUEUE 是移动赋值而其下层是深拷贝赋值
template <typename T>
QUEUE<T>::QUEUE(QUEUE &&s) noexcept: STACK<T>(move(s)), s2(move(s.s2)) { }
template <typename T>
QUEUE<T> &QUEUE<T>::operator=(QUEUE<T> &&s) noexcept {
    //以下赋值一定用 static_cast，否则 QUEUE 是移动赋值而其下层是深拷贝赋值
    *(STACK<T> *)this = static_cast<STACK<T> &&>(s);
    //等价于 STACK<T>::operator=(static_cast<STACK<T>&&>(s));
    //或等价于 STACK<T>::operator=(std::move(s));
    s2 = static_cast<STACK<T> &&>(s.s2);
    //等价于“s2=std::move(s.s2);”，可用“std::move”代替“static_cast<STACK<T>&&>”
    return *this;
}
```