C++基础与提高

王桂林

作者：王桂林　　技术交流：329973169

1.综述 C++

1.1.作者

1982 年，美国 AT&T 公司贝尔实验室的 Bjarne Stroustrup 博士在 c 语言的基础上引入并扩充了面向对象的概念，发明了一种新的程序语言。为了表达该语言与 c 语言的渊源关系，它被命名为 C++。而 Bjarne Stroustrup（本贾尼·斯特劳斯特卢普）博士被尊称为 C++语言之父。

1.1.1.历史背景

C 语言作为结构化和模块化的语言，在处理较小规模的程序时，比较得心应手。但是当问题比较复杂，程序的规模较大的时，需要高度的抽象和建模时，c 语言显得力不从心。

1.1.2.应“运”而生？运为何？

为了解决软件危机，20 世纪 80 年代，计算机界提出了 OOP(object oriented programming)思想，这需要设计出支持面向对象的程序设计语言。Smalltalk 就是当时问世的一种面向对象的语言。而在实践中，人们发现 c 是如此深入人心，使用如此之广泛，以至于最好的办法，不是发明一种新的语言去取代它，而是在原有的基础上发展它。在这种情况下 c++应运而生，最初这门语言并不叫 c++而是 c with class (带类的 c)。

1.1.3.C++发展大计事

在“C with Class”阶段，研制者在 C 语言的基础上加进去的特征主要有：类及派生类、共有和私有成员的区分、类的构造函数和析构函数、友元、内联函数、赋值运算符的重载等。

1985 年公布的 C++语言 1.0 版的内容中又添加了一些重要特征：虚函数的概念、函数和运算符的重载、引用、常量（constant）等。

1989 年推出的 2.0 版形成了更加完善的支持面向对象程序设计的 C++语言，新增加的内容包括：类的保护成员、多重继承、对象的初始化与赋值的递归机制、抽象类、静态成员

函数、const 成员函数等。

1993 年的 C++语言 3.0 版本是 C++语言的进一步完善，其中最重要的新特征是模板（template）,此外解决了多重继承产生的二义性问题和相应的构造函数与析构函数的处理等。

1998 年 C++标准（ISO/IEC14882 Standard for the C++ Programming Language）得到了国际标准化组织（ISO）和美国标准化协会（ANSI）的批准，标准 C++语言及其标准库更体现了 C++语言设计的初衷。名字空间的概念、标准模板库（STL）中增加的标准容器类、通用算法类和字符串类型等使得 C++语言更为实用。此后 C++是具有国际标准的编程语言，该标准通常简称 ANSI C++或 ISO C++ 98 标准，以后每 5 年视实际需要更新一次标准。

后来又在 2003 年通过了 C++标准第二版（ISO/IEC 14882:2003）：这个新版本是一次技术性修订，对第一版进行了整理——修订错误、减少多义性等，但没有改变语言特性。这个版本常被称为 C++03。[2]

此后，新的标准草案叫做 ***C++ 0x***。对于 C++ 0x 标准草案的最终国际投票已于 2011 年 8 月 10 日结束，并且所有国家都投出了赞成票，C++0x 已经毫无异议地成为正式国际标准。先前被临时命名为 C++0x 的新标准正式定名为 ISO/IEC 14882:2011，简称 ISO C++ 11 标准。**C++ 11** 标准将取代现行的 C++标准 C++98 和 C++03。国际标准化组织于 2011 年 9 月 1 日出版发布《ISO/IEC 14882:2011》，名称是：Information technology -- Programming languages -- C++ Edition: 3。

1.1.4.现今地位

Jan 2016	Jan 2015	Change	Programming Language	Ratings	Change
1	2	↑	Java	21.465%	+5.94%
2	1	↓	C	16.036%	-0.67%
3	4	↑	C++	6.914%	+0.21%
4	5	↑	C#	4.707%	-0.34%
5	8	↑	Python	3.854%	+1.24%
6	6		PHP	2.706%	-1.08%
7	16	⇈	Visual Basic .NET	2.582%	+1.51%
8	7	↓	JavaScript	2.565%	-0.71%
9	14	⇈	Assembly language	2.095%	+0.92%
10	15	⇈	Ruby	2.047%	+0.92%
11	9	↓	Perl	1.841%	-0.42%
12	20	⇈	Delphi/Object Pascal	1.786%	+0.95%
13	17	⇈	Visual Basic	1.684%	+0.61%
14	25	⇈	Swift	1.363%	+0.62%
15	11	⇊	MATLAB	1.228%	-0.16%
16	30	⇈	Pascal	1.194%	+0.52%
17	82	⇈	Groovy	1.182%	+1.07%
18	3	⇊	Objective-C	1.074%	-5.88%
19	18	↓	R	1.054%	+0.01%
20	10	⇊	PL/SQL	1.016%	-1.00%

数据来源 http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

1.2.应用领域

如果项目中，既要求效率又要建模和高度抽像，那就选择 c++吧。

1.2.1.系统层软件开发

C++的语言本身的高效和面向对象，使其成为系统层开发的不二之选。比如我们现在用的 window 桌面，GNOME 桌面系统, KDE 桌面系统。


1.2.2.服务器程序开发

面向对像，具有较强的抽像和建模能力。使其在电信，金融，电商，通信，媒体，交换路由等方面中不可或缺。

1.2.3.流戏，网络，分布式，云计算

以其效率与建模，上述领域无可取代。








1.2.4.基础类库/科学计算

比如大名鼎鼎的 MFC/ACE/QT/GTK 等类库。

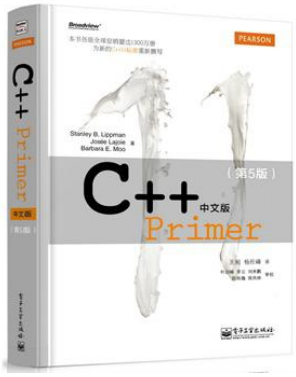
1.3.内容

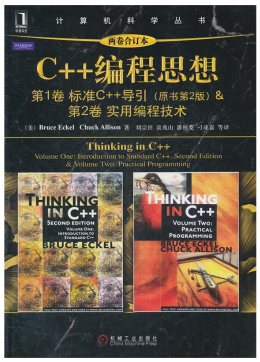
C++语言的名字，如果看作 c 的基本语法，是由操作数 c 和运算符后++构成。C++是本身这门语言先是 c,是完全兼容 c.然后在此基础上++。这个++包含三大部分，c++对 c 的基础语法的扩展，面向对像(继承，封装，多态)，STL 等。

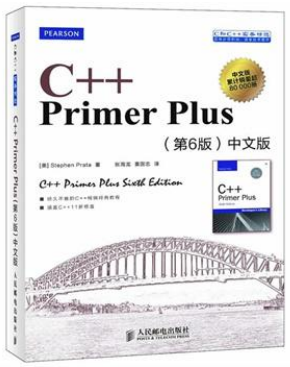
1.4.开发环境

下载地址：http://pan.baidu.com/s/1ntYhz8P

1.5.书籍推荐







2.C++对 C 的扩展(Externsion)

曾有人戏谑的说，C++作为一种面向对象的语言，名字起的不好，为什么呢？用 c 的语法来看，++ 操作符是 post ++ 。

2.1.类型增强

2.1.1.类型检查更严格

比如，把一个 const 类型的指针赋给非 const 类型的指针。c 语言中可以通的过，但是在 c++中则编不过去。

```
int main()
{
    const int a = 100;
    int b = a;

    const int *pa = &a;
    int *pb = pa;
    return 0;
}
```

2.1.2.布尔类型（bool）

c 语言的逻辑真假用 0 和非 0 来表示。而 c++中有了具体的类型。

```
int main()
{
    bool flag = true;
    if(flag != false)
    {
        printf("i know bool type now\n");
    }

    printf("bool size = %d\n",sizeof(bool));
    return 0;
}
```

2.1.3.真正的枚举(enum)

c 语言中枚举本质就是整型，枚举变量可以用任意整型赋值。而 c++中枚举变量，只能用被枚举出来的元素初始化。

```
enum season {SPR,SUM,AUT,WIN};
int main()
{
    enum season s = SPR;
    s = 0;
    return 0;
}
```

error

```
D:\WorkSpace\qtcpp\1\main.cpp:10: error: invalid conversion from 'int' to
 'season' [-fpermissive]
```

2.1.4. 表达式的值可被赋值

c 语言中表达式通常不能作为左值的，即不可被赋值，c++中某些表达式是可以赋值的。比如：

```
#include <iostream>

using namespace std;

int main(void)
{
    int a,b = 5;
    (a = b) = 10;
    cout<<"a = "<<a<<" b = "<<b<<endl;

    (a<b? a:b) = 200;
     cout<<"a = "<<a<<" b = "<<b<<endl;
    return 0;
}
```

2.2.输入与输出(cin /cout)

第一个真正意义上的 c++程序，c++程序的后缀名为 cpp。假设程序名叫 xxx 则应该写成 xxx.cpp。

2.2.1.cin && cout

cin 和 cout 是 C++的标准输入流和输出流。他们在头文件 iostream 中定义。

流名	含义	隐含设备	流名	含义	隐含设备
cin	标准输入	键盘	cerr	标准错误输出	屏幕
cout	标准输出	屏幕	clog	cerr 的缓冲输出	屏幕

```
int main()
{
    char name[30];
    int age;
    cout<<"pls input name and age:"<<endl;
    cin>>name;
    cin>>age;
// cin>>name>>age;
    cout<<"your name is: "<<name<<endl;
    cout<<"your age is: "<<age<<endl;
    return 0;
}
```

```
//string name  安全性对比
//%d%c 的问题
```

2.2.2.格式化

c 语言中 printf 拥有强大的格式化控制。c++亦可以实现，略复杂。

2.2.2.1.设置域宽及位数

对于实型，cout 默认输出六位有效数据，setprecision(2) 可以设置有效位数，setprecision(n)<<setiosflags(ios::fixed)合用，可以设置小数点右边的位数。

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    printf("%c\n%d\n%f\n",'a',100,120.00);
    printf("%5c\n%5d\n%6.2f\n",'a',100,120.00);

    cout<<setw(5)<<'a'<<endl<<setw(5)<<100<<endl
       <<setprecision(2)<<setiosflags(ios::fixed)<<120.00<<endl;
    return 0;
}
```

2.2.2.2.按进制输出

输出十进制，十六进制，八进制。默认输出十进制的数据。

```
    int  i = 123;
    cout<<i<<endl;
    cout<<dec<<i<<endl;
    cout<<hex<<i<<endl;
    cout<<oct<<i<<endl;
    cout<<setbase(16)<<i<<endl;
```

2.2.2.3.设置填充符

还可以设置域宽的同时，设置左右对齐及填充字符。

```
int main()
{
    cout<<setw(10)<<1234<<endl;
    cout<<setw(10)<<setfill('0')<<1234<<endl;
    cout<<setw(10)<<setfill('0')<<setiosflags(ios::left)<<1234<<endl;
    cout<<setw(10)<<setfill('-')<<setiosflags(ios::right)<<1234<<endl;
    return 0;
}
```

2.3.函数重载(function overload)

2.3.1.引例

如下函数分别求出整理数据和浮点型数据的绝对值:

```
int iabs(int a)
{
   return a>0? a:-a;
}
double fabs(double a)
{
```

```
    return a>0? a:-a;
}
```

C++ 致力于简化编程，能过函数重名来达到简化编程的目的。

```
int abs(int a)
{
    return a>0? a:-a;
}
double abs(double a)
{
    return a>0? a:-a;
}
```

2.3.2.重载规则与调用匹配（overload&match）

◆重载规则：

1，函数名相同。

2，参数个数不同，参数的类型不同，参数顺序不同，均可构成重载。

3，返回值类型不同则不可以构成重载。

如下：

```
void func(int a);     //ok
void func(char a);    //ok
void func(char a,int b);  //ok
void func(int a, char b); //ok

char func(int a);  //与第一个函数有冲突
```

有的函数虽然有返回值类型，但不与参数表达式运算，而作一条单独的语句。

◆匹配原则：

1，严格匹配，找到则调用。

2，通过隐式转换寻求一个匹配，找到则调用。

```
#include <iostream>
using namespace std;
void print(double a){
    cout<<a<<endl;
}
void print(int a){
    cout<<a<<endl;
}
int main(){
    print(1);       // print(int)
    print(1.1);     // print(double)
    print('a');     // print(int)
    print(1.11f);   // print(double)
    return 0;
}
```

注：

C++ 允许，int 到 long 和 double，double 到 int 和 float 隐式类型转换。遇到这种

情型，则会引起二义性。

例：将上题上的 print(int a)中的类型 int 改为 double。

```
error: call of overloaded 'print(int)' is ambiguous
     print(1);       // print(int)
error: call of overloaded 'print(char)' is ambiguous
     print('a');     // print(int)
```

解决方法，在调用时强转。

2.3.3.重载底层实现（name mangling）

C++利用 name mangling(倾轧)技术，来改名函数名，区分参数不同的同名函数。

实现原理：用 v-c- i-f- l- d 表示 void char int float long double 及其引用。

```
void func(char a);   // func_c(char a)
void func(char a, int b, double c);
//func_cid(char a, int b, double c)
```

2.3.4.extern “C”

name mangling 发生在两个阶段，.cpp 编译阶段，和.h 的声明阶段。

只有两个阶段同时进行，才能匹配调用。

mystring.h

```
extern "C" {
int myStrlen(char *str);
}
```

mystring.cpp

```
int myStrlen(char *str)
//#include "mystring.h"
int myStrlen(char *str)
{
    int len = 0;
    while(*str++)
        len++;
    return len;
}
```

main.cpp

```
#include <iostream>
#include "mystring.h"
using namespace std;

int main()
{
    char *p = "china";
    int len;
    len = myStrlen(p);
    return 0;
```

```
}
```

c++ 完全兼容 c 语言，那就面临着，**完全兼容 c 的类库**。由.c 文件的类库文件中函数名，并没有发生 name mangling 行为，而我们在包含.c 文件所对应的.h 文件时，.h 文件要发生 name manling 行为，因而会发生在链接的时候的错误。

C++为了避免上述错误的发生，重载了关键字 extern。只需要,要避免 name manling 的函数前，加 extern "C" 如有多个，则 extern "C"{}

我们看一个系统是怎么处理的：

sting.h

```
extern "C" {
  char * __cdecl _strset(char *_Str,int _Val) __MINGW_ATTRIB_DEPRECATED_S
EC_WARN;
  char * __cdecl _strset_l(char *_Str,int _Val,_locale_t _Locale) __MINGW
_ATTRIB_DEPRECATED_SEC_WARN;
  char * __cdecl strcpy(char * __restrict__ _Dest,const char * __restrict
__ _Source);
  char * __cdecl strcat(char * __restrict__ _Dest,const char * __restrict
__ _Source);
  int __cdecl strcmp(const char *_Str1,const char *_Str2);
  size_t __cdecl strlen(const char *_Str);
  size_t __cdecl strnlen(const char *_Str,size_t _MaxCount);
  void *__cdecl memmove(void *_Dst,const void *_Src,size_t _Size) __MINGW
_ATTRIB_DEPRECATED_SEC_WARN;
}
```

2.4.操作符重载(operator overload)

前面用到的<<本身在 c 语言中是位操作中的左移运算符。现在又用用流插入运算符，这种一个字符多种用处的现像叫作重载。在 c 语中本身就用重载的现像 ,比如 & 既表示取地址，又表示位操作中的与。*既表示解引用，又表示乘法运算符。只不过 c 语言并没有开放重载机制。

C++提供了运算符重载机制。可以为自定义数据类型重载运算符。实现构造数据类型也可以像基本数据类型一样的运算特性。

```
using namespace std;
struct COMP
{
    float real;
    float image;
};
COMP operator+(COMP one, COMP another)
{
    one.real += another.real;
    one.image += another.image;
    return one;
}
int main()
{
    COMP c1 = {1,2};
```

```
    COMP c2 = {3,4};
    COMP sum = operator+(c1,c2); //c1+c2;
    cout<<sum.real<<" "<<sum.image<<endl;
    return 0;
}
```

示例中重载了一个全局的操作符+号用于实现将两个自定义结构体类型相加。本质是函数的调用。

当然这个 COMP operator+(COMP one, COMP another)，也可以定义为 COMP add(COMP one, COMP another)，但这样的话，就只能 COMP sum = add(c1,c2)，而不能实现 COMP sum = c1 +c2 了。

后序我们在学习完成类以后，重点讲解重载。

2.5.默认参数(default parameters)

通常情况下，函数在调用时，形参从实参那里取得值。对于多次调用用一函数同一实参时，C++给出了更简单的处理办法。给形参以默认值，这样就不用从实参那里取值了。

2.5.1.示例

◆单个参数

```
#include <iostream>
#include <ctime>

using namespace std;

void weatherForcast(char * w="sunny")
{
    time_t t = time(0);
    char tmp[64];
    strftime( tmp, sizeof(tmp), "%Y/%m/%d %X %A ",localtime(&t) );
    cout<<tmp<< "today is weahter "<<w<<endl;
}

int main()
{
    //sunny windy cloudy foggy rainy
    weatherForcast();
    weatherForcast("rainny");
    weatherForcast();
    return 0;
}
```

◆多个参数

```
float volume(float length, float weight = 4,float high = 5)
{
    return length*weight*high;
}
int main()
{
    float v  = volume(10);
    float v1 = volume(10,20);
```

```
    float v2 = volume(10,20,30);
    cout<<v<<endl;
    cout<<v1<<endl;
    cout<<v2<<endl;
    return 0;
}
```

2.5.2.规则

1，默认的顺序，是从右向左，不能跳跃。

2，函数声明和定义一体时，默认认参数在定义(声明)处。声明在前，定义在后，**默认参数在声明处。**

3，一个函数，不能既作重载，又作默认参数的函数。当你少写一个参数时，系统无法确认是重载还是默认参数。

```
void print(int a)
{

}
void print(int a,int b =10)
{

}
int main()
{
    print(10);
    return 0;
}
```

```
main.cpp:16: error: call of overloaded 'print(int)' is ambiguous
     print(10);
```

2.6.引用(Reference)

2.6.1.引用的概念

变量名，本身是一段内存的引用,即别名(alias)。此处引入的引用，是为己有变量起一个别名。

声明如下：

```
int main()
{
    int a;
    int &b = a;
}
```

2.6.2.规则

1，引用没有定义，是一种关系型声明。声明它和原有某一变量(实体)的关系。故而类型与原类型保持一致，且不分配内存。与被引用的变量有相同的地址。

2，声明的时候必须初始化，一经声明，不可变更。

3，可对引用，再次引用。多次引用的结果，是某一变量具有多个别名。

4，&符号前有数据类型时，是引用。其它皆为取地址。

```
int main()
{
    int a,b;
    int &r = a;
    int &r = b; //错误，不可更改原有的引用关系
    float &rr = b; //错误，引用类型不匹配
    cout<<&a<<&r<<endl;  //变量与引用具有相同的地址。
    int &ra = r; //可对引用更次引用，表示 a 变量有两个别名，分别是 r 和 ra

}
```

2.6.3.应用

C++很少使用独立变量的引用，如果使用某一个变量，就直接使用它的原名，没有必要使用他的别名。

值作函数参数 (call by value)

```
void swap(int a, int b); //无法实现两数据的交换
void swap(int *p, int *q); //开辟了两个指针空间实现交换
```

引用作函数参数 (call by reference)

```
void swap(int &a, int &b){
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
int main(){
    int a = 3,b = 5;
    cout<<"a = "<<a<<"b = "<<b<<endl;
    swap(a,b);
    cout<<"a = "<<a<<"b = "<<b<<endl;
    return 0;
}
```

c++中引入引用后，可以用引用解决的问题。避免用指针来解决。

2.6.4.引用提高

引用的本质是指针，C++对裸露的内存地址(指针)作了一次包装。又取得的指针的优良特性。所以再对引用取地址，建立引用的指针没有意义。

1，可以定义指针的引用，但不能定义引用的引用。

```
int a;
int* p = &a;
int*& rp = p; // ok
int& r = a;
int&& rr = r; // error
```

案例：

```
#include <iostream>
```

```
using namespace std;

void swap(char *pa,char *pb)
{
    char *t;
    t = pa;
    pa = pb;
    pb = t;
}
void swap2(char **pa,char **pb)
{
    char *t;
    t = *pa;
    *pa = *pb;
    *pb = t;
}

void swap3(char * &pa,char *&pb)
{
    char *t;
    t = pa;
    pa = pb;
    pb = t;
}


int main()
{
    char *pa = "china";
    char *pb =  "america";

    cout<<"pa "<<pa<<endl;
    cout<<"pb "<<pb<<endl;
//    swap(pa,pb);
//    swap2(&pa,&pb);
    swap3(pa,pb);
    cout<<"pa "<<pa<<endl;
    cout<<"pb "<<pb<<endl;
    return 0;
}
```

2，可以定义指针的指针(二级指针)，但不能定义引用的指针。

```
int a;
int* p = &a;
int** pp = &p; // ok
int& r = a;
int&* pr = &r; // error
```

3，可以定义指针数组，但不能定义引用数组，可以定义数组引用。

```
int a, b, c;
int* parr[] = {&a, &b, &c}; // ok
int& rarr[] = {a, b, c}; // error
int arr[] = {1, 2, 3};
int (&rarr)[3] = arr; // ok 的
```

4，常引用

const 引用有较多使用。它可以防止对象的值被随意修改。因而具有一些特性。

(1)**const 对象的引用必须是 const 的，将普通引用绑定到 const 对象是不合法的。**这个原因比较简单。既然对象是 const 的，表示不能被修改，引用当然也不能修改，必须使用 const 引用。实际上，const　int a=1; int &b=a;这种写法是不合法的，编译不过。

(2)**const 引用可使用相关类型的对象(常量,非同类型的变量或表达式)初始化。**这个是 const 引用与普通引用最大的区别。const int &a=2;是合法的。double x=3.14; const int &b=a;也是合法的。

常引用原理：

const 引用的目的是，禁止通过修改引用值来改变被引用的对象。const 引用的初始化特性较为微妙，可通过如下代码说明：

```
double val = 3.14;
const int &ref = val; // int  const &   int & const ??
double & ref2 = val;
cout<<ref<<" "<<ref2<<endl;
val = 4.14;
cout<<ref<<" "<<ref2<<endl;
```

上述输出结果为 3　3.14 和 3　4.14。因为 ref 是 const 的，在初始化的过程中已经给定值，不允许修改。而被引用的对象是 val，是非 const 的，所以 val 的修改并未影响 ref 的值，而 ref2 的值发生了相应的改变。

那么，为什么非 const 的引用不能使用相关类型初始化呢？实际上，const 引用使用相关类型对象初始化时发生了如下过程：

```
int temp = val;
const int &ref = temp;
```

如果 ref 不是 const 的，那么改变 ref 值，修改的是 temp，而不是 val。期望对 ref 的赋值会修改 val 的程序员会发现 val 实际并未修改。

```
int i=5;
const int & ref = i+5;
//此时产生了与表达式等值的无名的临时变量，
//此时的引用是对无名的临时变量的引用。故不能更改。
cout<<ref<<endl;
```

5，尽可能使用 const

use const whatever possible 原因如下：

1，使用 const 可以避免无意修改数据的编程错误。

2，使用 const 可以处理 const 和非 const 实参。否则将只能接受非 const 数据。

3，使用 const 引用，可使函数能够正确的生成并使用临时变量（如果实参与引用参数不匹配，就会生成临时变量）

2.6.5.引用的本质浅析

2.6.5.1.大小与不可再引用

引用的本质是指针，是个什么样指针呢？可以通过两方面来探究，初始化方式和大小。

```
struct TypeP
{
    char *p;
};
struct TypeC
{
    char c;
};
struct TypeR
{
    char& r;  //把引用单列出来，不与具体的对像发生关系
};

int main()
{

//    int a;
//    int &ra = &a;
//    const int rb;  //const 类型必须要初始化。


 printf("%d %d %d\n",sizeof(TypeP),sizeof(TypeC),sizeof(TypeR));
    return 0;
}
```

结论：

引用的本质是，是对常指针 type * const p 的再次包装。

char &rc　==　*pc　　double &rd ==　*pd

2.6.5.2.反汇编对比指针和引用

原程序

```
#include <iostream>

using namespace std;

void Swap(int *p, int *q)
{
    int t = *p;
    *p = *q;
    *q = t;
}
void Swap(int &p, int &q)
{
    int t = p;
    p = q;
    q = t;
}

int main()
```

```
{
    int a = 3; int b =5;
    Swap(a,b);
    Swap(&a,&b);
    return 0;
}
```

汇编程序

```
[1]{
               55                      push   %ebp
<+0x0001>       89 e5                   mov    %esp,%ebp
<+0x0003>       83 e4 f0                and    $0xfffffff0,%esp
<+0x0006>       83 ec 20                sub    $0x20,%esp
<+0x0009>       e8 ce 0a 00 00          call   0x402130 <__main>
[1]    int a = 3; int b =5;
<+0x000e>       c7 44 24 1c 03 00 00 00  movl   $0x3,0x1c(%esp)
<+0x0016>       c7 44 24 18 05 00 00 00  movl   $0x5,0x18(%esp)
[1]    Swap(a,b);
<+0x001e>       8d 44 24 18             lea    0x18(%esp),%eax
<+0x0022>       89 44 24 04             mov    %eax,0x4(%esp)
<+0x0026>       8d 44 24 1c             lea    0x1c(%esp),%eax
<+0x002a>       89 04 24                mov    %eax,(%esp)
<+0x002d>       e8 ac ff ff ff          call   0x401632 <Swap(int&, int&)>
[1]    Swap(&a,&b);
<+0x0032>       8d 44 24 18             lea    0x18(%esp),%eax
<+0x0036>       89 44 24 04             mov    %eax,0x4(%esp)
<+0x003a>       8d 44 24 1c             lea    0x1c(%esp),%eax
<+0x003e>       89 04 24                mov    %eax,(%esp)
<+0x0041>       e8 76 ff ff ff          call   0x401610 <Swap(int*, int*)>
[1]    return 0;
<+0x0046>       b8 00 00 00 00          mov    $0x0,%eax
[1]}
<+0x004b>       c9                      leave
<+0x004c>       c3                      ret
```

0x401632 <Swap(int&, int&)>

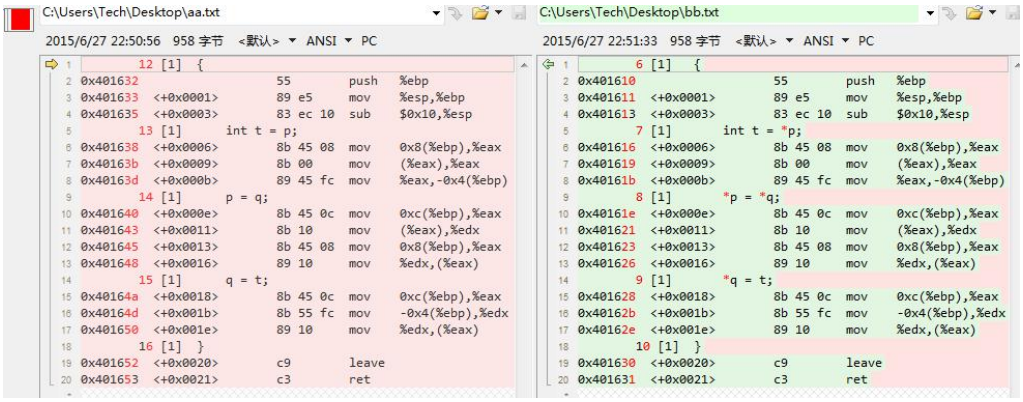
```
       12 [1]{
0x401632                 55        push   %ebp
0x401633  <+0x0001>       89 e5     mov    %esp,%ebp
0x401635  <+0x0003>       83 ec 10  sub    $0x10,%esp
       13 [1]    int t = p;
0x401638  <+0x0006>       8b 45 08  mov    0x8(%ebp),%eax
0x40163b  <+0x0009>       8b 00     mov    (%eax),%eax
0x40163d  <+0x000b>       89 45 fc  mov    %eax,-0x4(%ebp)
       14 [1]    p = q;
0x401640  <+0x000e>       8b 45 0c  mov    0xc(%ebp),%eax
0x401643  <+0x0011>       8b 10     mov    (%eax),%edx
0x401645  <+0x0013>       8b 45 08  mov    0x8(%ebp),%eax
0x401648  <+0x0016>       89 10     mov    %edx,(%eax)
       15 [1]    q = t;
0x40164a  <+0x0018>       8b 45 0c  mov    0xc(%ebp),%eax
0x40164d  <+0x001b>       8b 55 fc  mov    -0x4(%ebp),%edx
0x401650  <+0x001e>       89 10     mov    %edx,(%eax)
       16 [1]}
0x401652  <+0x0020>       c9        leave
0x401653  <+0x0021>       c3        ret
```

0x401610 <Swap(int*, int*)>

```
       6 [1]{
```

```
0x401610                    55        push   %ebp
0x401611  <+0x0001>         89 e5     mov    %esp,%ebp
0x401613  <+0x0003>         83 ec 10  sub    $0x10,%esp
       7 [1]    int t = *p;
0x401616  <+0x0006>         8b 45 08  mov    0x8(%ebp),%eax
0x401619  <+0x0009>         8b 00     mov    (%eax),%eax
0x40161b  <+0x000b>         89 45 fc  mov    %eax,-0x4(%ebp)
       8 [1]    *p = *q;
0x40161e  <+0x000e>         8b 45 0c  mov    0xc(%ebp),%eax
0x401621  <+0x0011>         8b 10     mov    (%eax),%edx
0x401623  <+0x0013>         8b 45 08  mov    0x8(%ebp),%eax
0x401626  <+0x0016>         89 10     mov    %edx,(%eax)
       9 [1]    *q = t;
0x401628  <+0x0018>         8b 45 0c  mov    0xc(%ebp),%eax
0x40162b  <+0x001b>         8b 55 fc  mov    -0x4(%ebp),%edx
0x40162e  <+0x001e>         89 10     mov    %edx,(%eax)
       10 [1]}
0x401630  <+0x0020>         c9        leave
0x401631  <+0x0021>         c3        ret
```

对比结果：



2.7.new/delete Operator

c 语言中提供了 malloc 和 free 两个系统函数，完成对堆内存的申请和释放。而 c++则提供了两关键字 new 和 delete；

2.7.1.new/new[]用法:

1.开辟单变量地址空间

```
int *p = new int; //开辟大小为 sizeof(int)空间
int *a = new int(5); //开辟大小为 sizeof(int)空间，并初始化为 5
```

2.开辟数组空间

```
一维: int *a = new int[100]{0};开辟一个大小为 100 的整型数组空间
      int **p = new int*[5]{NULL}
二维: int (*a)[6] = new int[5][6]
三维: int (*a)[5][6] = new int[3][5][6]
四维维及其以上:依此类推.
```

2.7.2.delete /delete[]用法:

1. int *a = new int;

```
delete a;   //释放单个 int 的空间
```

2.int *a = new int[5];

```
delete []a; //释放 int 数组空间
```

2.7.3.综合用法

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <typeinfo>

using namespace std;
int main()
{
    int *p = new int(5);
    cout<<*p<<endl;
     delete p;

    char *pp = new char[10];
    strcpy(pp,"china");
    cout<<pp<<endl;
    delete []pp;

    string *ps = new string("china");
    cout<<*ps<<endl; //cout<<ps<<endl;
    delete ps;

    char **pa= new char*[5];
    memset(pa,0,sizeof(char*[5]));
    pa[0] = "china";
    pa[1] = "america";
    char **pt = pa;
    while(*pt)
    {
        cout<<*pt++<<endl;
    }

    delete []pt;

    int (*q)[3] = new int[2][3];
    for(int i=0; i<2; i++)
    {
        for(int j=0; j<3; j++)
        {
            q[i][j] = i+j;
        }
    }

    for(int i=0; i<2; i++)
    {
        for(int j=0; j<3; j++)
        {
            cout<<q[i][j];
```

```
        }
        cout<<endl;
    }

    delete []q;

    int (*qq)[3][4] = new int [2][3][4];

    delete []qq;

}
```

2.7.4.关于返回值

```
int main()
{
    //c 语言版本
    char *ps = (char*)malloc(100);
    if(ps == NULL)
        return -1;
    //C++ 内存申请失败会抛出异常
    try{
        int *p  = new int[10];
    }catch(const std::bad_alloc e) {
        return -1;
    }

    //C++ 内存申请失败不抛出异常版本
    int *q = new (std::nothrow)int[10];
    if(q == NULL)
        return -1;

    return 0;
}
```

2.7.5.注意事项

1，new/delete 是关键字，效率高于 malloc 和 free.

2，配对使用，避免内存泄漏和多重释放。

3，避免，交叉使用。比如 malloc 申请的空间去 delete，new 出的空间被 free;

2.7.6.更进一步

如果只是上两步的功能，c 中的 malloc 和 free 完全可以胜任，C++就没有必要更进一步，引入这两个关键字。

此两关键字，重点用在**类对像的申请与释放**。申请的时候会调用构造器完成初始化，释放的时候，会调用析构器完成内存的清理。以后我们会重点讲。

2.8.内联函数(inline function)

2.8.1.内联

c 语言中有宏函数的概念。宏函数的特点是内嵌到调用代码中去 ,避免了函数调用的开销。

但是由于宏函数的处理发生在预处理阶段，缺失了语法检测和有可能带来的语意差错。

2.8.2.语法

C++提供了 inline 关键字，实现了真正的内嵌。

宏函数 VS inline 函数

```
#include <iostream>
#include <string.h>
using namespace std;
#if 0
优点：内嵌代码，辟免压栈与出栈的开销
缺点：代码替换，易使生成代码体积变大，易产生逻辑错误，无类型检查
#endif
#define SQR(x) ((x)*(x))

#if 0
优点：高度抽象，避免重复开发，类型检查
缺点：压栈与出栈，带来开销
#endif
inline int sqr(int x)
{
    return x*x;
}

#endif

int main()
{
    int i=0;
    while(i<5)
    {
//        printf("%d\n",SQR(i++));
        printf("%d\n",sqr(i++));
    }
    return 0;
}
```

2.8.3.评价

优点：避免调用时的额外开销（入栈与出栈操作）

代价：由于内联函数的函数体在代码段中会出现多个“副本”，因此会增加代码段的空间。

本质：以牺牲代码段空间为代价，提高程序的运行时间的效率。

适用场景：函数体很“小”，且被“频繁”调用。

2.9.类型强转(type cast)

类型转换有 c 风格的，当然还有 c++风格的。c 风格的转换的格式很简单（TYPE EXPRESSION),但是 c 风格的类型转换有不少的缺点,有的时候用 c 风格的转换是不合适的，

因为它可以在任意类型之间转换，比如你可以把一个指向 const 对象的指针转换成指向非 const 对象的指针，把一个指向基类对象的指针转换成指向一个派生类对象的指针，这两种转换之间的差别是巨大的，但是传统的 c 语言风格的类型转换没有区分这些。还有一个缺点就是，c 风格的转换不容易查找，他由一个括号加上一个标识符组成，而这样的东西在 c++程序里一大堆。所以 c++为了克服这些缺点，引进了 4 新的类型转换操作符。

2.9.1.静态类型转换：

◆ 语法格式：

```
static_cast<目标类型>（标识符）
```

◆ 转化规则：

在一个方向上可以作隐式转换，在另外一个方向上就可以作静态转换。

```
int a = 10;
int b = 3;
cout<<static_cast<float>(a)/b<<endl;  //float = int  int = float
return 0;
```

```
int *p; void *q;
p = static_cast<int*>(q);
```

```
char *p = static_cast<char*>(malloc(100));
```

2.9.2.重解释类型转换：

◆ 语法格式：

```
reinterpret_cast<目标类型>（标识符）
```

◆ 转化规则

“通常为操作数的位模式提供较低层的重新解释”也就是说将数据以二进制存在形式的重新解释，在双方向上都不可以隐式类型转换的，则需要重解释类型转换。

```
char *p; int *q;
p = q;
q = p;
```

```
int main()
{
    int x = 0x12345648;
char *p = reinterpret_cast<char*>(&x);
//char*p = static_cast<char*>(&x);
    printf("%x\n",*p);

    int a[5] = {1,2,3,4,5};

    int *q = reinterpret_cast<int*>((reinterpret_cast<int>(a) +1));
    printf("%x\n",*q);
```

```
    return 0;
}
```

2.9.3.(脱)常类型转换：

◆ 语法格式：

```
const_cast<目标类型>（标识符）//目标类类型只能是指针或引用。
```

◆语法规则

用来移除对象的常量性(cast away the constness)使用 const_cast 去除 const 限定的目的不是为了修改它的内容，使用 const_cast 去除 const 限定，通常是为了函数能够接受这个实际参数。

应用场景 1：

```
#include <iostream>
using namespace std;
void func(int & ref) //别人已经写好的程序或类库
{
    cout<<ref<<endl;
}
int main(void)
{

    const int m = 4444;
    func(const_cast<int&>(m));
    return 0;
}
```

脱掉 const 后的引用或指针可以改吗？

```
#include <iostream>

using namespace std;

int main()
{

    const int x = 200;

    int & a =const_cast<int&>(x); // int &a = x;
    a = 300;
    cout<<a<<x<<endl;
    cout<<&a<<"---"<<&x<<endl;

    int *p   =const_cast<int*>(&x); // int *p = &x;
    *p = 400;
    cout<<a<<*p<<endl;
    cout<<p<<"---"<<&x<<endl;

    struct A
    {
        int data;
    };

    const  A  xx = {1111};
```

```
    A &a1 = const_cast< A&>(xx);
    a1.data = 222;
    cout<<a1.data<<xx.data<<endl;

    A *p1 = const_cast<A*>(&xx);
    p1->data = 333;

    cout<<p1->data<<xx.data<<endl;
    return 0;
}
```

结论：

可以改变 const 自定义类的成员变量，但是对于内置数据类型，却表现未定义行为.

```
Depending on the type of the referenced object, a write operation throu
gh the resulting pointer, reference, or pointer to data member might pr
oduce undefined behavior.
```

◆const 常变量（补充）：

C++中 const 定义的变量称为常变量。**变量的形式，常量的作用，用作常量，常用于取代#define 宏常量。**

```
#include <iostream>
using namespace std;
#define N  200
int main()
{
    const int a = 200;
    int b = 300;
    int c = a +b;  //int c = N + b;
    return 0;
}
```

2.9.4.动态类型转换：

◆ 语法格式：

```
dynamic_cast<目标类型> (标识符)
```

用于多态中的父子类之间的强制转化，以后再讲。

2.10.命名空间(namespace scope)

2.10.1.为什么要引入namespace

命名空间为了**大型项目**开发，而引入的一种避免命名冲突的一种机制。比如说，在一个大型项目中，要用到多家软件开发商提供的**类库**。在事先没有约定的情况下，两套**类库**可能在存在**同名的函数或是全局变量**而产生冲突。项目越大，用到的类库越多，开发人员越多，

这种冲突就会越明显。

2.10.2.默认NameSpace（Global &Function）

Global scope 是一个程序中最大的 scope。也是引起命名冲突的根源。C 语言没有从语言层面提供这种机制来解决。也算是 C 语言的硬伤了。Global scope 是无名的命名空间。

```
//c 语言中如何访问被局部变量覆盖的全局变量
int val = 200;
int main()
{
    int *p = &val;
    int val = 100;
    printf("func   val = %d\n",val);
    printf("global val = %d\n",*p);
    return 0;
}
```

```
#include <iostream>
#include <string.h>
using namespace std;
int val = 200;
void func()
{
    return ;
}
int main()
{
    int val = 100;
    cout<<"func   val = "<<val<<endl;
    cout<<"global val = "<<::val<<endl;
    ::func(); //因为不能在函数内定义函数。所以前而的::没有意义。
    return 0;
}
```

2.10.3.语法规则

NameSpace 是对**全局(G**lobal scope)区域的**再次划分。**

2.10.3.1.声明

命令空间的声明及 namespace 中可以包含的内容

```
namespace NAMESPACE
{
    全局变量  int a;
    数据类型  struct Stu{};
    函数  void func();
    其它命名空间 namespace
}
```

2.10.3.2.使用方法

1.直接指定 命名空间： Space::a = 5;

2.使用 using+命名空间+空间元素：using **Space**::a; a = 2000;

3.使用 using +namespace+命名空间： using namespace Space;

```
#include <iostream>

using namespace std;

namespace MySpace
{
    int val = 5;
    int x,y,z;
}

int main()
{
//    MySpace::val = 200;
//    cout<<MySpace::val;

//    using MySpace::x;
//    using MySpace::y;
//    x = 100;
//    y = 200;
//    cout<<x<<y<<endl;

    using namespace MySpace;

    val = 1;
    x = 2;
    y = 3;
    z = 4;
    cout<<val<<x<<y<<z<<endl;

    return 0;
}
```

类比 std::cout /using std::cout using /namespact std;

无可辟免的冲突

```
#include <iostream>

using namespace std;


namespace Space
{
    int x;
}
namespace Other
{
    int x;
}
int main()
{
//    Space::x = 4;
//    cout<<Space::x<<endl;
```

```
//    Other::x = 5;
//    cout<<Other::x<<endl;

    {
        using Space::x;
        x = 5;
        cout<<x<<endl;
    }

    {
        using Other::x;
        x = 7;
        cout<<x<<endl;
    }

    {
        using namespace Space;
        x = 5;
    }

    {
        using namespace Other;
        x = 8;
    }


    return 0;
}
```

2.10.3.3.支持嵌套

```
#include <iostream>

using namespace std;

namespace MySpace
{
    int x = 1;
    int y = 2;

    namespace  Other {
        int m = 3;
        int n = 4;
    }
}

int main()
{
    using namespace MySpace::Other;

    cout<<m<<n<<endl;
    return 0;
}
```

2.10.3.4.协作开发

同名命名空间自动合并，对于一个命名空间中的类，要包含声明和实现。

a.h

```
#ifndef A_H
#define A_H

namespace XX {
class A
{
public:
    A();
    ~A();
};
}

#endif // A_H
```

a.cpp

```
#include "a.h"

using namespace XXX
{
    A::A()
    {
    }
    A::~A()
    {
    }
}
```

b.h

```
#ifndef B_H
#define B_H
namespace XX
{
    class B
    {
    public:
        B();
        ~B();
    };
}

#endif // B_H
```

b.cpp

```
#include "b.h"
namespace XX {

B::B()
{
}
B::~B()
```

```
{
}

}
```

main.cpp

```
#include <iostream>
#include "a.h"
#include "b.h"

using namespace std;
using namespace XX;
int main()
{
    A a;
    B b;
    return 0;
}
```

2.11.系统 string 类

除了使用字符数组来处理字符串以外，c++引入了字符串类型。可以定义字符串变量。

2.11.1.定义及初始化

```
int main()
{
    string str;
    str = "china";
    string str2 = " is great ";
    string str3 = str2;

    cout<<str<<str2<<endl<<str3<<endl;
    return 0;
}
```

2.11.2.类型大小

```
cout<<"sizeof(string) = "<<sizeof(string)<<endl;
cout<<"sizeof(str)    = "<<sizeof(str)<<endl;
```

2.11.3.常用运算

2.11.3.1.赋值

```
string str3 = str2;
```

2.11.3.2.加法

```
string combine = str + str2;
cout<<combine<<endl;
```

2.11.3.3.关系

```
string  s1 = "abcdeg";
string  s2 = "12345";
```

```
if(s1>s2)
    cout<<"s1>s2"<<endl;
else
    cout<<"s1<s2"<<endl;

string s3 = s1-s2;
cout<<s3<<endl;
```

2.11.4.常见的成员函数

2.11.4.1.下标操作

```
char & operator[](int n) ;
```

2.11.4.2.求串大小

```
int size();
```

2.11.4.3.返回 c 串

```
char *c_str();
```

2.11.4.4.查找

```
int find(char c, int pos = 0);
int find(char * s, int pos = 0);
//返回下标值，没有找到返回-1，默认从 0 下标开找
```

2.11.4.5.删除

```
string &erase(int idx=0，int n = npos);
//作用是删除从 idx 开始，往后数 n 位的字符串。
```

2.11.4.6.交换 swap

```
void swap(stirng &s2);
```

2.11.5.string 类型数组

```
string sArray[10] = {
    "0",
    "1",
    "22",
    "333",
    "4444",
    "55555",
    "666666",
    "7777777",
    "88888888",
    "999999999",
};

for(int i=0; i<10; i++)
{
    cout<<sArray[i]<<endl;
}
```

string 数组是高效的，如果用二维数组来存入字符串数组的话，则容易浪费空间，此时列数是由最长的字符串决定。如果用二级指针申请堆空间，依据大小申请相应的空间，虽然

解决了内存浪费的问题，但是操作麻烦。用 string 数组存储，字符串数组的话，效率即高又灵活。

2.12.C++之父给 C 程序员的建议

1、在 C++中几乎不需要用宏 ,用 const 或 enum 定义显式的常量 ,用 inline 避免函数 调用的额外开销，用模板去刻画一族函数或类型，用 namespace 去避免命名冲突。

2、不要在你需要变量之前去声明，以保证你能立即对它进行初始化。

3、不要用 malloc,new 运算会做的更好。

4、避免使用 void*、指针算术、联合和强制，大多数情况下，强制都是设计错误的指示器。

5、尽量少用数组和 C 风格的字符串，标准库中的 string 和 vector 可以简化程序。

6、更加重要的是，试着将程序考虑为一组由类和对象表示的相互作用的概念，而不是一堆数据结构和一些可以拨弄的二进制。

2.13.练习

2.13.1.格式输出时钟

用 cout 的格式控制，显示一个时钟，格式 00：00：00

2.13.2.string数组的使用

读字符串 char buf[100] = “xxxx:yyyy:zzzz:aaaa:bbb”.按:进行分解到，string 数组中去。

3.封装(Encapsulation)

3.1.封装

3.1.1.从struct说起

当单一变量无法完成描述需求的时候，结构体类型解决了这一问题。可以将多个类型**打包**成一体，形成新的类型。这是 c 语言中**封装**的概念。

但是，新类型并不包含，对数据类的操作。所的有操作都是通过函数的方式，去其进行封装。

struct Date

```
#include <iostream>

using namespace std;
struct Date
{
    int year;
    int month;
    int day;
};
void init(Date &d)
{
    cout<<"year,month,day:"<<endl;
    cin>>d.year>>d.month>>d.day;
}

void print(Date & d)
{
    cout<<"year  month   day"<<endl;
    cout<<d.year<<":"<<d.month<<":"<<d.day<<endl;
}

bool isLeapYear(Date & d)
{
    if((d.year%4==0&& d.year%100 != 0) || d.year%400 == 0)
        return true;
    else
        return false;
}

int main()
{
    Date d;
    init(d);
    print(d);
    if(isLeapYear(d))
        cout<<"leap year"<<endl;
    else
        cout<<"not leap year"<<endl;


    return 0;
}
```

struct Stack

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

using namespace std;

struct Stack
{
    int space[1024];
    int top;
};

void init(Stack &s)
{
    memset(s.space,0,sizeof(s.space));
    s.top = 0;
}

bool isEmpty(Stack &s)
{
    return s.top == 0;
}
bool isFull(Stack &s)
{
    return s.top == 1024;
}

void push(Stack& s,int data)
{
    s.space[s.top++] = data;
}
int pop(Stack &s)
{
    return s.space[--s.top];
}

int main()
{
    Stack s;
    init(s);
    if(!isFull(s))
        push(s,10);
    if(!isFull(s))
        push(s,20);
    if(!isFull(s))
        push(s,30);
    if(!isFull(s))
        push(s,40);
    if(!isFull(s))
        push(s,50);

    while(!isEmpty(s))
        cout<<pop(s)<<endl;


    return 0;
```

```
}
```

3.1.2.封装

封装，可以达到，对外提供接口，屏蔽数据，对内开放数据。

比如我们用 struct 封装的类，即知其接口，又可以直接访问其内部数据，这样却没有达到信息隐蔽的功效。而 class 则提供了这样的功能，屏蔽内部数据，对外开放接口。

struct 中所有行为和属性都是 public 的(默认)。C++中的 class 可以指定行为和属性的访问方式，默认为 pirvate。

访问属性	属性	对象内部	对象外部
public	公有	可访问	可访问
protected	保护	可访问	不可访问
private	私有	可访问	不可访问

3.1.3.用class去封装带行为的类

class 封装的本质，在于将**数据和行为**，绑定在一起然后通过对象来完成操作。

class Date

```
#include <iostream>

using namespace std;
class Date
{
public:
    void init(Date &d);
    void print(Date & d);
    bool isLeapYear(Date & d);

private:
    int year;
    int month;
    int day;
};
void Date::init(Date &d)
{
    cout<<"year,month,day:"<<endl;
    cin>>d.year>>d.month>>d.day;
}

void Date::print(Date & d)
{
    cout<<"year  month   day"<<endl;
    cout<<d.year<<":"<<d.month<<":"<<d.day<<endl;
}

bool Date::isLeapYear(Date & d)
```

```
{
    if((d.year%4==0&& d.year%100 != 0) || d.year%400 == 0)
        return true;
    else
        return false;
}

int main()
{

    Date d;
    d.init(d);
    d.print(d);
    if(d.isLeapYear(d))
        cout<<"leap year"<<endl;
    else
        cout<<"not leap year"<<endl;


    return 0;
}
```

Date 类 访问自己的成员 可以**不需要能过传引用**的方式

```
#include <iostream>

using namespace std;
class Date
{
public:
    void init();
    void print();
    bool isLeapYear();

private:
    int year;
    int month;
    int day;
};
void Date::init()
{
    cout<<"year,month,day:"<<endl;
    cin>>year>>month>>day;
}

void Date::print()
{
    cout<<"year  month   day"<<endl;
    cout<<year<<":"<<month<<":"<<day<<endl;
}

bool Date::isLeapYear()
{
    if((year%4==0&& year%100 != 0) || year%400 == 0)
        return true;
    else
        return false;
}
```

```
int main()
{

    Date d;
    d.init();
    d.print();
    if(d.isLeapYear())
        cout<<"leap year"<<endl;
    else
        cout<<"not leap year"<<endl;


    return 0;
}
```

class Stack

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

using namespace std;

class Stack
{
public:
    void init();
    bool isEmpty();
    bool isFull();
    void push(int data);
    int pop();

private:
    int space[1024];
    int top;
};

void Stack::init()
{
    memset(space,0,sizeof(space));
    top = 0;
}

bool Stack::isEmpty()
{
    return top == 0;
}
bool Stack::isFull()
{
    return top == 1024;
}

void Stack::push(int data)
{
    space[top++] = data;
}
int Stack::pop()
```

```
{
    return space[--top];
}

int main()
{
    Stack s;
    s.init();
    if(!s.isFull())
        s.push(10);
    if(!s.isFull())
        s.push(20);
    if(!s.isFull())
        s.push(30);
    if(!s.isFull())
        s.push(40);
    if(!s.isFull())
        s.push(50);

    while(!s.isEmpty())
        cout<<s.pop()<<endl;


    return 0;
}
```

3.2.练习封装

3.2.1.封装自己的list

提示：

```
typedef struct node
{
    int data; //数据域
    struct node *next; //指针域
}Node;

class List
{
public:
    void initList();
    void insertList();
    void deleteNode(Node * pfind);
    Node * searchList(int find);
    void sortList();
    void destroy();

private:
    Node * head;
};
```

4.类与对象(Class &&object)

4.1.stack 声明与定义

引入构造器实现 自定义 栈大小

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

using namespace std;

class Stack
{
public:
    Stack(int size=1024);
    ~Stack();
    void init();
    bool isEmpty();
    bool isFull();
    void push(int data);
    int pop();

private:
    int* space;
    int top;
};

Stack::Stack(int size)
{
    space = new int[size];
    top = 0;
}

Stack::~Stack()
{
    delete []space;
}

//void Stack::init()
//{
//    memset(space,0,sizeof(space));
//    top = 0;
//}

bool Stack::isEmpty()
{
    return top == 0;
}
bool Stack::isFull()
{
    return top == 1024;
}

void Stack::push(int data)
{
    space[top++] = data;
```

```
}
int Stack::pop()
{
    return space[--top];
}

int main()
{
//    Stack s;
    Stack s(100);
//    s.init();
    if(!s.isFull())
        s.push(10);
    if(!s.isFull())
        s.push(20);
    if(!s.isFull())
        s.push(30);
    if(!s.isFull())
        s.push(40);
    if(!s.isFull())
        s.push(50);

    while(!s.isEmpty())
        cout<<s.pop()<<endl;


    return 0;
}
```

4.2.构造器（Constructor）

4.2.1.定义及意义

```
class 类名
{
    类名(形式参数)
        构造体
}
```

```
class  A
{
    A(形参)
    {}
}
```

在类对象创建时，自动调用，完成类对象的初始化。尤其是动态堆内存的申请。

规则:

1 在对象创建时自动调用,完成初始化相关工作。

2 无返回值，与类名同，

3 可以重载，可默认参数。

4 默认无参空体，一经实现，默认不复存在。

```
Stack::Stack(int size)
{
    space = new int[size];
    top = 0;
}
```

4.2.2.参数初始化表

```
Stack::Stack(int size)
    :space(new int[size]),top(0)
{
}
```

下面代码中有错误吗？

```
#include <iostream>
#include <string.h>

using namespace std;

class A
{
public:
    A(char * ps)
        :name(ps),len(strlen(name.c_str())){}

    void dis()
    {
        cout<<len<<endl;
    }

private:
    int len;
    string name;

};


int main()
{
    A a("china");
    a.dis();
    return 0;
}
```

◆结论：

1，此格式只能用于类的构造函数。

2，初始化列表中的初始化顺序，与声明顺序有关，与前后赋值顺序无关。

3，必须用此格式来初始化非静态 const 数据成员(c++98)。

4，必须用此格式来实始化引用数据。

4.3.析造器(Destructor)

4.3.1.对象销毁时期

1，栈对象离开其作用域。

2，堆对象被手动 delete.

4.3.2.析构器的定义及意义

```
class 类名
{
    ~类名()
        析造体
}
```

```
 class A
 {
    ~A()
    {}
 }
```

在类对像销毁时，自动调用，完成对象的销毁。尤其是类中己申请的堆内存的释放.

规则:

1 对象销毁时，自动调用。完成销毁的善后工作。

2 无返值，与类名同，无参。不可以重载与默认参数。

3 系统提供默认空析构器，一经实现，不复存在。

```
Stack::~Stack()
{
    delete []space;
}
```

4.3.3.小结

析构函数的作用，并不是删除对象，而在对象销毁前完成的一些清理工作。

4.4.构造与析构小结

```
struct Student
{
    char *name;
    int age;
};

int main()
{
    Student stu;
    stu.name = (char*)malloc(100);
```

```
    strcpy(stu.name,"hehe");
    stu.age = 100;

    free(stu.name);



    Student *ps = new Student;
    ps->name = (char*)malloc(100);
    strcpy(ps->name,"bob");
    ps->age = 23;
    free(ps->name);
    free(ps);

}
```

4.5.多文件编程

通常我们将类的声明，放到.h 文件中去，而将实现放到.cpp 中去。

4.6.拷贝构造(Copy contructor)

4.6.1.拷贝构造的定义及意义

由己存在的对象，创建新对象。也就是说新对象，不由构造器来构造，而是由拷贝构造器来完成。拷贝构造器的格式是固定的。

```
class 类名
{
    类名(const 类名 & another)
        拷贝构造体
}
```

```
class A
{
    A(const A & another)
    {}
}
```

规则：

1 系统提供默认的拷贝构造器。一经实现，不复存在。

2 系统提供的时等位拷贝，也就是所谓的浅浅的拷贝。

3 要实现深拷贝，必须要自定义。

4.6.2.拷贝构造发生的时机。

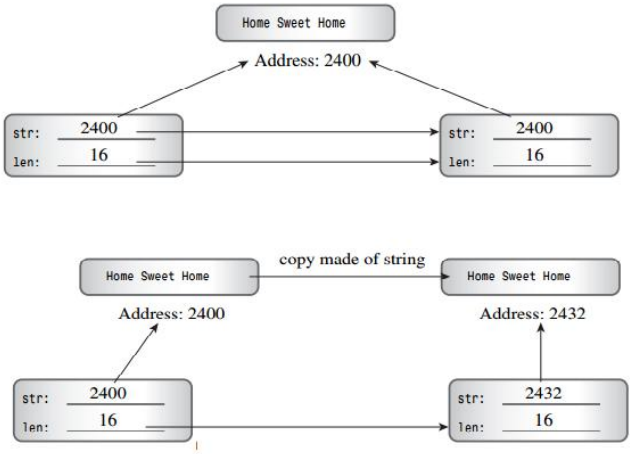
1，制作对象的副本。

2，以对象作为参数和返回值。

4.6.3.深拷贝与浅拷贝

系统提供默认的拷贝构造器，一经定义不再提供。但系统提供的默认拷贝构造器是等位拷贝，也就是通常意义上的浅拷贝。如果类中包含的数据元素全部在栈上，浅拷贝也可以满足需求的。但如果堆上的数据，则会发生多次析构行为。

图示：




```
#include <iostream>
#include <string.h>

using namespace std;

class A
{
public:
    A(int d,char *p):data(d)
    {
        pd = new char[strlen(p)+1];
        strcpy(pd,p);
    }
    ~A()
    {
        delete[]pd;
    }
    A(const A& another)
    {
        pd = new char[strlen(another.pd)+1];
        strcpy(pd,another.pd);
    }

    void dis()
    {
        cout<<data<<endl;
        cout<<pd<<endl;
    }
```

```
private:
    int data;
    char *pd;
};


int main()
{
//    A a(20);
//    a.dis();
//    A b(a);
//    b.dis();
//    A c = b;
//    c.dis();

    A a(20,"china");
    a.dis();
    A b(a);
    b.dis();
    A c = b;
    c.dis();
    return 0;
}
```

4.7.this 指针

4.7.1.意义

系统在创建对象时，默认生成的指向当前对象的指针。这样作的目的，就是为了带来方便。

4.7.2.作用

1，避免构造器的入参与成员名相同。

2，基于 this 指针的自身引用还被广泛地应用于那些支持**多重串联调用**的函数中。比如连续赋值。

```
#include <iostream>

using namespace std;
class Stu
{
public:
    Stu(string name, int age) // :name(name),age(age)
    {
        this->name = name;
        this->age = age;
    }

    Stu & growUp()
    {
        this->age++;
        return *this;  // return this; ??
    }
    void display()
    {
        cout<<name<<" : "<<age<<endl;
```

```
    }

private:
    string name;
    int age;
};

int main()
{
    Stu s("wangguilin",30);
    s.display();
    s.growUp().growUp().growUp().growUp().growUp();
    s.display();
    return 0;
}
```

4.8.赋值运算符重载(Operator=)

4.8.1.发生的时机

用一个已有对象，给另外一个已有对象赋值。两个对象均已创建结束后，发生的赋值行为。

4.8.2.定义

```
类名
{
    类名& operator=(const 类名& 源对象)
        拷贝体
}
```

```
class A
{
    A& operator=(const A& another)
    {
        //函数体
        return *this;
    }
};
```

4.8.3.规则

1 系统提供默认的赋值运算符重载，一经实现，不复存在。

2 系统提供的也是等位拷贝，也就浅拷贝，会造成内存泄漏，重析构。

3 要实现深深的赋值，必须自定义。

4 自定义面临的问题有三个：

1，自赋值

2，内存泄漏

3，重析构。

5 返回引用，且不能用 const 修饰。a = b = c => (a+b) = c

4.9.返回栈上引用与对象

4.9.1.c语言返回栈变量

返回的过程产生了”中间变量”作为纽带。

```
#include <stdio.h>


int func()
{
    int a = 4;
    return a;
}

int main(void)
{
    int i = 3;
    i = func();
    return 0;
}
```

不管是返回指针还是返回值，return 将 return 之后的值存到 eax 寄存器中，回到父函数再将返回的值赋给变量。

4.9.2.c++返回栈对象

```
class A
{
public:
    A(){
        cout<<this<<"  constructor"<<endl;
    }
    A(const A &other)
    {
        cout<<this<<" cp contructor from "<<&other<<endl;
    }
    A & operator=(const A &other)
    {
        cout<<this<<" operator =  "<<&other<<endl;
    }

    ~A()
    {

        cout<<this<<"  destructor"<<endl;
    }
```

```
};
```

4.9.2.1.本质推演

a 传值：发生拷贝

```
void foo(A a)
{}
int main()
{
    A a;
    foo(a);
    return 0;
}
```

b 传引用 没有发生拷贝

```
void foo(A& a)
{}
int main()
{
    A a;
    foo(a);
    return 0;
}
```

c 返回对象

```
A foo(A& a)
{
    return a;
}

int main()
{
    A a;
    foo(a);
    return 0;
}
```

在main的栈上事先开辟了一个临时空间 ,把这个空间的地址隐式的转到foo函数栈上。然后，把 a 内的东西，拷贝到临时空间中。 所以发生一次构造，一次拷贝，两次析构。

测试：

```
A foo(A& a)
{
    cout<<"in foo :"<<(void*)&a<<endl;
    return a;
}

int main()
{
    A a;
    A t = foo(a);
    cout<<"in main:"<<(void*)&t<<endl;
    return 0;
}
```

此时 main 函数中产生的临时空间 ,由 t 来取而代之。所以也发生一次构造 ,一次拷贝，

两次析构。此时 t 的地址，同 a 的地址不同。

```
A foo(A& a)
{
    cout<<"in foo :"<<(void*)&a<<endl;
    return a;
}

int main()
{
    A a;
    A t ;
    t = foo(a);
    cout<<"in main:"<<(void*)&t<<endl;
    return 0;
}
```

此时 main 栈上通过拷贝构造产生了中间变量，中间变量向 t 发生了赋值。

4.9.2.2.本质结论

以上 windows 和 linux 相同，以下 linux 作了更深层次的优化，

```
A foo()
{
    A b;
    cout<<"in foo :"<<(void*)&b<<endl;
    return b;
}

int main()
{
    A t = foo();
    cout<<"in main:"<<(void*)&t<<endl;
    return 0;
}
```

此时发生了一次构造，一次析构。也就是 main 中的 t 取代了临时空间，而 b 的构造完成了 t 的构造。所在完成了一次构造，一次析构。

此时 t 的地址，同 b 的地址相同。

```
A func()
{
    A b;
    cout<<"in func &b"<<&b<<endl;
    return b;
}


int main()
{

    A t ;
    cout<<"int main &t"<<&t<<endl;
    t = func();
    return 0;
}
```

此时发生了两次构造，一次赋值，两次析构，其中 b 的构造，完成了 main 函数中临时空间的构造，构造的临时对象作为赋值运算符重载的参数传入。然后发生赋值。两次析构分别是临时空间和 t 的析构。

4.9.3.c++返回栈对象引用

返回栈对象的引用，多用于产生串联应用。比如连等式。 **栈对象是不可以返回引用的。除非，函数的调用者返回自身对象。**

比如：

```
MyString & MyString::operator=(const MyString & another)
{
    if(this == &another)
        return *this;
    else
    {
        delete []this->_str;
        int len = strlen(another._str);
        this->_str = new char[len+1];
        strcpy(this->_str,another._str);
        return *this;
    }
}
```

提高：非要返回栈引用会发生什么

```
#include <iostream>
#include <typeinfo>

using namespace  std;

class A
{
public:
    A(){
        cout<<this<<"  constructor"<<endl;
    }
    A(const A &a)
    {
        cout<<this<<"cp contructor from "<<&a<<endl;
    }
    ~A()
    {

        cout<<this<<"  destructor"<<endl;
    }
};


const A& func()
{
```

```
    A b;
    cout<<"in func &a"<<&b<<endl;
    return b;
}


int main()
{

    A t = func();
    cout<<"int main &t"<<&t<<endl;

    return 0;
}
```

返回的引用，完成了一次拷贝，但是被拷贝的对象，已经析构。结果是未知的，所以不要返回栈上的引用。

4.10.案例系统 string 与 MyString

4.10.1.string的使用

```
int main()
{
//    string s("china");
    string s = "china";
    string s2(s);
    string s3 = s2;

    string s4;
    s4 = s3;
}
```

4.10.2.MyString 声明

```
#ifndef MYSTRING_H_
#define MYSTRING_H_
#include <stddef.h>
#include <iostream>

class MyString {
public:
    MyString(const char *str=NULL);
    MyString(const MyString & other);
    MyString & operator=(const MyString & another);
    MyString  operator+(const MyString & other);

    bool operator==(const MyString &other);
    bool operator>(const MyString &other);
    bool operator<(const MyString &other);
    char& operator[](int idx);
    void dis();
    virtual ~MyString();

private:
    char * _str;
```

```
};

#endif /* MYSTRING_H_ */
```

4.10.3.构造

```
MyString::MyString(const char *str) {

    if(str == NULL)
    {
        _str = new char[1];
        *_str = '\0';
    }
    else
    {
        int len = strlen(str);
        _str = new char[len+1];
        strcpy(_str,str);
    }
}
```

4.10.4.析构

```
MyString::~MyString() {
    delete []_str;
}
```

4.10.5.拷贝构造（深拷贝）

```
MyString::MyString(const MyString & other)
{
    int len = strlen(other._str);
    this->_str = new char[len+1];
    strcpy(this->_str,other._str);
}
```

4.10.6.赋值运算符重载

```
MyString & MyString::operator=(const MyString & another)
{
    if(this == &another)
        return *this;
    else
    {
        delete []this->_str;
        int len = strlen(another._str);
        this->_str = new char[len+1];
        strcpy(this->_str,another._str);
        return *this;
    }
}
```

4.10.7.加法运算符重载

```
MyString  MyString::operator+(const MyString & other)
{
```

```
    int len = strlen(this->_str) + strlen(other._str);

    MyString str;
    delete []str._str;

    str._str = new char[len+1];
    memset(str._str,0,len+1);

    strcat(str._str,this->_str);
    strcat(str._str,other._str);

    return str;
}
```

4.10.8.关系运算符重载

```
bool MyString::operator==(const MyString &other)
{
    if(strcmp(this->_str,other._str) == 0)
        return true;
    else
        return false;
}

bool MyString::operator>(const MyString &other)
{
    if(strcmp(this->_str,other._str) > 0)
        return true;
    else
        return false;
}
bool MyString::operator<(const MyString &other)
{
    if(strcmp(this->_str,other._str) < 0)
        return true;
    else
        return false;
}
```

4.10.9.[]运算符重载

```
char& MyString::operator[](int idx)
{
    return _str[idx];
}
```

4.10.10.测试

```
#include <iostream>
#include "mystring.h"

using namespace std;

int main()
{
```

```
//  MyString s = "china";
    MyString s("china");  //构造器
    s.dis();

//  MyString s2 = s;
    MyString s2(s);       //拷贝构造器
    s2.dis();

    MyString s3;
    s3 = s2 = s;          //赋值运算符重载
    s3.dis();

    MyString s4;
    s4 = "america";       //"america" 无名对象的构造器，赋值运算符重载
    return 0;
}
```

4.11.课常练习

4.11.1.实现钟表类

属性：时，分，秒

行为：run()　在屏幕上实现电子时钟 13：34：45 每隔一秒更新一个显示。

4.11.2.分析：

构造时，初始化为当前系统时间，然后每隔一秒，刷屏。

4.11.3.代码

```
#include <iostream>
#include <time.h>
#include <unistd.h>

using namespace std;


class Clock
{
public:
    Clock()
    {
        time_t t = time(NULL);
        tm local = * localtime(&t);
        _hour = local.tm_hour;
        _minute = local.tm_min;
        _second = local.tm_sec;
    }
    void run()
    {
        for(;;)
        {
            tick();
            show();
        }
```

```
    }

private:
    void tick()
    {
        sleep(1);
        if(++_second == 60)
        {
            _second = 0;
            if(++_minute == 60)
            {
                _minute = 0;
                if(++_hour == 24)
                {
                    _hour = 0;
                }
            }
        }
    }

    void show()
    {
        system("cls");
        cout<<_hour<<":"<<_minute<<":"<<_second<<endl;
    }

    int _hour;
    int _minute;
    int _second;
};

int main()
{
    Clock c;
    c.run();
    return 0;
}
```

4.12.栈和堆上的对象及对象数组

4.12.1.引例

```
#include <iostream>

using namespace std;

class Stu
{
public:
    Stu(string n):_name(n){}
    void dis()
    {
        cout<<_name<<endl;
    }
```

```
private:
    string _name;
};

int main()
{
//  Stu s;  //没有无参构造器
//  Stu s[5]= {Stu("zhangsan"),Stu("lisi")}; 不能指定个数，或部分初始化，则会报错。
    Stu s[]= {Stu("zhangsan"),Stu("lisi")};

//  Stu * ps = new Stu[4]{Stu("zhangsan")};
//  C11 中支持此种初始化方法，但必须对指定的类个数初始化,否则会报错。
    Stu * ps = new Stu[1]{Stu("zhangsan")};
    return 0;
}
```

4.12.2.用new 和delete生成销毁堆对象

new 一个堆对象，会自动调用构造函数，delete 一个堆对象对自动调用析构函数。这同 c 中 malloc 和 free 不同的地方。

4.12.3.栈对象数组

如果生成的数组，未初始化，则必调用无参构造器。或手动调用带参构造器。

4.12.4.堆对象数组

如果生成的数组，未初始化，则必调用无参构造器。或手动调用带参构造器。

4.12.5.结论

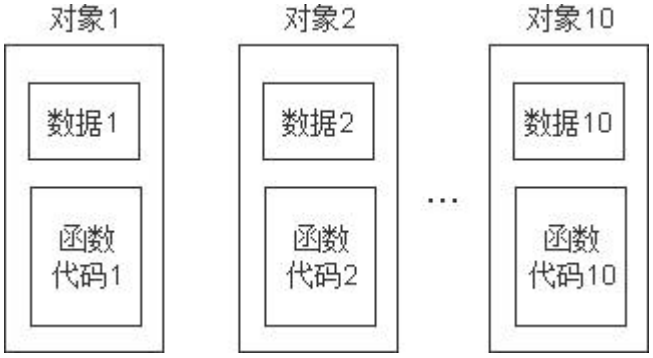
构造器无论是重载还是默认参数，一定要把系统默认的无参构造器包含进来。不然生成数组的时候，可能会有些麻烦。

4.13.成员函数的存储方式

4.13.1.类成员可能的组成

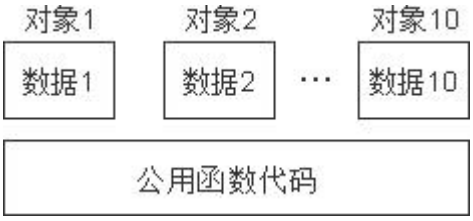
用类去定义对象时，系统会为每一个对象分配存储空间。如果一个类包括了数据和函数，要分别为数据和函数的代码分配存储空间。

按理说，如果用同一个类定义了 10 个对象，那么就需要分别为 10 个对象的数据和函数代码分配存储单元。



4.13.2.类成员实际的组成

能否只用一段空间来存放这个共同的函数代码段，在调用各对象的函数时，都去调用这个公用的函数代码。



显然，这样做会大大节约存储空间。C++编译系统正是这样做的，因此每个对象所占用的存储空间只是该对象的数据部分所占用的存储空间，而不包括函数代码所占用的存储空间。

```
class Time
{
public:
    void dis()
    {
        cout<<hour<<minute<<sec<<endl;
    }
private:
   int hour;
   int minute;
   int sec;
};

int main()
{
    cout<<sizeof(Time)<<endl;  //12
    //一个对象所占的空间大小只取决于该对象中数据成员所占的空间，而与成员函数无关
    return 0;
}
```

4.13.3.调用原理

所的有对象都调用共用的函数代码段，如何区分的呢，执行不同的代码可能有不用的结

果。原因，c++设置了 this 指针，this 指针指向调用该函数的不同对象。当 t 调用 dis()函数时，this 指向 t。当 t1 调用 dis()函数时，this 指向 t1;

```
class Time
{
public:
    Time(int h, int m,int s)
        :hour(h),minute(m),sec(s){}
    void dis()  //void dis(Time *p)
    {
        cout<<"this="<<this<<endl;
        cout<<this->hour<<":"<<this->minute<<":"<<this->sec<<endl;
    }
private:
   int hour;
   int minute;
   int sec;
};

int main()
{
   Time t(1,2,3);
   Time t2(2,3,4);
   t.dis();  //等价于 t.dis(&t)
   t2.dis();
    return 0;
}
```

4.13.4.注意事项

1，不论成员函数在类内定义还是在类外定义，成员函数的代码段都用同一种方式存储。

2，不要将成员函数的这种存储方式和 inline(内置)函数的概念混淆。inline 的逻辑意义是将函数内嵌到调用代码处，减少压栈与出栈的开支。

3，应当说明，常说的“某某对象的成员函数”，是从逻辑的角度而言的，而成员函数的存储方式，是从物理的角度而言的，二者是不矛盾的。类似于二维数组是逻辑概念 ,而物理存储是线性概念一样。

4.14.const 修饰符

4.14.1.常数据成员

const 修饰类的成员变量，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(c11 中支持类中实始化)。可被 const 和非 const 成员函数调用，而不可以修改。

```
class A
{
public:
    A():iValue(199){}

private:
    const int iValue;
```

```
};
```

4.14.2.常成员函数

4.14.2.1.const 修饰函数的意义

承诺在本函数内部不会修改类内的数据成员，不会调用其它非 const 成员函数

4.14.2.2.const 修饰函数位置

const 修饰函数放在，声明这后，实现体之前，大概也没有别的地方可以放了。

```
void dis() const
{}
```

4.14.2.3.const 构成函数重载

```
class A
{
public:
    A():x(199),y(299){}
    void dis() const  //const 对象调用时，优先调用
    {
        //input(); 不能调用 非 const 函数，因为本函数不会修改，无法保证所调的函数也不会修改
        cout<<"x "<<x<<endl;
        cout<<"y "<<y<<endl;
        //y =200;  const 修饰函数表示承诺不对数据成员修改。
    }
    void dis() //此时构成重载,非 const 对象时，优先调用。
    {
        y = 200;
        input();
        cout<<"x "<<x<<endl;
        cout<<"y "<<y<<endl;
    }
    void input()
    {
        cin>>y;
    }

private:
    const int x;
    int y;

};

int main()
{
    A a;
    a.dis();

//  const A a;
//  a.dis();
    return 0;
}
```

```
```

小结：

1，如果 const 构成函数重载，const 对象只能调用 const 函数，非 const 对象优先调用非 const 函数。

2，const 函数只能调用 const 函数。非 const 函数可以调用 const 函数。

3，类体外定义的 const 成员函数，在定义和声明处都需要 const 修饰符。

4.14.3.常对象

```
const A a;
a.dis();
```

小结：

1，const 对象，只能调用 const 成员函数。

2，可访问 const 或非 const 数据成员，不能修改。

4.15.static 修饰符

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(相比全局变量的优点)**的原则，保证了安全性还可以节省内存。

类的静态成员，属于类，也属于对象，但终归属于类。

4.15.1.类静态数据成员的定义及初始化

4.15.1.1.声明：

```
static 数据类型 成员变量;  //在类的内部
```

4.15.1.2.初始化

```
数据类型 类名::静态数据成员 = 初值; //在类的外部
```

4.15.1.3.调用

```
类名::静态数据成员
类对象.静态数据成员
```

4.15.1.4.案例

中国校园设计的“一塔湖图”

```
#include <iostream>

using namespace std;

class School
{
public:
    static void addLibBooks(string book)
    {
        lib += book;
```

```
    }

public:
    string tower;
    string lake;
    static string lib;

};

//类外实始化
string School::lib = "Beijing lib:";

int main()
{
    School a,b,c,d;
    //static 数据成员，属于类，并不属于对象。
    //存储于 data 区的 rw 段
    cout<<sizeof(a)<<sizeof(b)<<sizeof(c)<<sizeof(d)<<endl;
    //类的方式访问,编译有问题，必须要初始化。
    //在无对象生成的时候，亦可以访问。
    School::lib = "China lib:";
    cout<<School::lib<<endl;

    //lib 虽然属于类，但是目的是为了实现类对象间的共享
    //故对象也是可以访问的。
    cout<<a.lib<<endl;
    cout<<b.lib<<endl;

    //为了搞好图书馆的建设，提设 static 接口
    School::addLibBooks("mao xuan");
    cout<<School::lib<<endl;

    return 0;
}
```

4.15.1.5.小结

1，static 成员变量实现了**同簇类**对象间信息共享。

2，static 成员类外存储，求类大小，并不包含在内。

3，static 成员是命名空间属于类的全局变量，存储在 data 区 rw 段。

4，static 成员使用时必须实始化，且只能类外初始化。

5，可以通过类名访问（无对象生成时亦可），也可以通过对象访问。

4.15.2.类静态成员函数的定义

为了管理静态成员，c++提供了静态函数，以对外提供接口。并静态函数只能访问静态成员。

4.15.2.1.声明

```
static 函数声明
```

4.15.2.2.调用

```
类名::函数调用
类对象.函数调用
```

4.15.2.3.案例

```
#include <iostream>
using namespace std;
class Student
{
public:
    Student(int n,int a,float s):num(n),age(a),score(s){}
    void total()
    {
        count++;
        sum += score;
    }
    static float average();

private:
    int num;
    int age;
    float score;
    static float sum;
    static int count;
};

float Student::sum = 0;
int Student::count = 0;

float Student::average()
{
    return sum/count;
}

int main()
{
    Student stu[3]= {
        Student(1001,14,70),
        Student(1002,15,34),
        Student(1003,16,90)
    };
    for(int i=0; i<3; i++)
    {
        stu[i].total();
    }
    cout<<Student::average()<<endl;
    return 0;
}
```

4.15.2.4.小结

1，静态成员函数的意义，不在于信息共享，数据沟通，而在于管理静态数据成员，完成对静态数据成员的封装。

2，静态成员函数只能访问静态数据成员。原因：非静态成员函数，在调用时 this指针时被当作参数传进。而静态成员函数属于类，而不属于对象，没有 this 指针。

4.15.3.综合案例

4.15.3.1.单例模式

4.15.3.2.cocos 渲染树的模拟

每生成一个节点，自动挂到静态表头上去。

```
#include <iostream>
#include <string.h>

using namespace std;

class Student
{
public:
    Student(string n)
        :name(n)
    {
        if(head == NULL)
        {
           head = this;
           this->next = NULL;
        }
        else{
            this->next = head;
            head= this;
        }
        //可优化

    }

    static void printStudentList();
    static void deleteStudentList();

private:
    string name;
    Student * next;
    static Student * head;
};

void Student::printStudentList()
{
    Student * p = head;
    while(p != NULL)
    {
        cout<<p->name<<endl;
        p = p->next;
    }
}

void Student::deleteStudentList()
{
    Student *p = head;
```

```
    while(head)
    {
        head = head->next;
        delete p;
        p = head;
    }
}

Student * Student::head = NULL;

int main()
{
    string name;
    string postName;
    char buf[1024];
    for(int i=0; i<10; i++)
    {
        name = "stu";
        postName = (itoa(i,buf,10));
        name += postName;

        new Student(name); //Student tmp(name);
    }

    Student::printStudentList();
    Student::deleteStudentList();
    return 0;
}
```

4.16.static const 成员

如果一个类的成员，既要实现共享，又要实现不可改变，那就用 static const 修饰。修饰成员函数，格式并无二异，修饰数据成员。必须要类内部实始化。

```
class A
{
public:
    static const void dis()
    {
        cout<<i<<endl;
    }

private:
    const static int i = 100;
};

int main()
{

    A::dis();
    return 0;
}
```

4.17.指向类成员的指针

在 C++语言中，可以定义一个指针，使其指向类成员或成员函数，然后通过指针来访问类的成员。这包括指向属性成员的指针和指向成员函数的指针。

4.17.1.指向普通变量和函数的指针

```
#include <iostream>

using namespace std;

void func(int a)
{
    cout<<a<<endl;
}

int main()
{
    int a = 100;
    int *p = &a;
    cout<<*p<<endl;

    void (*pf)(int) = func;
    pf(10);
    return 0;
}
```

4.17.2.指向类数据成员的指针

◆定义

```
<数据类型><类名>::*<指针名>
```

◆赋值&初始化

```
<数据类型><类名>::*<指针名>[=&<类名>::<非静态数据成员>]
```

指向非静态数据成员的指针在定义时必须和类相关联，在使用时必须和具体的对象关联。

◆解用引

由于类不是运行时 存在的对象。因此，在使用这类指针时，需要首先指定类的一个对象，然后，通过对象来引用指针所指向的成员。

```
<类对象名>.*<指向非静态数据成员的指针>
<类对象指针>->*<指向非静态数据成员的指针>
```

◆案例

```
#include <iostream>

using namespace std;


class Student
{
public:
```

```
    Student(string n, int nu):name(n),num(nu){}

    string name;
    int num;
};


int main()
{
    Student s("zhangsan",1002);
    Student s2("lisi",1001);

//    string *ps = &s.name;
//    cout<< *ps<<endl;

    string Student::*ps = &Student::name;

    cout<<s.*ps<<endl;
    cout<<s2.*ps<<endl;

    Student *pp = new Student("wangwu",1003);
    cout<<pp->*ps<<endl;

    return 0;
}
```

4.17.3.指向类成员函数的指针

定义一个指向非静态成员函数的指针必须在三个方面与其指向的成员函数保持一致 :参数列表要相同、返回类型要相同、所属的类型要相同

◆定义

```
<数据类型>(<类名>::*<指针名>)(<参数列表>)
```

◆赋值&初始化

```
<数据类型>(<类名>::*<指针名>)(<参数列表>)[=&<类名>::<非静态成员函数>]
```

◆解用引

由于类不是运行时存在的对象。因此，在使用这类指针时，需要首先指定类的一个对象，然后，通过对象来引用指针所指向的成员。

```
(<类对象名>.*<指向非静态成员函数的指针>)(<参数列表>)
(<类对象指针>->*<指向非静态成员函数的指针>)(<参数列表>)
```

◆案例

```
#include <iostream>

using namespace std;
```

```
class Student
{
public:

    Student(string n, int nu):name(n),num(nu){}

    void dis()
    {
        cout<<"name "<<name<<" num "<<num<<endl;
    }

private:
    string name;
    int num;
};


int main()
{
    Student s("zhangsan",1002);
    Student s2("lisi",1001);
    Student *ps = new Student("lisi",1003);

    void (Student::*pf)() =  & Student::dis;

    (s.*pf)();
    (s2.*pf)();
    (ps->*pf)();

    return 0;
}
```

4.17.4.指向类成员指针小结：

与普通意义上的指针不一样。存放的是**偏移量**。

指向非静态成员函数时，必须用类名作限定符，使用时则必须用类的实例作限定符。指向静态成员函数时，则不需要使用类名作限定符。

4.17.5.应用提高

用指向类员函数的指针，实现**更加隐蔽**的接口。

```
#include <iostream>

using namespace std;

class Widget
{
public:
    Widget()
    {
        fptr[0] = &f;
        fptr[1] = &g;
        fptr[2] = &h;
        fptr[3] = &i;
```

```
    }
    void select(int idx, int val)
    {
        if(idx<0 || idx>cnt) return;
           (this->*fptr[idx])(val);
    }
    int count()
    {
        return cnt;
    }

private:
    void f(int val){cout<<"void f() "<<val<<endl;}
    void g(int val){cout<<"void g() "<<val<<endl;}
    void h(int val){cout<<"void h() "<<val<<endl;}
    void i(int val){cout<<"void i() "<<val<<endl;}

    enum{ cnt = 4};
    void (Widget::*fptr[cnt])(int);

};

int main()
{
    Widget w;
    for(int i=0; i<w.count(); i++)
    {
        w.select(i,1);
    }

    return 0;
}
```

4.17.6.指向类静态成员的指针

◆指向类静态数据成员的指针

指向静态数据成员的指针的定义和使用与普通指针相同，在定义时无须和类相关联，在使用时也无须和具体的对象相关联。

◆指向类静态成员函数的指针

指向静态成员函数的指针和普通指针相同，在定义时无须和类相关联，在使用时也无须和具体的对象相关联。

```
#include <iostream>

using namespace std;
class A
{
public:
    static void dis();
    static int data;
```

```
};

void A::dis()
{
    cout<<data<<endl;
}

int A::data = 100;

int main()
{
    int *p = & A::data;

    cout<<*p<<endl;

    void (*pfunc)() = &A::dis;

    pfunc();
    return 0;
}
```

4.18.作业

4.18.1.按需求设计一个圆类

输入圆的半径和圆柱的高，依次输出圆周长、圆面积、圆球表面积、圆柱体积（以空格分隔，π取 3.14）。

4.18.2.编写C++程序完成以下功能：

1)定义一个 Point 类，其属性包括点的坐标，提供计算两点之间距离的方法；

2)定义一个圆形类，

a.其属性包括圆心和半径；

b.创建两个圆形对象，提示用户输入圆心坐标和半径，判断两个圆是否相交，并输出结果。

5.友元(Friend)

采用类的机制后实现了数据的隐藏与封装，类的数据成员一般定义为私有成员，成员函数一般定义为公有的，依此提供类与外界间的**通信接口**。但是，有时需要定义一些函数，这些函数不是类的一部分，但又需要**频繁地访问**类的数据成员，这时可以将这些函数定义为该类的友元函数。除了友元函数外，还有友元类，两者统称为友元。友元的作用是**提高了程序的运行效率**（即减少了类型和安全性检查及调用的时间开销），但它**破坏了类的封装性和隐藏性**，使得非成员函数可以访问类的私有成员。

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。

5.1.同类对象间无私处

```
MyString::MyString(const MyString & other)
{
    int len = strlen(other._str);
    this->_str = new char[len+1];
    strcpy(this->_str,other._str);
}
```

5.2.异类对象间有友员

友元目的本质，是让其它不属于本类的成员(全局函数，其它类的成员函数)，成为类的成员而具备了本类成员的属性。

5.2.1.友元函数

友元函数是可以直接访问类的私有成员的非成员函数。它是定义在类外的普通函数，它不属于任何类，但需要在类的定义中加以声明，声明时只需在友元的名称前加上关键字 friend，其格式如下：

```
friend 类型 函数名(形式参数);
```

一个函数可以是多个类的友元函数，只需要在各个类中分别声明。

5.2.1.1.全局函数作友元函数

```
#include<iostream>
#include<cmath>
using namespace std;
class Point
{
public:
    Point(double xx, double yy)
    {
        x = xx;
```

```
        y = yy;
    }
    void Getxy();
    friend double Distance(Point &a, Point &b);
private:
    double x, y;
};
void Point::Getxy()
{
    cout << "(" << x << "," << y << ")" << endl;
}
double Distance(Point &a, Point &b)
{
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}
int main(void)
{
    Point p1(3.0, 4.0), p2(6.0, 8.0);
    p1.Getxy();
    p2.Getxy();
    double d = Distance(p1, p2);
    cout << "Distance is" << d << endl;
    return 0;
}
```

5.2.1.2.类成员函数作友元函数

```
#include<iostream>
#include<cmath>
using namespace std;

class Point;

class ManagerPoint
{
public:
    double Distance(Point &a, Point &b);
};

class Point
{
public:
    Point(double xx, double yy)
    {
        x = xx;
        y = yy;
    }
    void Getxy();
    friend double ManagerPoint::Distance(Point &a, Point &b);
private:
    double x, y;
};
void Point::Getxy()
{
    cout << "(" << x << "," << y << ")" << endl;
}
```

```
double ManagerPoint::Distance(Point &a, Point &b)
{
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}

int main(void)
{
    Point p1(3.0, 4.0), p2(6.0, 8.0);
    p1.Getxy();
    p2.Getxy();
    ManagerPoint mp;
    float d = mp.Distance(p1,p2);
    cout << "Distance is" << d<< endl;
    return 0;
}
```

补充：

前向声明，是一种不完全型（forward declaration）声明，即只需提供类名(无需提供类实现)即可。正因为是（incomplete type）功能也很有限:

(1)不能定义类的对象。

(2)可以用于定义指向这个类型的指针或引用。

(3)用于声明(不是定义)，使用该类型作为形参类型或者函数的返回值类型。

5.2.2.友元类

友元类的**所有成员函数**都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包括私有成员和保护成员）。

当希望一个类可以存取另一个类的私有成员时，可以将该类声明为另一类的友元类。定义友元类的语句格式如下：

```
friend class 类名;
```

其中：friend 和 class 是关键字，类名必须是程序中的一个已定义过的类。

例如，以下语句说明类 B 是类 A 的友元类：

```
class A
{
    …
    public:
        friend class B;
    …
};
```

经过以上说明后，类 B 的所有成员函数都是类 A 的友元函数，能存取类 A 的私有成员和保护成员。

```
class A
{
public:
    inline void Test()
    {
    }

private:
    int x ,y;
    friend Class B;
}

class B
{
public:
    inline void Test()
    {
        A a;
        printf("x=%d,y=%d".a.x,a.y);
    }
}
```

5.3.论友元

5.3.1.声明位置

友元声明以关键字 friend 开始，它只能出现在类定义中。因为友元不是授权类的成员，所以它不受其所在类的声明区域 public　private 和 protected 的影响。通常我们选择把所有友元声明组织在一起并放在类头之后.

5.3.2.友元的利弊

友元不是类成员 ,但是它可以访问类中的私有成员。友元的作用在于提高程序的运行效率，但是，它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。不过，类的访问权限确实在某些应用场合显得有些呆板，从而容忍了友元这一特别语法现象。

5.3.3.注意事项

(1) 友元关系不能被继承。

(2) 友元关系是单向的，不具有交换性。若类 B 是类 A 的友元，类 A 不一定是类 B的友元，要看在类中是否有相应的声明。

(3) 友元关系不具有传递性。若类 B 是类 A 的友元，类 C 是 B 的友元，类 C 不一定是类 A 的友元，同样要看类中是否有相应的申明

6.运算符重载(Operator OverLoad)

运算符重载的本质是函数重载。

6.1.重载入门

6.1.1.语法格式

重载函数的一般格式如下：

```
返值类型 operator 运算符名称(形参表列)
{
    重载实体;
}
```

operator 运算符名称 在一起构成了新的函数名。比如

```
const Complex operator+(const Complex &c1,const Complex &c2);
```

我们会说，operator+ 重载了重载了运算符+。

6.1.2.友元重载

```
#include <iostream>

using namespace std;
class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }

    friend const Complex operator+(const Complex &c1,const Complex &c2);

private:
    float _x;
    float _y;
};

const Complex operator+(const Complex &c1,const Complex &c2)
{
    return Complex(c1._x + c2._x,c1._y + c2._y);
}

int main()
{
    Complex c1(2,3);
    Complex c2(3,4);
    c1.dis();
    c2.dis();
//    Complex c3 = c1+c2;
    Complex c3 = operator+(c1,c2);
    c3.dis();
```

```
    return 0;
}
```

6.1.3.成员重载

```
#include <iostream>

using namespace std;
class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }

    friend const Complex operator+(const Complex &c1,const Complex &c2);

    const Complex operator+(const Complex &another);
private:
    float _x;
    float _y;
};

const Complex operator+(const Complex &c1,const Complex &c2)
{
    cout<<"友元函数重载"<<endl;
    return Complex(c1._x + c2._x,c1._y + c2._y);
}
const Complex Complex::operator+(const Complex & another)
{
    cout<<"成员函数重载"<<endl;
    return Complex(this->_x + another._x,this->_y + another._y);
}

int main()
{
    Complex c1(2,3);
    Complex c2(3,4);
    c1.dis();
    c2.dis();
//    Complex c3 = c1+c2;
//    Complex c3 = operator+(c1,c2);
    Complex c3 = c1+c2; //优先调用成员函数重载？？
    c3.dis();
    return 0;
}
```

备注：

◆关于返回值：

```
int a = 3;
int b = 4;
```

```
(a+b) = 100; 这种语法是错的，所以重载函数+的返回值加 const 是来修饰。

string a = “china”,b = “ is china”, c;
(c = a) = b; 此时的语法，是重载= 返回值不需加 const 。
```

所以重载运算符，不要破坏语意。

◆ 关于重载全局和成员

6.1.4.重载规则

（1）C++不允许用户自己定义新的运算符，只能对已有的 C++运算符进行重载。

例如，有人觉得 BASIC 中用“* *”作为幂运算符很方便，也想在 C++中将“* *”定义为幂运算符，用“3* *5”表示 3^5，这是不行的。

（2）C++允许重载的运算符

C++中绝大部分运算符都是可以被重载的。

C++中可被重载的操作符

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	,	->*
->	()	[]	new	delete	new[]	delete[]

不能重载的运算符只有 4 个：

.	成员运算符
.*	成员对象选择符
::	域解析运算符
?:	条件运算符
sizeof	类型大小运算符

前两个运算符不能重载是为了保证访问成员的功能不能被改变，域运算符合 sizeof 运算符的运算对象是类型而不是变量或一般表达式，不具备重载的特征。

（3）重载不能改变运算符运算对象（即操作数）的个数。

如，关系运算符“>”和“<”等是双目运算符，重载后仍为双目运算符，需要两个参数。运算符”+“,”-“,”*“,”&“等既可以作为单目运算符，也可以作为双目运算符，可以分别将它们重载为单目运算符或双目运算符。

（4）重载不能改变运算符的优先级别。

例如”*“和”/“优先级高于”+“和”-“，不论怎样进行重载，各运算符之间的优先级不会改变。有时在程序中希望改变某运算符的优先级，也只能使用加括号的方法强制改变重载运算符的运算顺序。

（5）重载不能改变运算符的结合性。

如，复制运算符”=“是右结合性（自右至左），重载后仍为右结合性。

（6）重载运算符的函数不能有默认的参数

否则就改变了运算符参数的个数，与前面第（3）点矛盾。

（7）重载运算符的运算中至少有一个操作数是自定义类。

重载的运算符必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象（或类对象的引用）。

也就是说，参数不能全部是 C++的标准类型，以防止用户修改用于标准类型数据成员的运算符的性质，如下面这样是不对的：

复制代码 代码如下：

```
int operator + (int a,int b)
{
        return(a-b);
}
```

原来运算符+的作用是对两个数相加，现在企图通过重载使它的作用改为两个数相减。

如果允许这样重载的话，如果有表达式 4+3，它的结果是 7 还是 1 呢？显然，这是绝对要禁止的。

（8）不必重载的运算符（= &）

用于类对象的运算符一般必须重载，但有两个例外，运算符”=“和运算符”&“不必用户重载。

复制运算符”=“可以用于每一个类对象，可以用它在同类对象之间相互赋值。因为系统已为每一个新声明的类重载了一个赋值运算符，它的作用是逐个复制类中的数据成员

地址运算符&也不必重载，它能返回类对象在内存中的起始地址。

（9）对运算符的重载，不应该失去其原有的意义

应当使重载运算符的功能类似于该运算符作用于标准类型数据时候时所实现的功能。

例如，我们会去重载”+“以实现对象的相加，而不会去重载”+“以实现对象相减的功能，因为这样不符合我们对”+“原来的认知。

6.2.重载例举

6.2.1.双目运算符例举

6.2.1.1.双目运逢符重载格式

形式

```
 L#R
```

全局函数

```
operator#(L,R);
```

成员函数

```
L.operator#(R)
```

6.2.1.2.重载+=实现

```
#include <iostream>

using namespace std;

class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }
    Complex& operator+=(const Complex &c)
    {
        this->_x += c._x;
        this->_y += c._y;

        return * this;
    }
private:
    float _x;
    float _y;
};

int main()
```

```
{
//    int a = 10, b = 20,c = 30;
//    a += b;
//    b += c;
//    cout<<"a = "<<a<<endl;
//    cout<<"b = "<<b<<endl;
//    cout<<"c = "<<c<<endl;

//    Complex a1(10,0),b1(20,0), c1(30,0);

//    //(1)此时的+=重载函数返回 void
//    a1 += b1;
//    b1 += c1;

//    a1.dis();
//    b1.dis();
//    c1.dis();
//-----------------------------------------------------
//    int a = 10, b = 20,c = 30;
//    a += b += c;

//    cout<<"a = "<<a<<endl;
//    cout<<"b = "<<b<<endl;
//    cout<<"c = "<<c<<endl;

//    Complex a1(10,0),b1(20,0), c1(30,0);


//    //(2)此时重载函数+=返回的是 Complex
//    a1 += b1 += c1;

//    a1.dis();
//    b1.dis();
//    c1.dis();
//-----------------------------------------------------
    int a = 10, b = 20,c = 30;
    (a += b) += c;

    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;

    Complex a1(10,0),b1(20,0), c1(30,0);

//   //(3)此时重载函数+=返回的是 Complex &
//    一定要注意在连等式中,返回引用和返回对象的区别
    (a1 += b1) += c1;

    a1.dis();
    b1.dis();
    c1.dis();
    return 0;
}
```

◆operator-=

```
friend Complex& operator-=(Complex &c1, const Complex & c2)
```

```
{
}
```

6.2.2.单目运算符例举

6.2.2.1.重载单目运算符格式

形式

```
#M  或M#
```

全局函数

```
 operator#(M)
```

成员函数

```
M.operator#()
```

6.2.2.2.重载-运算符

```
#include <iostream>

using namespace std;


class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }
   const Complex operator-(void) const
    {
        return Complex(-_x,-_y);
    }

private:
    float _x;
    float _y;
};

int main()
{
    int n = 10;
    cout<<n<<endl;
    cout<<-n<<endl;
    cout<<n<<endl;
    cout<<-(-n)<<endl;
//    -n = 100;

    Complex c(1,2);
    Complex c2 = -c;

    c2 = -(-c);
```

```
    -c = Complex(3,4); //编译通不过

    c.dis();
    c2.dis();
    c.dis();
    return 0;
}
```

6.2.2.3.重载++（前++）

```
#include <iostream>

using namespace std;
class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }
//    Complex & operator++(void)
//    {
//        _x++;
//        _y++;
//        return *this;
//    }

    friend Complex & operator++(Complex& c);
private:
    float _x;
    float _y;
};
Complex & operator++(Complex& c)
{
    c._x++;
    c._y++;
    return c;
}

int main()
{
    int n = 10;
    cout<<n<<endl;      //10
    cout<<++n<<endl;    //11
    cout<<n<<endl;      //11
    cout<<++++n<<endl;  //13
    cout<<n<<endl;      //13


    Complex c(10,10);
    c.dis();           //10 10
    Complex c2=++c;
    c2.dis();          //11 11
    c.dis();           //11 11
```

```
    c2 = ++++c;
    c2.dis();          //13 13
    c.dis();           //13 13


    return 0;
}
```

6.2.2.4.重载++(后++)

```
#include <iostream>

using namespace std;


class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }

//    const Complex operator++(int) //哑元
//    {
//        Complex t = *this;
//        _x++;
//        _y++;
//        return t;
//    }

    friend const Complex operator++(Complex &c,int);

private:
    float _x;
    float _y;
};

const Complex operator++(Complex &c,int)
{
    Complex t(c._x,c._y);
    c._x++;
    c._y++;
    return t;
}


int main()
{
    int n = 10;
    cout<<n<<endl;      //10
    cout<<n++<<endl;    //10
    cout<<n<<endl;      //11
```

```
//    cout<<n++++<<endl;  //13 后++表达式不能连用
    cout<<n<<endl;      //11

    Complex c(10);
    c.dis();;
    Complex c2 = c++;
    c2.dis();
    c.dis();;

//    c2 = c++++;

//    c2.dis();
    c.dis();

    return 0;
}
```

6.2.3.流输入输出运算符重载

函数形式

```
istream & operator>>(istream &,自定义类&);
ostream & operator<<(ostream &,自定义类&);
```

通过友元来实现，避免修改 c++的标准库。

◆operator<< vs operator>>

```
 class Complex
{
public:
    Complex(float x=0, float y=0)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }

    friend ostream & operator<<(ostream &os, const Complex & c);
    friend istream & operator>>(istream &is, Complex &c);
    //成员的话必须保证左值为该类的对象
private:
    float _x;
    float _y;
};

ostream & operator<<(ostream &os, const Complex & c)
{
    os<<"("<<c._x<<","<<c._y<<")";
    return os;
}
istream & operator>>(istream &is, Complex &c)
{
    is>>c._x>>c._y;
    return is;
```

```
}



int main()
{
    Complex c(2,3);
    cout<<c<<endl;
    cin>>c;
    cout<<c<<endl;

    return 0;
}
```

补充 MyString 类的流输入与输出

```
friend ostream & operator<<(ostream & os,const MyString &outStr);
friend istream & operator>>(istream &in, MyString & inStr);
```

```
ostream & operator<<(ostream & os,const MyString &outStr)
{
    os<<outStr._str;
    return os;
}

istream & operator>>(istream &in, MyString & inStr)
{
    delete [] inStr._str;
    char buf[1024];

    scanf("%s",buf); //scanf  fgets  in.getline(buf,1024);
    int len;
    len = strlen(buf);

    inStr._str = new char[len+1];
    strcpy(inStr._str,buf);

    return in;
}
```

6.3.运算符重载小结

6.3.1.重载格式

操作符与函数调用符之间的关系

操作符	表达式	成员函数表示	友员函数表示
前缀单目操作符@	@a	(a).operator@()	operator@(a)
后缀单目操作符@	a@	(a).operator@(0)	operator@(a, 0)
双目操作符@	a@b	(a).operator@(b)	operator@(a, b)
赋值操作符=	a=b	(a).operator=(b)	无
下标操作符[]	a[b]	(a).operator[](b)	无
类成员访问操作符->	a->	(a).operator->()	无

6.3.2.不可重载的运算符

```
.          （成员访问运算符）
.*         （成员指针访问运算符）
::         （域运算符）
sizeof     （长度运算符）
?:         （条件运算符）
```

6.3.3.只能重载为成员函数的运算符

```
 =     赋值运算符
[]     下标运算符
()     函数运算符
->     间接成员访问
->*    间接取值访问
```

6.3.4.常规建议

运算符	建议使用
所有的一元运算符	成员
+= -= /= *= ^= &= != %= >>= <<=	成员
其它二员运算符	非成员

6.3.5.友元还是成员？

◆引例

假设，我们有类 Sender 类和 Mail 类，实现发送邮件的功能。

```
Sender sender; Mail mail;
sender<< mail;
```

sender 左操作数，决定了 operator<<为 Sender 的成员函数，而 mail 决定了 operator<<要作 Mail 类的友员。

```
#include <iostream>

using namespace std;

class Mail;
class Sender
{
public:
    Sender(string s):_addr(s){}
    Sender& operator<<(const Mail & mail); //成员

private:
    string _addr;
};

class Mail
{
public:
    Mail(string _t,string _c ):_title(_t),_content(_c){}
    friend  Sender& Sender::operator<<(const Mail & mail);
    //友元
private:
    string _title;
    string _content;
};

 Sender& Sender::operator<<(const Mail & mail)
{
    cout<<"Address:"<<_addr<<endl;
    cout<<"Title  :"<<mail._title<<endl;
    cout<<"Content:"<<mail._content<<endl;
    return *this;
}

int main()
{
    Sender sender("guilin_wang@163.com");
    Mail   mail("note","meeting at 3:00 pm");
    Mail   mail2("tour","One night in beijing");
    sender<<mail<<mail2;
    return 0;
}
```

◆结论：

1，一个操作符的左右操作数不一定是相同类型的对象，这就涉及到将该操作符函数定义为谁的友元，谁的成员问题。

2，一个操作符函数，被声明为哪个类的成员，取决于该函数的调用对象(通常是左操作数)。

3，一个操作符函数，被声明为哪个类的友员，取决于该函数的参数对象(通常是右操作数)。

6.4.类型转换

6.4.1.标准类型间转换

◆隐式类型转换

```
5/8 5.0/8
```

◆显式类型转换

```
 static_cast<float>(5)/8
```

对于上述两类，我们不再展开赘述，系统有章可循，而对于用户自定义的类型。编译系统则不知道怎么进行转化。解决这个问题的办法是，自定义专门的函数。

如下文：

6.4.2.用类型转换构造函数进行类型转换

实现**其它类型**到**本类类型**的转化。

◆转换构造函数格式

```
class  目标类
{
    目标类(const 源类 & 源类对象引用)
    {
        根据需求完成从源类型到目标类型的转换
    }
}
```

◆特点：

转换构造函数，本质是一个构造函数。是**只有一个参数**的构造函数。如有多个参数，只能称为构造函数，而不是转换函数。

◆代码

```
#include <iostream>

using namespace std;
class Point3D;
class Point2D
{
public:
    Point2D(int x,int y)
        :_x(x),_y(y){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }
    friend Point3D; //friend Point3D::Point3D( Point2D &p2);
```

```
private:
    int _x;
    int _y;
};

class Point3D
{
public:
    Point3D(int x,int y,int z)
        :_x(x),_y(y),_z(z){}

    Point3D(Point2D &p)
    {
        this->_x = p._x;
        this->_y = p._y;
        this->_z = 0;
    }

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<","<<_z<<")"<<endl;
    }

private:
    int _x;
    int _y;
    int _z;
};
void func(Point3D d3)
{

}


int main()
{
    Point2D p2(1,2);
    p2.dis();

    Point3D p3(3,4,5);
    p3.dis();

    Point3D p3a = p2;
    p3a.dis();

    func(d2)

    return 0;
}
```

◆explicit 关键字的意义

以显示的方式完成转化 static_cast<目标类> (源类对象)。否则会报错。

```
 explicit Point3D(Point2D &p) //注:explicit 是个仅用于声明的关键字
 {
        this->_x = p._x;
        this->_y = p._y;
        this->_z = 0;
 }

Point2D p2(1,2);
p2.dis();
Point3D p3a = static_cast<Point3D> (p2); //(Point3D)p2;
p3a.dis();
```

6.4.3.用类型转换操作符函数进行转换

◆类型转化函数格式

```
class 源类{
    operator 转化目标类(void)
    {
        根据需求完成从源类型到目标类型的转换
    }
}
```

◆特点

转换函数必须是类方法，转换函数无参数，无返回。

◆代码

```
#include <iostream>

using namespace std;

class Point3D;

class Point2D
{
public:
    Point2D(int x,int y)
        :_x(x),_y(y){}

    operator Point3D();
    void dis()
    {
        cout<<"("<<_x<<","<<_y<<")"<<endl;
    }

private:
    int _x;
    int _y;
};

class Point3D
{
public:
    Point3D(int x,int y,int z)
```

```
        :_x(x),_y(y),_z(z){}

    void dis()
    {
        cout<<"("<<_x<<","<<_y<<","<<_z<<")"<<endl;
    }

private:
    int _x;
    int _y;
    int _z;
};

Point2D::operator Point3D()
{
    return Point3D(_x,_y,0);
}

int main()
{
    Point2D p2(1,2);
    p2.dis();

    Point3D p3(3,4,5);
    p3.dis();

    Point3D p3a = p2;
    p3a.dis();
    return 0;
}
```

6.4.4.小结

1，只有一个参数的类构造函数，可将参数类型转化为类类型。例如，将 int 类型的变量赋给 obj 类型的对象。在构造函数中声明为 explicit 可以防止隐式转换，只允许显式转换。

2，称为转换函数的特殊类成员运算符函数，用于将类对象转换为其他类型。转换函数为类成员函数。没有返回，没有参数。形如： operator typename(); typename 是对象将被换成的类型。将类对象赋给 typname 变量时或将其强制转换为 typename 类型是，自动被调用。

3，应用于构造及初始化，赋值，传参，返回等场合。

6.4.5.作业

实现如下功能：

```
Complex c (1,2);
Complex c2 = c + 2;  //Complex(double real) 类型转化构造函数 在目标类中
```

```
Complex c (1,2);
double  a  = 2 + c;   //operator double()    类型转化操作符函数 在源类中
```

6.5.运算符重载提高篇

6.5.1.函数操作符（()）---仿函数

把类对象像函数名一样使用。

仿函数(functor)，就是使一个类的使用看上去象一个函数。其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了。

◆格式

```
class 类名
{
    返值类型 operator()(参数类型)
         函数体
}
```

◆应用

```
#include <iostream>
#include <vector>

using namespace std;


class Pow
{
public:
    int operator()(int i)
    {
        return i*i;
    }
    double operator ()(double d)
    {
        return d*d;
    }
};

int main()
{
    Pow pow;
    int i = pow(4);      //pow.opreator()(4);
    double d = pow(5.5);
    cout<<i<<endl;
    cout<<d<<endl;
    return 0;
}
```

注：

主要应用于 STL 和模板

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool funcCmp(int i, int j)
{
    return i<j;
}

class objCmp
{
public:
    bool operator()(int i,int j)
    {
        return i>j;
    }
};

int main()
{
    int array[] = {100,89,23,56,12,200,22,500};
    vector<int> vi(array, array+8);

    // sort(vi.begin(),vi.end(),funcCmp);
    sort(vi.begin(),vi.begin() +4),objCmp());
    //lambda [=](int i, int j){return i>j;}

    for(vector<int>::iterator itr = vi.begin(); itr != vi.end(); ++itr)
    {
        cout<<*itr<<endl;
    }

    return 0;
}
```

6.5.2.堆内存操作符 （new delete）

适用于极个别情况需要定制的时候才用的到。一般很少用。

◆格式

如下：

```
void *operator new(size_t)
void operator delete(void *)
void *operator new[](size_t)
void operator delete[](void *)
```

注: operator new 中 size_t 参数是编译器自动计算传递的。

◆全局重载

```
void * operator new (size_t size)
{
    cout<<"new "<<size<<endl;
    return malloc(size);
}

void  operator delete(void *p)
{
    cout<<"delete"<<endl;
    free(p);
}

void * operator new[] (size_t size)
{
    cout<<"new[] "<<size<<endl;
    return malloc(size);
}

void  operator delete[](void *p)
{
    cout<<"delete[] "<<endl;
    free(p);
}
```

测试：

```
class A
{
public:
    A()
    {
        cout<<"A constructor"<<endl;
    }

    ~A()
    {
        cout<<"A destructor"<<endl;
    }
private:
    int a;
};


int main()
{
    int *p = new int;
    delete p;

    int *pa = new int[20];
    delete []pa;

    A * cp = new A;
    delete cp;
    A * cpa = new A[20];
```

```
    delete []cpa;

    return 0;
}
```

◆类中重载

```
class A
{
public:
    A()
    {
        cout<<"A constructor"<<endl;
    }

    ~A()
    {
        cout<<"A destructor"<<endl;
    }

    void * operator new (size_t size)
    {
        cout<<"new "<<size<<endl;
        void *p = malloc(size); // ((A*)p)->a = 100;
        return  p;
    }

    void  operator delete(void *p)
    {
        cout<<"delete"<<endl;
        free(p);
    }

    void * operator new[] (size_t size)
    {
        cout<<"new[] "<<size<<endl;
        return malloc(size);
    }

    void  operator delete[](void *p)
    {
        cout<<"delete[] "<<endl;
        free(p);
    }
private:
    int a;
};


int main()
{
//    int *p = new int;
//    delete p;

//    int *pa = new int[20];
//    delete []pa;

    A * cp = new A;
    delete cp;
```

```
    A * cpa = new A[20];
    delete []cpa;

    return 0;
}
```

6.5.3.解引用与智能指针（-> /*）

常规意义上讲，new 或是 malloc 出来的堆上的空间，都需要手动 delete 和 free 的。但在其它高级语言中，只需申请无需释放的功能是存在的。

c++中也提供了这样的机制。我们先来探究一下实现原理。

◆常规应用

```
class A
{
public:
    A()
    {
        cout<<"A()"<<endl;
    }
    ~A()
    {
        cout<<"~A()"<<endl;
    }

    void func()
    {
        cout<<"hahaha"<<endl;
    }
};
void foo()
{
    A*p = new A;
    //
    delete p;
}
```

◆智能指针

```
#include <memory>
void foo()
{
     auto_ptr<A> p (new A);
     p->func(); //两种访问方式
     (*p).func();
}
```

◆推演

1st step

```
class A
{
```

```
public:
    A()
    {
        cout<<"A constructor"<<endl;
    }
    ~A()
    {
        cout<<"A destructor"<<endl;
    }
};

class PMA
{
public:
    PMA(A *p)
        :_p(p){}
    ~PMA()
    {
        delete  _p;
    }


private:
    A * _p;
};

int main()
{
    A * p = new A;
    PMA pma(new A);

    return 0;
}
```

2sd step

```
#include <iostream>
#include <memory>

using namespace std;
class A
{
public:
    A()
    {
        cout<<"A constructor"<<endl;
    }
    ~A()
    {
        cout<<"A destructor"<<endl;
    }
    void dis()
    {
        cout<<"in class A's dis"<<endl;
    }

};

class PMA
{
```

```
public:
    PMA(A *p)
        :_p(p){}
    ~PMA()
    {
        delete  _p;
    }

    A& operator*()
    {
        return *_p;
    }

    A* operator->()
    {
        return _p;
    }

private:
    A * _p;
};

int main()
{
    A * p = new A;
    PMA pma(new A);
//    pma._p->dis();   private 的原因破坏了封装
    (*pma).dis(); //(pma*).dis();  (*pms).dis();
    pma->dis();   // pma->->dis(); pma->dis();

    return 0;
}
```

◆ ->和* 重载格式

```
类名& operator*()
{
     函数体
}
```

```
类名* operator->()
{
     函数体
}
```

6.6.作业

6.6.1.设计TDate类

定义一个处理日期的类 TDate，它有 3 个私有数据成员：Month,Day,Year 和若干个公有成员函数，并实现如下要求：

a.构造函数重载

b.成员函数设置缺省参数

c. 可使用不同的构造函数来创建不同的对象

d.定义一个友元函数来打印日期

6.6.2.设计一个矩阵类

设计一个 3*3 的矩阵类 class Matrix，通过一数组进行初始化。要求如下：

a.默认构造(初始化为 0)，有参构造(数组作实参)

b.重载+ / +=

c. 重载* / *=

d.实现输出

提示：

```
class Matrix
{
public:
    Matrix(void);
    Matrix(int p[][3]);

private:
    int data[3][3];
};
```

6.6.3.设计代理类

好的软件工程有两个基本原则，一是接口与实现的分析，二是隐藏实现细节。为此，我们向用户提供头文件 ,而实现在一个 cpp 文件中。但是 ,用户提供的头文件中 ,还是有些 private，虽然用户不能访问，但还是把私有信息暴露给了客户。

通过向客户提供只知道类的 public 接口的代理类，就可以使客户能够使用类的服务，而无法访问类的实现细节。

```
#include <iostream>

using namespace std;

class Implementation
{
public:
    Implementation(int v)
        :value(v){}

    void setValue(int v)
    {
        value = v;
    }
    int getValue() const
```

```
    {
        return value;
    }

private:
    int value;
};

class Delegate
{
public:
    Delegate(int v)
        :pi(new Implementation(v))
    {}

    void setValue(int v)
    {
       pi->setValue(v);
    }
    int getValue() const
    {
        return pi->getValue();
    }

private:

    Implementation * pi;
};

int main()
{
    Delegate  d(5);
    d.setValue(100);

    cout<<d.getValue()<<endl;


    return 0;
}
```

7.继承与派生(Inherit&&Derive)

7.1.引入

在C++中可**重用性**(software reusability)是通过继承(inheritance)这一机制来实现的。如果没有掌握继承性，就没有掌握类与对象的精华。

引例：

7.1.1.归类

对于学生和老师进行归类。

学生	姓名	年龄	学号	吃饭	学习
老师	姓名	年龄	工资	吃饭	教学

7.1.2.抽取

姓名	年龄	吃饭

7.1.3.继承

姓名	年龄	吃饭

学生	学号	学习

工资	教学	老师

7.1.4.重用

```
#include <iostream>

using namespace std;

class Person
{
public:

    void eat(string food)
    {
        cout<<"i am eating "<<food<<endl;
    }

};

class Student:public Person
{
public:
```

```
    void study(string course)
    {
        cout<<"i am a student  i study "<<course<<endl;
    }
};

class Teacher:public Person
{
public:
    void teach(string course)
    {
        cout<<"i am a teacher i teach "<<course<<endl;
    }

};




int main()
{
    Student s;
    s.study("C++");
    s.eat("黄焖鸡");


    Teacher t;
    t.teach("Java");
    t.eat("驴肉火烧");
    return 0;
}
```

7.2.定义

类的继承，是新的类从已有类那里得到已有的特性。或从已有类产生新类的过程就是类的派生。原有的类称为基类或父类，产生的新类称为派生类或子类。

派生与继承，是同一种意义两种称谓。

7.3.继承

7.3.1.关系定性is-a/has-a

is-a 是一种属于关系 ,如 :狗属于一种动物 ,车属于一种交通工具(Dog is an Animal. Car is a Vehicle. ）在面向对象中表现为一种继承关系。可以设计一个 Animal 类，Dog 类作为 Animal 类（基类）的派生类；设计一个 Vehicle 类，Car 类作为 Vehicle 类（基类）的派生类。

has-a 是一种包含、组合关系。如 :车包含方向盘、轮胎、发动机(Car has steering-wheel, wheels, engine ），但不能说方向盘/轮胎/发动机是一种车；狗包含腿、尾巴，但不能说腿/尾巴是一种狗。正确的应该说车聚合（包含）了方向盘、轮胎、发动机。

is-a 这种关系可以完成代码复用，是继承。把比较抽象的类（如例子中的动物、交通工具）定义为基类，把比较具体的定义为子类（派生类），比如狗、兔子、马都以动物做为基类而做出派生类。基类都可以有跑、吃、睡觉等共同方法，高度、体重等共同属性；具体到狗、兔子、马则有自己特别的属性如食物，特别的方法如叫声等。

因此，如果 A 是 B，则 B 是 A 的基类，A 是 B 的派生类。为继承关系。

has-a 这种关系可以把一个复杂的类处理为一个个相对简单的类，是聚合。比如创建方向盘类、轮胎类、发动机类，最后创建车类。车类调用 4 个轮胎类实例（假如该车有 4 个轮胎）和 1 个方向盘类实例（对象）和一个发动机类实例（对象）。每个类本身相对简单，通过聚合（组合）成为一个复杂的类。

因此，如果 A 包含 B，则 B 是 A 的组成部分。为聚合关系，可以由组成部分聚合成为一个类。

7.3.2.语法

派生类的声明：

```
class 派生类名：[继承方式] 基类名
{
    派生类成员声明；
};
```

一个派生类可以同时有多个基类，这种情况称为多重继承，派生类只有一个基类，称为单继承。下面从单继承讲起。

7.3.3.继承方式

继承方式规定了如何**访问基类继承的成员**。继承方式有 public, private, protected。继承方式不影响派生类的访问权限，影响了**从基类继承来的成员的访问权限，包括派生类内的访问权限和派生类对象。**

简单讲：

公有继承：基类的公有成员和保护成员在派生类中保持原有访问属性，其私有成员仍为基类的私有成员。

私有继承：基类的公有成员和保护成员在派生类中成了私有成员，其私有成员仍为基类的私有成员。

保护继承：基类的公有成员和保护成员在派生类中成了保护成员，其私有成员仍为基类的私有成员。

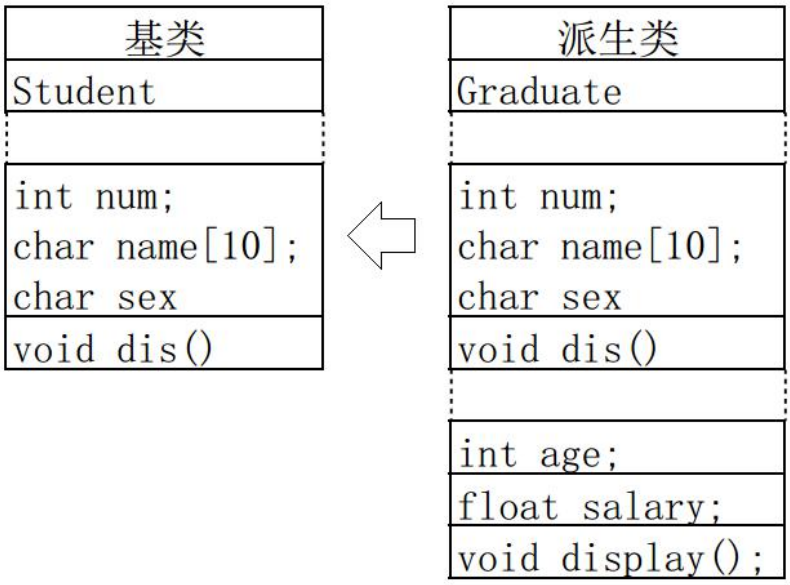
pretected 对于外界访问属性来说，等同于私有，但可以派生类中可见。

```
#include <iostream>
```

```
using namespace std;

class Base
{
public:
    int pub;
protected:
    int pro;
private:
    int pri;
};

class Drive:public Base
{
public:
    void func()
    {
        pub = 10;
        pro = 100;
//        pri = 1000;

public;
    int a;
protected:
    int b;
private:
    int c
};

//

int main()
{
    Base b;
    b.pub = 10;
//    b.pro = 100;

//    b.pri = 1000;
    return 0;
}
```

7.3.4.派生类的组成

派生类中的成员，包含两大部分，一类是从基类继承过来的，一类是自己增加的成员。从基类继承过过来的表现其共性，而新增的成员体现了其个性。



几点说明：

1，全盘接收，除了构造器与析构器。基类有可能会造成派生类的成员冗余，所以说基类是需设计的。

2，派生类有了自己的个性，使派生类有了意义。

```
#include <iostream>
#include <typeinfo>

using namespace std;

class A
{
public:
    A()
    {
        cout<<this<<endl;
        cout<<typeid(this).name()<<endl;
    }

   int a;
};

class B:public A
{
public:
    B()
    {
        cout<<this<<endl;
        cout<<typeid(this).name()<<endl;
    }
    int b;
```

```
};

class C:public B
{
public:
    C()
    {
        cout<<this<<endl;
        cout<<typeid(this).name()<<endl;
    }
    void func()
    {
        cout<<&a<<endl;
        cout<<&b<<endl;
        cout<<&c<<endl;
    }

    int c;
};

int main()
{
    C c;

    cout<<"&c "<<&c<<endl;
    cout<<"**************"<<endl;
    c.func();

    return 0;
}
```

7.4.派生类的构造

派生类中由基类继承而来的成员的初始化工作还是由基类的构造函数完成，然后派生类中新增的成员在派生类的构造函数中初始化。

7.4.1.派生类构造函数的语法：

```
派生类名::派生类名（参数总表）
    ：基类名（参数表）,内嵌子对象（参数表）
{
    派生类新增成员的初始化语句; //也可出现地参数列表中
}
```

注：

◆构造函数的初始化顺序并不以上面的顺序进行，而是根据声明的顺序初始化。

◆如果基类中**没有**默认构造函数(无参)，那么在派生类的构造函数中必须**显示**调用基类构造函数，以初始化基类成员。

◆派生类构造函数执行的次序：

```
基类-->成员-->子类
```

a 调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左到右）；

b 调用内嵌成员对象的构造函数，调用顺序按照它们在类中声明的顺序；

c 派生类的构造函数体中的内容。

7.4.2.代码实现

◆祖父类

student.h

```
class Student
{
public:
    Student(string sn,int n,char s);

    ~Student();

    void dis();

private:
    string name;
    int num;
    char sex;

};
```

student.cpp

```
Student::Student(string sn, int n, char s)
    :name(sn),num(n),sex(s)
{

}

Student::~Student()
{

}

void Student:: dis()
{
    cout<<name<<endl;
    cout<<num<<endl;
    cout<<sex<<endl;
}
```

◆父类

graduate.h

```
class Graduate:public Student
{
```

```
public:
    Graduate(string sn,int in,char cs,float fs);
    ~Graduate();

    void dump()
    {
        dis();
        cout<<salary<<endl;
    }

private:
    float salary;
};
```

graduate.cpp

```
Graduate::Graduate(string sn, int in, char cs, float fs)
    :Student(sn,in,cs),salary(fs)
{

}

Graduate::~Graduate()
{

}
```

◆ 类成员

birthday.h

```
class Birthday
{
public:
    Birthday(int y,int m,int d);
    ~Birthday();

    void print();

private:
    int year;
    int month;
    int day;
};
```

birthday.cpp

```
 Birthday::Birthday(int y, int m, int d)
    :year(y),month(m),day(d)
{

}

Birthday::~Birthday()
{
}

void Birthday::print()
{
```

```
    cout<<year<<month<<day<<endl;
}
```

◆ 子类

doctor.h

```
class Doctor:public Graduate
{
public:
    Doctor(string sn,int in,char cs,float fs,string st,int iy,int im,in
t id);
    ~Doctor();

    void disdump();

private:
    string title; //调用的默认构造器，初始化为""

    Birthday birth;  //类中声明的类对象
};
```

doctor.cpp

```
Doctor::Doctor(string sn, int in, char cs, float fs, string st, int iy,
int im, int id)
    :Graduate(sn,in,cs,fs),birth(iy,im,id),title(st)
{
}

Doctor::~Doctor()
{

}

void Doctor::disdump()
{
    dump();
    cout<<title<<endl;
    birth.print();

}
```

◆测试代码

```
int main()
{

    Student s("zhaosi",2001,'m');
    s.dis();
    cout<<"----------------"<<endl;
    Graduate g("liuneng",2001,'x',2000);
    g.dump();

    cout<<"----------------"<<endl;
    Doctor d("qiuxiang",2001,'y',3000,"doctor",2001,8,16);
    d.disdump();
    return 0;
```

```
}
```

7.4.3.结论

子类构造器中，要么显示的调用父类的构造器(传参)，要么隐式的调用。发生隐式调用时，父类要有无参构造器或是可以包含无参构造器的默认参数函数。子类对象亦然。

7.5.派生类的拷贝构造

7.5.1.格式

```
派生类::派生类(const 派生类& another)
    :基类(another),派生类新成员(another.新成员)
{

}
```

7.5.2.代码

◆ 父类

student.h

```
class Student
{
public:
    Student(string sn,int n,char s);

    Student(const Student & another);

    ~Student();

    void dis();

private:
    string name;
    int num;
    char sex;

};
```

student.cpp

```
 Student::Student(string sn, int n, char s)
    :name(sn),num(n),sex(s)
{

}

Student::~Student()
{

}

void Student:: dis()
{
```

```
    cout<<name<<endl;
    cout<<num<<endl;
    cout<<sex<<endl;
}

Student::Student(const Student & another)
{
    name = another.name;
    num  = another.num;
    sex  = another.sex;
}
```

◆子类

graduate.h

```
class Graduate:public Student
{
public:
    Graduate(string sn,int in,char cs,float fs);
    ~Graduate();

    Graduate(const Graduate & another);

    void dump()
    {
        dis();
        cout<<salary<<endl;
    }

private:
    float salary;
};
```

graduate.cpp

```
Graduate::Graduate(string sn, int in, char cs, float fs)
    :Student(sn,in,cs),salary(fs)
{

}

Graduate::~Graduate()
{

}

Graduate::Graduate(const Graduate & another)
    :Student(another),salary(another.salary)
{

}
```

◆测试代码

```
int main()
{
```

```
    Graduate g("liuneng",2001,'x',2000);
    g.dump();
    Graduate gg = g;
    gg.dump();

    return 0;
}
```

7.5.3.结论：

派生类中的默认拷贝构造器会调用父类中默认或自实现拷贝构造器 ,若派生类中自实现拷贝构造器，**则必须显示**的调用父类的拷贝构造器。

7.6.派生类的赋值运算符重载

赋值运算符函数不是构造器，所以可以继承，语法上就没有构造器的严格一些。

7.6.1.格式

```
子类& 子类::operator=(const 子类& another)
{
    if(this == &another)
        return *this; //防止自赋值

    父类::operator =(another); // 调用父类的赋值运算符重载

    this->salary = another.salary;//子类成员初始化

    return * this;
}
```

7.6.2.代码

◆ 基类

student.h

```
Student & operator=(const Student & another);
```

student.cpp

```
Student &  Student::operator=(const Student & another)
{
    this->name = another.name;
    this->num  = another.num;
    this->sex  = another.sex;

    return * this;
}
```

◆ 派生类

graduate.h

```
Graduate & operator=(const Graduate & another);
```

graduate.cpp

```
Graduate & Graduate::operator=(const Graduate & another)
{
    if(this == &another)
        return *this;

    Student::operator =(another);

    this->salary = another.salary;

    return * this;
}
```

◆测试代码

```
int main()
{

    Graduate g("liuneng",2001,'x',2000);
    g.dump();
    Graduate gg = g;
    gg.dump();
    cout<<"-----------"<<endl;
    Graduate ggg("gege",2001,'x',4000);
    ggg.dump();
    ggg = g;
    ggg.dump();

    return 0;
}
```

7.6.3.结论：

派生类的默认赋值运算符重载函数，会调用父类的默认或自实现函数。派生类若自实现，**则不会发生调用行为，也不报错(**区别拷贝**)，赋值错误，**若要正确，需要显示的调用父类的构造器。

7.7.派生类友元函数

由于友元函数并非类成员，因引不能被继承，在某种需求下，可能希望派生类的友元函数能够使用基类中的友元函数。为此可以通过强制类型转换，将**派生类的指针或是引用强转为其类的引用或是指针**，然后使用转换后的引用或是指针来调用基类中的友元函数。

```
#include <iostream>

using namespace std;

class Student
{
```

```
friend ostream &operator<<(ostream & out, Student & stu);
private:
    int a;
    int b;
};

ostream &operator<<(ostream & out, Student & stu)
{
    out<<stu.a<<"--"<<stu.b<<endl;
}

class Graduate:public Student
{
friend ostream &operator<<(ostream & out, Graduate & gra);
private:
    int c;
    int d;
};

ostream &operator<<(ostream & out, Graduate & gra)
{
    out<<(Student&)gra<<endl;
    out<<gra.c<<"**"<<gra.d<<endl;
}

int main()
{
//    Student a;
//    cout<<a<<endl;

    Graduate g;

    cout<<g<<endl;

    return 0;
}
```

7.8.派生类析构函数的语法

派生类的析构函数的功能是在该对象消亡之前进行一些必要的清理工作，析构函数没有类型，也没有参数。析构函数的执行顺序与构造函数**相反**。

◆析构顺序

```
子类->成员->基类
```

无需指明析构关系。why？析构函数只有一种，无重载，无默参。

7.9.派生类成员的标识和访问

7.9.1.作用域分辨符

◆格式：

```
基类名::成员名；基类名::成员名（参数表）；
```

如果某派生类的多个基类拥有同名的成员，同时，派生类又新增这样的同名成员，在这种情况下，派生类成员将 shadow(**隐藏**)所有基类的同名成员。这就需要这样的调用方式才能调用基类的同名成员。

◆代码:

```
#include <iostream>

using namespace std;

class Base
{
public:
    void func(int)
    {
        cout<<"haha"<<endl;
    }

};

class Drive:public Base
{
public:
    void func()
    {
//        func();               //func 死循环
//        Base::func();          //被 shadow 的成员，可以这样访问
        cout<<"hehe"<<endl;
    }

};

//

int main()
{
    Drive d;
    d.func();     // 访问派生类成员
//    d.Base::func(3); //访问基类成员
    return 0;
}
```

◆小结

重载：同一作用域 ，函数同名不同参(个数，类型，顺序);

隐藏：父子类中，标识符(函数，变量)相同，无关乎返值和参数(函数)，或声明类　型

(变量)。

7.9.2.继承方式

7.9.2.1.图示

继承方式 / 成员	public	protected	private
public	public	protected	private
protected	protected	protected	inaccessable
private	inaccessable	inaccessable	inaccessable

7.9.2.2.详解

◆public 公有继承

当类的继承方式为公有继承时，基类的公有和保护成员的访问属性在派生类中不变，而基类的私有成员不可访问。即基类的公有成员和保护成员被继承到派生类中仍作为派生类的公有成员和保护成员。派生类的其他成员可以直接访问它们。无论派生类的成员还是派生类的对象都无法访问基类的私有成员。

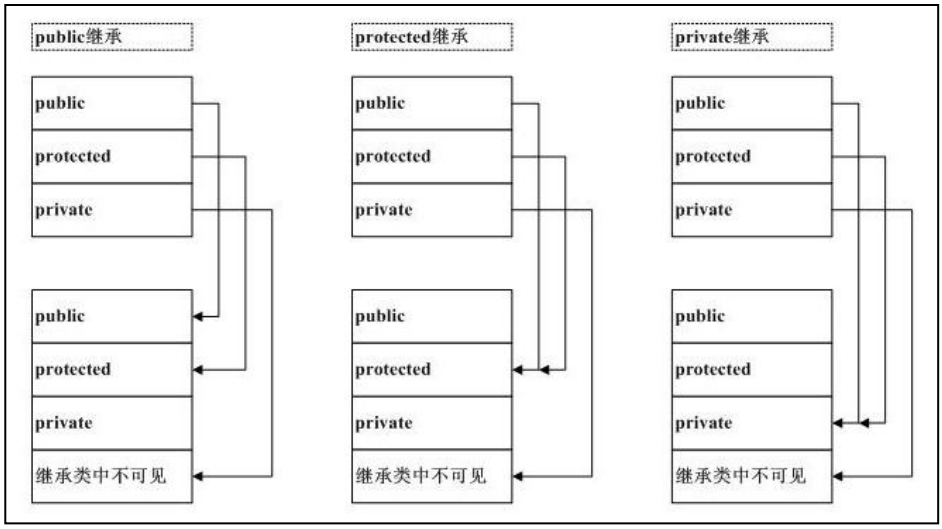
◆private 私有继承

当类的继承方式为私有继承时，基类中的公有成员和保护成员都以私有成员身份出现在派生类中，而基类的私有成员在派生类中不可访问。基类的公有成员和保护成员被继承后作为派生类的私有成员，派生类的其他成员可以直接访问它们，但是在类外部通过派生类的对象无法访问。无论是派生类的成员还是通过派生类的对象，都无法访问从基类继承的私有成员。通过多次私有继承后，对于基类的成员都会成为不可访问。因此私有继承比较少用。

◆protected 保护继承

保护继承中，基类的公有成员和私有成员都以保护成员的身份出现在派生类中，而基类的私有成员不可访问。派生类的其他成员可以直接访问从基类继承来的公有和保护成员，但是类外部通过派生类的对象无法访问它们，无论派生类的成员还是派生类的对象，都无法访问基类的私有成员。

图示之：



7.9.3.派生类成员属性划分为四种：

公有成员；保护成员；私有成员；不可访问的成员；

```
#include <iostream>
using namespace std;

class Base
{
public:
    int pub;
protected:
    int pro;
private:
    int pri;
};

class Drive:public Base
{
public:
    void func()
    {
        pub = 10;
        pro = 100;
//        pri = 1000;
    }
};

//

int main()
{
    Base b;
    b.pub = 10;
//    b.pro = 100;

//    b.pri = 1000;
    return 0;
}
```

7.10.why public

7.10.1.继承方式与成员访问属性

继承方式 / 成员	public	protected	private
public	public	protected	private
protected	protected	protected	inaccessable
private	inaccessable	inaccessable	inaccessable

7.10.2.公有继承的意义：

```
#include <iostream>
using namespace std;
class Base
{
public:
    int pub;
protected:
    int pro;
private:
    int pri;
};
class Drive:public Base
{
public:
};
```

private 在子类中不可见，但仍可通过父类接口访问。

继承方式 / 成员	public	public	public
public :pub	pub	pub	pub
protected:pro	pro	pro	pro
private :pri			

public 作用：传承接口　间接的传承了数据(protected)。

继承方式 / 成员	protected	protected	protected
public　:pub	pro	pro	pro
protected:pro	pro	pro	pro
private　:pri			

protected 作用：传承数据，间接封杀了对外接口。

继承方式 / 成员	private	private	private
public　:pub	pri		
protected:pro	pri		
private　:pri			

private 统杀了数据和接口

1.只要是私有成员到派生类中,均不可访问. 正是体现的数据隐蔽性.其私有成员仅可被本类的成员函数访问

2.如果多级派生当中，均采用 public,直到最后一级，派生类中均可访问基类的 public,protected 成员.

兼顾了数据的隐蔽性和接口传承和数据传递

3.如果多级派生当中，均采用 private,直到最后一级,派生类中基类的所有成员均变为不可见.

只兼顾了数据的隐蔽性

4.如果多级派生当中,均采用 protected,直到最后一级，派生类的基类的所有成员即使可见,也均不可被类外调用

只兼顾了数据的隐蔽性和数据传递

综上所述：记住 public 足矣。

7.10.3.使用Qt类库

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class MainWindow : public/protected/private QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
};

#endif // MAINWINDOW_H
```

7.10.4.私有继承和保护继承的存在意义

此两种方式有效的防止了基类公有接口的扩散。是一种实现继承，而不是一种单纯的 isA 的关系了。

```
class Windows
{
public:
    一般常见基础接口
protected:
    常用特效接口
    高级特效接口
private:
    duang 特效接口
};
```

7.11.多继承

从继承类别上分，继承可分为单继承和多继承，前面讲的都是单继承。

7.11.1.多继承的意义

俗话讲的，鱼与熊掌不可兼得，而在计算机就可以实现，生成一种新的对象，叫熊掌鱼，多继承自鱼和熊掌即可。还比如生活中，“兼”。

7.11.2.继承语法

```
派生类名::派生类名（参数总表）
     ：基类名 1（参数表 1），基类名（参数名 2）....基类名 n（参数名 n），
      内嵌子对象 1（参数表 1），内嵌子对象 2（参数表 2）....内嵌子对象 n（参数表 n）
{
    派生类新增成员的初始化语句；
}
```

7.11.3.沙发床实现

7.11.3.1.继承结构：

7.11.3.2.代码:

◆床类

bed.h

```
#ifndef BED_H
#define BED_H
```

```
class Bed
{
public:
    Bed();
    ~Bed();

    void sleep();
};

#endif // BED_H
```

bed.cpp

```
#include "bed.h"
#include "iostream"
using namespace std;

Bed::Bed()
{
}

Bed::~Bed()
{
}
void Bed::sleep()
{
    cout<<"take a good sleep"<<endl;
}
```

◆沙发类

sofa.h

```
#ifndef SOFA_H
#define SOFA_H
class Sofa
{
public:
    Sofa();
    ~Sofa();

    void sit();
};

#endif // SOFA_H
```

sofa.cpp

```
#include "sofa.h"
#include "iostream"
using namespace std;

Sofa::Sofa()
{
}
Sofa::~Sofa()
{
}
void Sofa::sit()
{
```

```
    cout<<"take a rest"<<endl;
}
```

◆沙发床类

sofabed.h

```
#ifndef SOFABED_H
#define SOFABED_H

#include "sofa.h"
#include "bed.h"


class SofaBed:public Sofa,public Bed
{
public:
    SofaBed();
    ~SofaBed();
};

#endif // SOFABED_H
```

sofabed.cpp

```
#include "sofabed.h"

SofaBed::SofaBed()
{
}

SofaBed::~SofaBed()
{
}
```

◆测试代码：

main.cpp

```
#include <iostream>
#include "sofa.h"
#include "bed.h"
#include "sofabed.h"
using namespace std;

int main()
{
    Sofa  s;
    s.sit();

    Bed   b;
    b.sleep();


    SofaBed sb;
    sb.sit();
    sb.sleep();
    return 0;
}
```

7.11.4.三角问题(二义性问题)

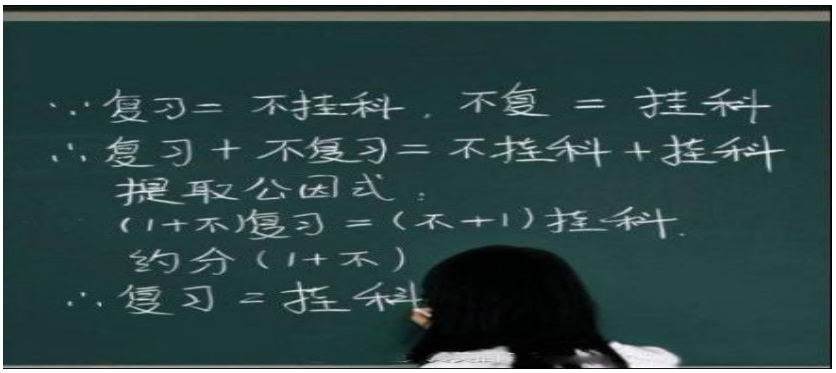
多个父类中重名的成员，继承到子类中后，为了避免冲突，携带了各父类的作用域信息，子类中要访问继承下来的重名成员，则会产生二义性，为了避免冲突，访问时需要还有父类的作用域信息。

```
#include <iostream>

using namespace std;

class X
{
public:
    X(int d)
        :_data(d){}

    void setData(int i)
    {
        _data = i;
    }

    int _data;
};

class Y
{
public:
    Y(int d)
        :_data(d){}
    int getData()
    {
        return _data;
    }
    int _data;
};

class Z:public X,public Y
{
public:
    Z():X(2),Y(3)
    {}

    void dis()
    {
        cout<<X::_data<<endl;
        cout<<Y::_data<<endl;
    }
};

int main()
{
    Z z;
    z.dis();
    z.setData(2000);
    cout<<z.getData()<<endl;
    return 0;
}
```

7.11.5.钻石问题

7.11.5.1. 三角转四角

采用提取公因式的方法




```
#include <iostream>

using namespace std;

class M
{
public:
    M(int i)
        :_data(i){}

    int _data;
};

class X:public M
{
public:
    X(int d)
        :M(d){}

    void setData(int i)
    {
        _data = i;
```

```
    }

};

class Y:public M
{
public:
    Y(int d)
        :M(d){}
    int getData()
    {
        return _data;
    }
};

class Z:public X,public Y
{
public:
    Z():X(2),Y(3)
    {}

    void dis()
    {
        cout<<X::_data<<endl;
        cout<<Y::_data<<endl;
    }
};

int main()
{
    Z z;
    z.dis();
    z.setData(2000);
    cout<<z.getData()<<endl;
    return 0;
}
```

7.11.5.2.虚继承

```
#include <iostream>

using namespace std;


class M
{
public:
    M(int d)
        :_data(d)
    {}
    int _data;
};



class X :virtual public M
{
```

```
public:
    X(int d)
        :M(d){}

    void setD(float d)
    {
       _data = d;
    }
};

class Y:virtual public M
{
public:
    Y(int d)
        :M(d){}

    int getD()
    {
        return _data;
    }
};

class Z:public X,public Y
{
public:
    Z(int _x,int _y):X(_x),Y(_y),M(100)
    {}
    void dis()
    {
//        cout<<X::_data<<endl;
//        cout<<Y::_data<<endl;
        cout<<_data<<endl;
    }
};

int main()
{
    Z z;
    z.dis();
    z.setData(2000);
    cout<<z.getData()<<endl;
    return 0;
}
```

7.11.5.3.小结

◆虚继承的意义

在多继承中，保存共同基类的多份同名成员，虽然有时**是必要**的，可以在不同的数据成员中分别存放不同的数据，但在大多数情况下，是我们**不希望出现**的。因为保留多份数据成员的拷贝，不仅占有较多的存储空间，还增加了访问的困难。

为此，c++提供了，虚基类和虚继承机制，实现了在多继承中只保留一份共同成员。

虚基类，需要设计和抽象，虚继承，是一种继承的扩展。

◆语法总结

a.M 类称为虚基类(virtual base class)，是抽象和设计的结果。

b.虚继承语法

```
class 派生类名:virtual 继承方式 基类
```

c.虚基类及间接类的实始化

```
 class A{
    A(int i)
    {}
};
class B:virtual public A
{
    B(int n):A(n){}
};
class C:virtual public A
{
    C(int n):A(n){}
};
class D:public B,public C
{
    D(int n)
        :A(n),B(n),C(n)
    {}
};
```

7.11.5.4.改造沙发床

7.11.6.多继承实现的原理

8.多态（PolyMorphism）

8.1.浅析多态的意义

如果有几个上似而不完全相同的对象，有时人们要求在向它们发出同一个消息时，它们的反应各不相同，分别执行不同的操作。这种情况就是多态现象。

例如，甲乙丙 3 个班都是高二年级，他们有基本相同的属性和行为，在同时听到上课铃声的时候，他们会分别走向 3 个不同的教室，而不会走向同一个教室。

同样，如果有两支军队，当在战场上听到同种号声，由于事先约定不同，A 军队可能实施进攻，而 B 军队可能准备 kalalok。

又如在 winows 环境下，用鼠标双击一个对象（这就是向对象传递一个消息），如果对象是一个可执行文件，则会执行此程序，如果对象是一个文本文件，由会启动文本编辑器并打开该文件。

C++中所谓的多态(polymorphism)是指，由继承而产生的相关的不同的类，其对象对同一消息会作出不同的响应。

多态性是面向对象程序设计的一个重要特征，能增加程序的灵活性。可以减轻系统升级,维护,调试的工作量和复杂度.

8.2.赋值兼容(多态实现的前提)

8.2.1.规则

赋值兼容规则是指在需要基类对象的任何地方都可以使用**公有派生类**的对象来替代。赋值兼容是一种默认行为，不需要任何的显示的转化步骤。

赋值兼容规则中所指的替代包括以下的情况：

◆派生类的对象可以赋值给基类对象。
◆派生类的对象可以初始化基类的引用。
◆派生类对象的地址可以赋给指向基类的指针。
在替代之后，派生类对象就可以作为基类的对象使用，**但只能使用从基类继承的成员。**

8.2.2.代码

```
#include <iostream>

using namespace std;

class Shape
{
public:
    Shape(int x,int y)
        :_x(x),_y(y){}
```

```
    void draw()
    {
        cout<<"draw Shap ";
        cout<<"start ("<<_x<<","<<_y<<") "<<endl;
    }

//private:
protected:
    int _x;
    int _y;
};

class Circle:public Shape
{
public:
   Circle(int x, int y,int r)
       :Shape(x,y),_r(r){}

   void draw()
   {
       cout<<"draw Circle ";
       cout<<"start ("<<_x<<","<<_y<<") ";
       cout<<"radio r = "<<_r<<endl;
   }

private:
    int _r;

};

int main()
{
    Shape s(3,5);
    s.draw();

    Circle c(1,2,4);
    c.draw();

    s = c;
    s.draw();
    Shape &rs  = c;
    rs.draw();


    Shape *ps = &c;
    ps->draw();

    return 0;
}
```

8.2.3.补充：

父类也可以通过强转的方式转化为子类。 父类对象强转为子类对象后，访问从父类继承下来的部分是可以的，但访问子类的部分，则会发生**越界**的风险，越界的结果是未知的。

```
//c = static_cast<Circle>(s);  //缺少转化函数
//c.draw();
  Circle * pc = static_cast<Circle*>(&s);
  pc->draw();
```

8.3.多态形成的条件

8.3.1.多态

8.3.1.1.静多态

前面学习的函数重载，也是一种多态现象，通过命名倾轧在编译阶段决定，故称为静多态。

8.3.1.2.动多态

动多态，不是在编译器阶段决定，而是在运行阶段决定，故称为动多态。动多态行成的条件如下:

1，父类中有虚函数。

2，子类 override(覆写)父类中的虚函数。

3，通过已被子类对象赋值的父类指针或引用，调用共用接口。

8.3.2.虚函数

◆格式

```
class 类名
{
 virtual  函数声明;
}
```

◆例举

Shape 类中

```
   virtual void draw()
   {
       cout<<"draw Shap ";
       cout<<"start ("<<_x<<","<<_y<<") "<<endl;
   }
```

Circle 类中

```
  void draw()
  {
      cout<<"draw Circle ";
      cout<<"start ("<<_x<<","<<_y<<") ";
      cout<<"radio r = "<<_r<<endl;
  }
```

Rect 类中

```
   void draw()
   {
       cout<<"draw Rect";
       cout<<"start ("<<_x<<","<<_y<<") ";
       cout<<"len = "<<_len<<" wid = "<<_wid<<endl;
   }
```

◆测试

```
int main()
{

    Circle c(1,2,4);
    c.draw();

    Rect   r(2,3,4,5);
    r.draw();

    Shape *ps;

    int choice;
    while(1)  //真正的实现了动多态，在运行阶段决定。
    {
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                ps = &c;
                ps->draw();
                break;
            case 2:
                ps = &r;
                ps->draw();
                break;
        }
    }

    return 0;
}
```

8.3.3.虚函数小结

1，在基类中用 virual 声明成员函数为虚函数。类外实现虚函数时，不必再加 virtual.

2，在派生类中重新定义此函数称为覆写，要求函数名，返值类型，函数参数个数及类型全部匹配。并根据派生类的需要重新定义函数体。

3，当一个成员函数被声明为虚函数后，其派生类中完全相同的函数（显示的写出）也为虚函数。 可以在其前加 virtual 以示清晰。

4，定义一个指基类对象的指针，并使其指向其子类的对象，通过该指针调用虚函数，此时调用的就是指针变量指向对象的同名函数。

5，子类中的覆写的函数，**可以为任意访问类型**，依子类需求决定。

```
#include <iostream>
```

```
using namespace std;

class Base
{
public:
    virtual void func()
    {
        cout<<"Base"<<endl;
    }

};

class Derive:public Base
{
private:
    void func()
    {
        cout<<"Derive"<<endl;
    }
};

int main()
{
    Derive d;
//    d.func();

    Base*p  = &d;
    p->func();
    return 0;
}
```

8.3.4.纯虚函数

◆格式

```
class 类名
{
 virtual  函数声明 = 0;
}
```

◆例举

Shape 类中

```
virtual void draw() = 0;
```

Circle 类中

```
   void draw()
   {
       cout<<"draw Circle ";
       cout<<"start ("<<_x<<","<<_y<<") ";
       cout<<"radio r = "<<_r<<endl;
   }
```

◆测试

```
int main()
{
//    Shape s(1,2); //函数纯虚函数的类称为抽象基类

    Circle c(1,2,3);
    Rect   r(1,2,3,5);

    Shape *pc = &c;
    pc->draw();

    pc = &r;
    pc->draw();

    return 0;
}
```

8.3.5.纯虚函数小结

1.含有纯虚函数的类，称为**抽象基类**，不可实列化。即不能创建对象，**存在的意义就是被继承，提供族类的公共接口，java 中称为 interface。**

2.纯虚函数只有声明，没有实现，被“初始化”为 0。

3.如果一个类中声明了纯虚函数，而在派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数，派生类仍然为纯虚基类。

8.3.6.含有虚函数的析构

含有虚函数的类，析构函数也应该声明为虚函数。在 delete 父类指针的时候，会调用子类的析构函数，实现完整析构。

8.3.7.若干限制

1)只有类的成员函数才能声明为虚函数

虚函数仅适用于有继承关系的类对象，所以普通函数不能声明为虚函数。

2)静态成员函数不能是虚函数

静态成员函数不受对象的捆绑，只有类的信息。

3)内联函数不能是虚函数

4)构造函数不能是虚函数

构造时，对象的创建尚未完成。构造完成后，才能算一个名符其实的对象。

5)析构函数可以是虚函数且通常声明为虚函数。

8.4.案例

8.4.1.覆写--基于qt覆写鼠标事件

◆qmainwindow.h

```
#ifndef MAINWINDOW_H
```

```
#define MAINWINDOW_H

#include <QMainWindow>

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
    virtual voidmousePressEvent(QMouseEvent * event);

};

#endif // MAINWINDOW_H
```

◆**qmainwindow.cpp**

```
#include "mainwindow.h"
#include <QMouseEvent>
#include <QDebug>

#include <iostream>
using namespace std;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    this->setFixedSize(500,300);
}

MainWindow::~MainWindow()
{

}
void MainWindow::mousePressEvent(QMouseEvent * event)
{
    if(event->button() == Qt::LeftButton)
    {
        qDebug()<<"MouePressEnent "<<event->x()<<""<<event->y()<<""
          <<event->globalX()<<""<<event->globalY()<<endl;
    }

}
```

◆**main.cpp**

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

8.4.2.虚析构--动物园里欢乐多

多态由栈对象到堆对象的过度，**堆对象销毁**的问题。

◆ 动物类

animal.h

```
#ifndef ANIMAL_H
#define ANIMAL_H


class Animal
{
public:
    Animal();
    virtual ~Animal();
    virtual void voice() = 0;
};

#endif // ANIMAL_H
```

◆animal.cpp

```
#include "animal.h"
#include <iostream>
using namespace  std;

Animal::Animal()
{
    cout<<"Animal::Animal()"<<endl;
}

Animal::~Animal()
{
    cout<<"Animal::~Animal()"<<endl;
}
```

◆ 狗类

dog.h

```
#ifndef DOG_H
#define DOG_H
#include "animal.h"


class Dog:public Animal
{
public:
    Dog();
    ~Dog();
    virtual void voice();
};

#endif // DOG_H
```

dog.cpp

```
#include "dog.h"
#include <iostream>
using namespace std;

Dog::Dog()
{
    cout<<"Dog::Dog()"<<endl;
}

Dog::~Dog()
{
    cout<<"Dog::~Dog()"<<endl;
}

void Dog:: voice()
{
    cout<<"wang wang"<<endl;
}
```

◆ 猫类

cat.h

```
#ifndef CAT_H
#define CAT_H

#include "animal.h"


class Cat:public Animal
{
public:
    Cat();
    ~Cat();

    virtual void voice();
};

#endif // CAT_H
```

cat.cpp

```
#include "cat.h"
#include <iostream>

using namespace std;
Cat::Cat()
{
    cout<<"Cat::Cat()"<<endl;
}
Cat::~Cat()
{
    cout<<"Cat::~Cat()"<<endl;
}
void Cat::voice()
{
     cout<<"miao miao "<<endl;
}
```

◆ 测试

```
int main()
```

```
{
//    Animal ani; 抽象基类，不能实例化。
    Animal * pa = new Dog;
    pa->voice();

    delete pa;

    cout<<"---------------"<<endl;

    pa = new Cat;
    pa->voice();
    delete pa;
    return 0;
}
```

8.4.3.设计模式--听妈妈讲故事

C++中有一种设计原则叫依赖倒置。也是基于多态的。

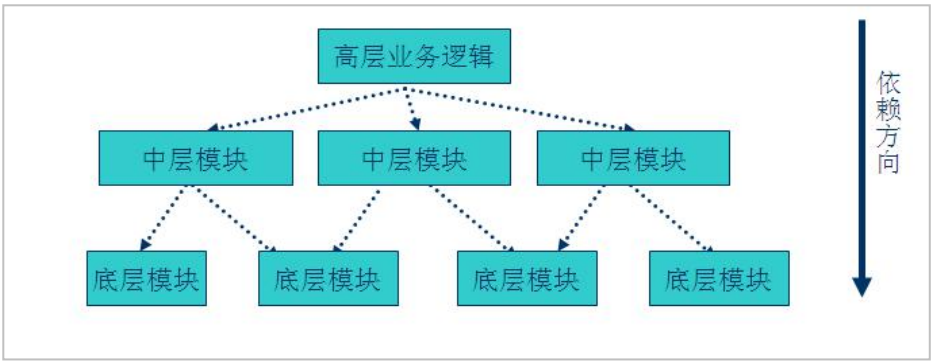
定义：高层模块不应该依赖低层模块，二者都应该依**赖其抽象**；抽象不应该依赖细节；细节应该依赖抽象。

问题由来：类 A 直接依赖类 B，假如要将类 A 改为依赖类 C，则必须通过修改类 A 的代码来达成。这种场景下，类 A 一般是高层模块，负责复杂的业务逻辑；类 B 和类 C 是低层模块，负责基本的原子操作；假如修改类 A，会给程序带来不必要的风险。

解决方案：将类 A 修改为依赖接口 I，类 B 和类 C 各自实现接口 I，类 A 通过接口 I 间接与类 B 或者类 C 发生联系，则会大大降低修改类 A 的几率。

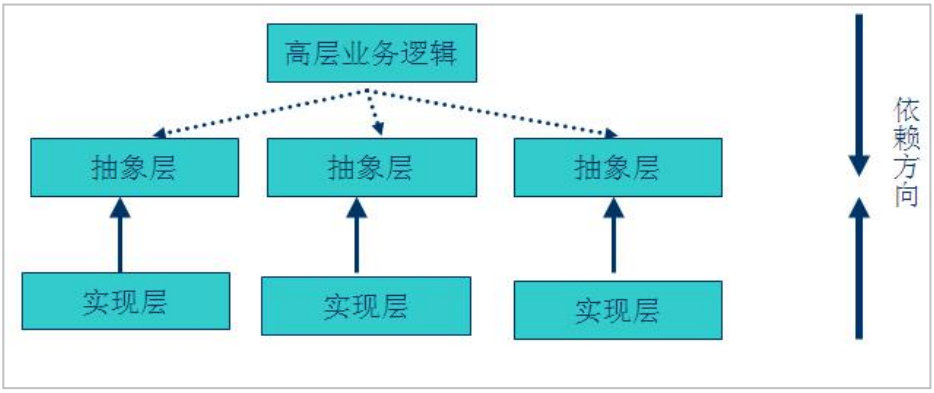
依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在 c++中，抽象指的是抽象类(c++中称为接口)，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

图示: 依赖倒置原则



传统的过程式设计倾向于使高层次的模块依赖于低层次的模块，抽象层依赖于具体的

层次。



依赖倒置原则的核心思想是**面向接口编程**，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

```
#include <iostream>

using namespace std;

class Book
{
public:
    string getContents()
    {
        return "从前有座山，山里有座庙，庙里有个小和沿，要听故事";
    }
};

class Mother
{
public:
    void tellStroy(Book *b)
    {
        cout<<b->getContents()<<endl;
    }
};

int main()
{
    Mother m;
    Book *b = new Book;
    cout<<"Mother start to tell story"<<endl;
    m.tellStroy(b);

    delete b;
    return 0;
}
```

运行结果：

```
Mather start to tellstory:
从前有座山，山里有座庙，庙里有个小和沿，要听故事运行良好
```

假如有一天,需求变成这样:不是给书而是给一份报纸,让这位母亲讲一下报纸上的故事，报纸的代码如下：

```
class NewsPaper
{
public:
    string getContents()
    {
        return "希拉里，赢得的下一届美国总统大选";
    }
};
```

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改 Mother 才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改 Mother，这显然不是好的设计。原因就是 Mother 与 Book 之间的耦合性太高了，必须降低他们之间的耦合度才行。

我们引入一个抽象的接口 IReader。读物，只要是带字的都属于读物：

```
class IReader // InterfaceReader
{
public:
    virtual string getContent() = 0;
};
```

Mother 类与**接口 IReader** 发生依赖关系，而 Book 和 Newspaper 都属于读物的范畴，他们各自都去实现 IReader 接口，这样就符合依赖倒置原则了，代码修改为：

```
#include <iostream>

using namespace std;

class IReader
{
public:
    virtual string getContents() = 0;
};

class Book :public IReader
{
public:
    string getContents()
    {
        return "从前有座山，山里有座庙，庙里有个小和尚，要听故事";
    }
};

class NewsPaper:public IReader
```

```
{
public:
    string getContents()
    {
        return "希拉里，赢得的下一届美国总统大选";
    }
};

class Mother
{
public:
    void tellStroy(IReader *i)
    {
        cout<<i->getContents()<<endl;
    }
};

int main()
{
    Mother m;
    Book *b = new Book;
    NewsPaper *n = new NewsPaper;
    cout<<"Mother start to tell story"<<endl;
    m.tellStroy(b);
    cout<<"Mother start to tell news"<<endl;
    m.tellStroy(n);

    delete b;
    return 0;
}
```

运行结果：

```
Mother start to tell story
从前有座山，山里有座庙，庙里有个小和沿，要听故事
Mother start to tell news
希拉里，赢得的下一届美国总统大选
```

这样修改后，无论以后怎样扩展 Client 类，都不需要再修改 Mother 类了。这只是一个简单的例子，实际情况中，**代表高层模块的 Mother 类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。**

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本 Mother 类与 Book 类直接耦合时，Mother 类必须等 Book 类编码完成后才可以进行编码，因为 Mother 类依赖于 Book 类。修改后的程序则可以同时开工，互不影响，因为 Mother 与 Book 类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的 TDD 开发模式就是依赖倒置原则最成功的应用。

依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

8.4.4.组装电脑系统

原理同上。依赖倒置原则。

```
#include <iostream>

using namespace std;

class HardDisk
{
public:
    virtual void run() = 0;
};

class Memory
{
public:
    virtual void run() = 0;
};

class Cpu
{
public:
    virtual void run() =0 ;
};


class WDHardDisk:public HardDisk
{
public:
    void run()
    {
        cout<<"我是西数硬盘，500g 5400r/m"<<endl;
    }
};

class IntelCpu:public Cpu
{
public:
    void run()
    {
        cout<<"我是 intel Cpu,3.4gh "<<endl;
    }
};

class kingStonMem:public Memory
{
public:
    void run()
    {
        cout<<"我是金士顿内存，16g 1333"<<endl;
    }
```

```
};


class Computer
{
public:
    Computer( Cpu *c,Memory *m,HardDisk *d)
        :disk(d),mem(m),cpu(c){}

    void work()
    {
        cpu->run();
        mem->run();
        disk->run();
    }

private:
    HardDisk *disk;
    Memory   * mem;
    Cpu      * cpu;
};

int main()
{
    IntelCpu *pc = new IntelCpu;
    kingStonMem *pm = new kingStonMem;
    WDHardDisk *ph = new WDHardDisk;

    Computer cpt(pc,pm,ph);
    cpt.work();

    delete pc;
    delete pm;
    delete ph;

    return 0;
}
```

8.4.5.企业员工信息管理系统

8.4.5.1.需求

一个小型公司的人员信息管理系统

某小型公司，主要有四类人员：经理、技术人员、销售经理和推销员。现在，需要存储这些人员的姓名、编号、级别、当月薪水．计算月薪总额并显示全部信息。

人员编号基数为 1000，每输入一个人员信息编号顺序加 1。

程序要有对所有人员提升级别的功能。本例中为简单起见，所有人员的初始级别均为 1 级。然后进行升级，经理升为 4 级，技术人员和销售经理升为 3 级，推销员仍为 1 级。

月薪计算办法是： 经理拿固定月薪 8000 元 ;技术人员按每小时 100 元领取月薪； 推销员的月薪按该推销员当月销售额的 4%提成；销售经理既拿固定月薪也领取销售提成，固定月薪为 5000 元，销售提成为所管辖部门当月销售总额的 5%。

8.4.5.2.详细设计

父类

属性：

string 姓名、int 编号、int 级别、float 当月薪水

static startNum;

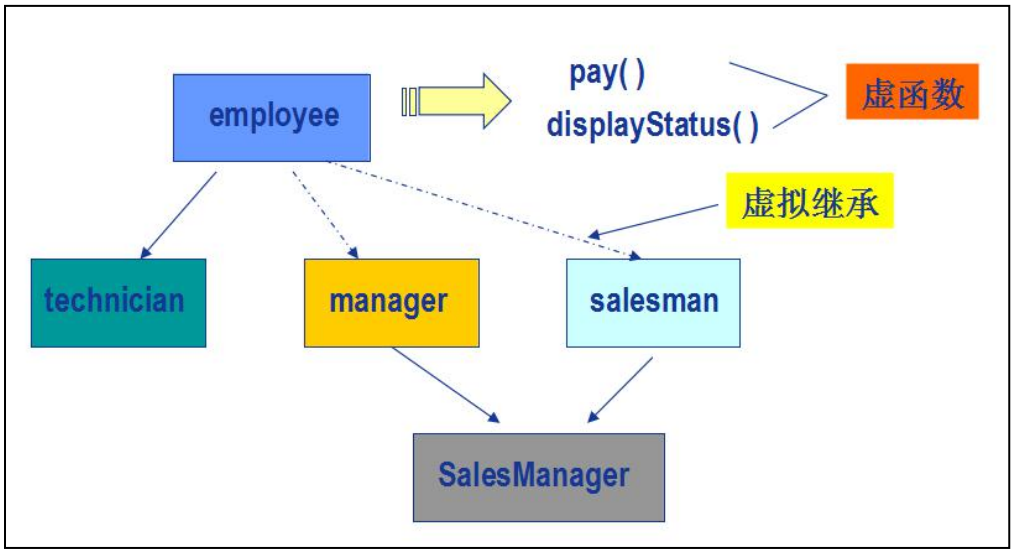
行为:

void getPay()计算月薪总额

void disInfor()显示全部信息

职位	薪酬
经理	固定月薪 8000 元
技术人员	按每小时 100 元领取月薪
推销员	当月销售额的 4%提成
销售经理	固定月薪为 5000 元，销售提成为所管辖部门当月销售总额的 5%

8.4.5.3.类设计



8.4.5.4. 代码实现

◆employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <iostream>
using namespace std;


class Employee
{
public:
    Employee();
    virtual ~Employee();

    virtual void promote(int increment = 0) =0;
    virtual void getPay() = 0;
    virtual void disInfor() = 0;

protected:
    string name;
    int num;
    int grade;
    float  salary;

    static int  startNum;

};


#endif // EMPLOYEE_H
```

◆manager.h

```
#ifndef MANAGER_H
#define MANAGER_H
#include "employee.h"


class Manager:virtual public Employee
{
public:
    Manager();
    ~Manager();

    void promote(int increment = 0);
    void getPay();
    void disInfor();

protected:
    float fixSalay;
};

#endif // MANAGER_H
```

◆**technician.h**

```
#ifndef TECHNICIAN_H
#define TECHNICIAN_H
#include "employee.h"


class Technician:public Employee
{
public:
    Technician();
    ~Technician();

    virtual void promote(int increment = 0) ;
    virtual void getPay() ;
    virtual void disInfor() ;
private:
    int moneyPerHour;
    int hourCount;
};

#endif // TECHNICIAN_H
```

◆**saleman.h**

```
#ifndef SALESMAN_H
#define SALESMAN_H
#include "employee.h"


class SalesMan:virtual public Employee
{
public:
    SalesMan();
    ~SalesMan();

    virtual void promote(int increment = 0) ;
    virtual void getPay();
    virtual void disInfor();

protected:
    float saleAmount;
    float persent;
};

#endif // SALESMAN_H
```

◆**salesmanager.h**

```
#ifndef SALEMANAGER_H
#define SALEMANAGER_H
```

```
#include "salesman.h"
#include "manager.h"


class SaleManager:public SalesMan,public Manager
{
public:
    SaleManager();
    ~SaleManager();

    void promote(int increment = 0);
    void getPay();
    void disInfor();

};

#endif // SALEMANAGER_H
```

◆main.cpp

```
#include <iostream>
#include "employee.h"
#include "technician.h"
#include "manager.h"
#include "salesman.h"
#include "salemanager.h"

using namespace std;

int main()
{
    Employee * p[] = {new Manager,new Technician,,new SalesMan,new SaleManager};
    for(int i=0; i<sizeof(p)/sizeof(p[0]); i++)
    {
        p[i]->promote(i);
        p[i]->getPay();
        cout<<"-----------------"<<endl;
        p[i]->disInfor();
        cout<<"-----------------"<<endl;

    }

    return 0;
}
```

优化补充：

8.4.5.5.补充

```
#include <iostream>

using namespace std;

class A
```

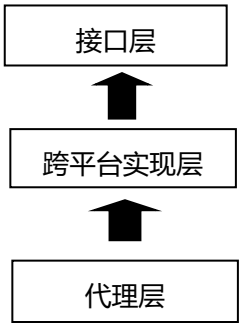
```
{
public:
    virtual void func()
    {
        cout<<"aaaaaaaaaaa"<<endl;
    }
    int a;
};
class B:virtual public A
{
public:
//    void func()
//    {
//        cout<<"bbbbbbbbbbb"<<endl;
//    }

};

class C:virtual public A
{
public:

//    void func()
//    {
//        cout<<"ccccccccccc"<<endl;
//    }
};

class D:public B,public C
{
public:
//    void func()
//    {
//        cout<<"dddddddddddd"<<endl;
//    }
    void func()
    {
        cout<<"dddddddddddd"<<endl;
    }

};

int main()
{
    D d;
    B *pb = &d;
    pb->func();

    C *pc = &d;
    pc->func();

    A * pa = &d;
    pa->func();

    return 0;
}
```

错误信息:

```
error: no unique final overrider for 'virtual void A::func()' in 'D'
class D:public B,public C
      ^
```

8.4.6.cocos跨平台入口分析

8.4.6.1.代理跨平台设计原理



8.4.6.2.类设计

8.4.6.3.代码实现

- **CCApplicationProtocol.h**

```
#ifndef CCAPPLICATIONPROTOCOL_H_
#define CCAPPLICATIONPROTOCOL_H_

class CCApplicationProtocol {
public:
CCApplicationProtocol(){}
virtual ~CCApplicationProtocol(){}

virtual bool applicationDidFinishLaunching() = 0;
};

#endif /* CCAPPLICATIONPROTOCOL_H_ */
```

- **AppDelegate.h**

```
#ifndef APPDELEGATE_H_
```

```
#define APPDELEGATE_H_

#include "CCApplication.h"

class AppDelegate: private CCApplication {
public:
    AppDelegate();
    virtual ~AppDelegate();
    bool applicationDidFinishLaunching();
};

#endif /* APPDELEGATE_H_ */
```

◆ **AppDelegate.cpp**

```
#include "AppDelegate.h"
#include "iostream"
using namespace std;

AppDelegate::AppDelegate() {
// TODO Auto-generated constructor stub

}

AppDelegate::~AppDelegate() {
// TODO Auto-generated destructor stub
}

bool AppDelegate::applicationDidFinishLaunching()
{
cout<<"star game"<<endl;
}
```

◆**CCApplication.h**

```
#ifndef CCAPPLICATION_H_
#define CCAPPLICATION_H_

#include "CCApplicationProtocol.h"

class CCApplication: public CCApplicationProtocol {
public:
    CCApplication();
    virtual ~CCApplication();

    int run();

    static CCApplication * sharedApplication();
    static CCApplication * sm_pSharedApplication;
};

#endif /* CCAPPLICATION_H_ */
```

◆ **CCApplication.cpp**

```
#include "CCApplication.h"
#include "stddef.h"

CCApplication * CCApplication::sm_pSharedApplication = NULL;

CCApplication::CCApplication() {
// TODO Auto-generated constructor stub

    sm_pSharedApplication = this;

}

CCApplication::~CCApplication() {
// TODO Auto-generated destructor stub
}

CCApplication * CCApplication::sharedApplication()
{
    if(sm_pSharedApplication != NULL)
        return sm_pSharedApplication;
}

int  CCApplication::run()
{
    applicationDidFinishLaunching();
}
```

◆ **main.cpp**

```
#include "AppDelegate.h"
#include "CCApplication.h"

int main()
{
    AppDelegate app;

    return CCApplication::sharedApplication()->run();
}
```

8.4.7.实现一个简单渲染树

提示如下:

```
#include <iostream>
using namespace std;
class ShapeTree
{
```

```
public:
    virtual void show() = 0;
    static void render();
    static void initShapeTree();
    static void destroyShapeTree();
protected:
    static ShapeTree * head;
    int x, y;
};


class Circle:public ShapeTree
{

    int radius;
};
class Rect:public ShapeTree
{

    int len;
    int wid;
};
class ellipse:public ShapeTree
{

   int longradius;
   int shortradius;
};

int main()
{
    ShapeTree::initShapeTree();  //初始化,带头节点
    //对象生成于堆上。
    while(1)
    {
        sleep(1);
        ShapeTree::render();
    }
}
```

8.5.运行时类型信息(RTTI)

typeid　dynamic_cast 是 C++ 运行时类型信息 RTTI(run time type identificaiton）重要组成部分。**运行时信息，来自于多态，所以以下运算符只用于基于多态的继承体系中。**

8.5.1.typeid

运算符 typeid 返回包含操作数数据类型信息的 type_info 对象的一个引用，信息中包括

数据类型的名称，要使用 typeid,程序中需要包含头文件<typeinfo>。

其中 type_info 重载了操作符==, !=,分别用来比较是否相等、不等，函数 name()返回类型名称。type_info 的拷贝和赋值均是私有的，故不可拷贝和赋值。

常用于返回检查,调试之用。

```
#include <iostream>
#include <typeinfo>

using namespace std;

typedef void (*Func)();


class Base
{
public:
    virtual  ~Base(){}
};

class Derive:public Base
{

};

int main()
{

    cout<<typeid(int).name()<<endl;
    cout<<typeid(double).name()<<endl;

    cout<<typeid(char *).name()<<endl;
    cout<<typeid(char **).name()<<endl;

    cout<<typeid(const char *).name()<<endl;
    cout<<typeid(const char * const ).name()<<endl;

    cout<<typeid(Func).name()<<endl;

    cout<<typeid(Base).name()<<endl;
    cout<<typeid(Derive).name()<<endl;

    Derive d;
    Base &b = d;    //Base 中没有虚函数时，有时？

    cout<<typeid(b).name()<<endl;
    cout<<typeid(d).name()<<endl;


    Base *p = &d;
    cout<<typeid(p).name()<<endl; //判断指针是，其实是看不出其类型信息的
    cout<<typeid(*p).name()<<endl;
    cout<<typeid(d).name()<<endl;

    cout<<boolalpha<<(typeid(*p)== typeid(d))<<endl;
    return 0;
}
```

多态下使用 typeid 时要注意的问题:

1).确保基类定义了至少一个虚函数(虚析构也算)。
2).不要将 typeid 作用于指针，应该作用于引用，或解引用的指针。
3)typeid 是一个运算符，而不是函数。
4)typeid 运算符返回的 type_info 类型，其拷贝构造函数和赋值运算函数都声明为 private 了 ,这意味着其不能用于 stl 容器 ,所以我们一般不能不直接保存 type_info 信息，而保存 type_info 的 name 信息

注解：

Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class Base *). However, when typeid is applied to objects (like *a and *b) typeid yields their dynamic type (i.e. the type of their most derived complete object).

If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeidthrows a bad_typeid exception.

8.5.2.typecast

8.5.2.1.static_cast

在一个方向上可以作隐式转换的 ,在另外一个方向上可以作静态转换。发生在编译阶段，不保证后序使用的正确性。

8.5.2.2.reinterpreter_cast

既不在编译器期也不在运行期进行检查，安全性完全由程序员决定。

8.5.2.3.dynamic_cast

dynamic_cast 一种运行时的类型转化方式,所以要在运行时作转换判断。检查指针所指类型，然后判断这一类型是否与正在转换成的类型有一种 “is a” 的关系，如果是，dynamic_cast 返回对象地址。如果不是，dynamic_cast 返回 NULL。

dynamic_cast 常用多态继承中，判断父类指针的真实指向。

```
#include <iostream>
#include <typeinfo>

using namespace std;


class A
{
public:
    virtual ~A(){}
```

```
};

class B:public A
{

};

class C:public A
{

};

class D
{

};


int main()
{
    B b;
    A  *pa = &b;


    B *pb = dynamic_cast<B*>(pa);  //成功
    cout<<pb<<endl;
    C *pc = dynamic_cast<C*>(pa);  //成功 安全
    cout<<pc<<endl;
    D *pd = dynamic_cast<D*>(pa);  //成功 安全
    cout<<pd<<endl;

    pb = static_cast<B*>(pa);      //成功
    cout<<pb<<endl;
    pc = static_cast<C*>(pa);      //成功  不安全
    cout<<pc<<endl;
//    pd = static_cast<D*>(pa);      //编译 不成功
//    cout<<pd<<endl;

    pb = reinterpret_cast<B*>(pa);      //成功 不安全
    cout<<pb<<endl;
    pc = reinterpret_cast<C*>(pa);      //成功  不安全
    cout<<pc<<endl;
    pd = reinterpret_cast<D*>(pa);      //成功  不安全
    cout<<pd<<endl;
    return 0;
}
```

8.5.3.RTTI应用

前面设计的公司系统中，如果要实现对特定的员工加薪，该如何设计呢。main 函数中拥有了成有成员统一的类型信息(Employee*),又如何体现，特定的员工呢。

假设要增加经理的固定薪水，该哪何作呢?

8.6.多态实现浅浅析

8.6.1.虚函数表

C++的多态是通过一张虚函数表（Virtual Table）来实现的，简称为 V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆写的问题，保证其真实反应实际的函数。这样，在有虚函数的类的实例中**这个表被分配在了这个实例的内存中**，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

这里我们着重看一下这张虚函数表。C++的编译器应该是保证虚函数表的指针存在于对象实例中**最前面的位置**（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

假设我们有这样的一个类：

```
class Base {
public:
    virtual void f() { cout << "Base::f" << endl; }
    virtual void g() { cout << "Base::g" << endl; }
    virtual void h() { cout << "Base::h" << endl; }

};
```

按照上面的说法，我们可以通过 Base 的实例来得到虚函数表。 下面是实际例程：

```
int main()
{
    Base b;
    cout<<&b<<endl;

    cout<<"基类 b 的起始地址:"<<(int*)(&b)<<endl;
    cout<<"VTalbe 的起始地址:"<<(int**)(int*)(&b)<<endl;



    FUNC pf = NULL;
    pf = (FUNC)*((int**)*(int*)(&b)+0);
    pf();
```

```
    pf = (FUNC)*((int**)*(int*)(&b)+1);
    pf();
    pf = (FUNC)*((int**)*(int*)(&b)+2);
    pf();


    return 0;
}
```

实际运行结果如下

```
4
基类 b 的起始地址:0x28fea8
VTalbe 的起始地址:0x28fea8
Base::f
Base::g
Base::h
```

通过这个示例，我们可以看到，我们可以通过强行把&b 转成 int *，取得虚函数表的地址，然后，再次取址就可以得到第一个虚函数的地址了，也就是 Base::f()，这在上面的程序中得到了验证（把 int* 强制转成了函数指针）。通过这个示例，我们就可以知道如果要调用 Base::g()和 Base::h()，其代码如下：

```
 (Fun)*((int**)*(int*)(&b)+0);  // Base::f()
 (Fun)*((int**)*(int*)(&b)+1);  // Base::g()
 (Fun)*((int**)*(int*)(&b)+2);  // Base::h()
```

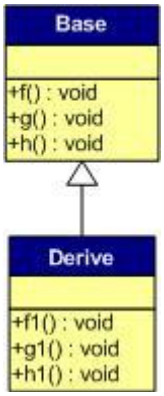
画个图解释一下。如下所示：

注意：在上面这个图中，我在虚函数表的最后多加了一个结点，这是虚函数表的结束结点，就像字符串的结束符“/0”一样，其标志了虚函数表的结束。这个结束标志的值在不同的编译器下是不同的。

下面，我将分别说明“无覆写”和“有覆写”时的虚函数表的样子。没有覆写父类的虚函数是毫无意义的。我之所以要讲述没有覆写的情况，主要目的是为了给一个对比。在比较之下，我们可以更加清楚地知道其内部的具体实现。

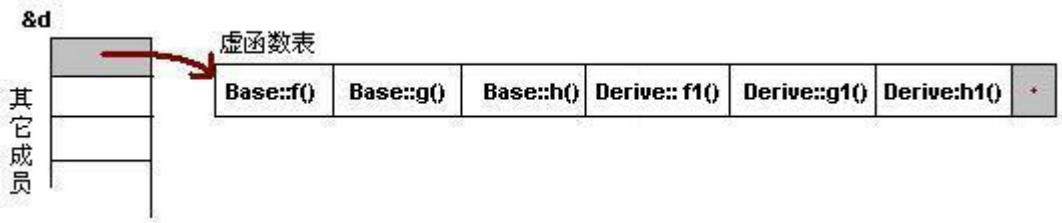
8.6.2.一般继承（无虚函数覆写）

下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：



请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，其虚函数表如下所示：

对于实例：Derive d; 的虚函数表如下：



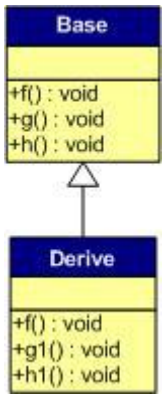
我们可以看到下面几点：

1）虚函数按照其声明顺序放于表中。

2）父类的虚函数在子类的虚函数**前面。**

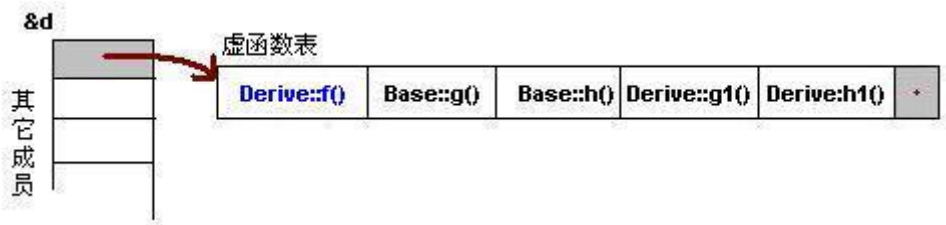
8.6.3.一般继承（有虚函数覆写）

覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。



为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：f()。

那么，对于派生类的实例，其虚函数表会是下面的一个样子：



我们从表中可以看到下面几点，

1）覆写的 f()函数被放到了虚表中原来父类虚函数的位置。

2）没有被覆盖的函数依旧。

这样，我们就可以看到对于下面这样的程序，

```
Base *b = new Derive();

b->f();
```

由 b 所指的内存中的**虚函数表的 f()的位置已经被 Derive::f()函数地址所取代**，于是在实际调用发生时，是 Derive::f()被调用了。这就实现了多态。

8.6.4.静态代码发生了什么？

```
Base *b = new Derive();
b->f();
```

当编译器看到这段代码的时候，并不知道 b 真实身份。编译器能作的就是用一段代码代替这段语句。

1，明确 b 类型。

2，然后通过指针虚函数表的指针 **vptr 和偏移量,**匹配虚函数的入口。

3，根据入口地址调用虚函数。

8.6.5.评价多态

1，实现在动态绑定。

2，些许的牺牲了一些空间和效率。

8.6.6.常见问答

◆为什么虚函数必须是类的成员函数：

虚函数诞生的目的就是为了实现多态，在类外定义虚函数毫无实际用处。

◆为什么类的静态成员函数不能为虚函数：

如果定义为虚函数，那么它就是动态绑定的，也就是在派生类中可以被覆盖的，这与静态成员函数的定义（：在内存中只有一份拷贝；通过类名或对象引用访问静态成员）本身就是相矛盾的。

◆为什么构造函数不能为虚函数：

因为如果构造函数为虚函数的话，它将在执行期间被构造，而执行期则需要对象已经建立，构造函数所完成的工作就是为了建立合适的对象，因此在没有构建好的对象上不可能执行多态（虚函数的目的就在于实现多态性）的工作。在继承体系中，构造的顺序就是从基类到派生类，其目的就在于确保对象能够成功地构建。**构造函数同时承担着虚函数表的建立，如果它本身都是虚函数的话，如何确保 vtbl 的构建成功呢？**

注意：当基类的构造函数内部有虚函数时，会出现什么情况呢？结果是在**构造函数中，虚函数机制不起作用了**，调用虚函数如同调用一般的成员函数一样。当基类的析构函数内部有虚函数时，又如何工作呢？与构造函数相同，只有“局部”的版本被调用。但是，行为相同，原因是不一样的。**构造函数只能调用“局部”版本，是因为调用时还没有派生类版本的信息。析构函数则是因为派生类版本的信息已经不可靠了。**我们知道，析构函数的调用顺序与构造函数相反，是从派生类的析构函数到基类的析构函数。当某个类的析构函数被调用时，其派生类的析构函数已经被调用了，相应的数据也已被丢失，如果再调用虚函数的派生类的版本，就相当于对一些不可靠的数据进行操作，这是非常危险的。因此，**在析构函数中，虚函数机制也是不起作用的。**

```
#include <iostream>

using namespace std;

class A
{
public:
    A()
    {
        p = this;
        p->func();
    }
    virtual void func()
    {
        cout<<"aaaaaaaaaaaaaaa"<<endl;
    }

private:
    A *p;
};

class B:public A
{
public:
```

```
    void func()
    {
        cout<<"bbbbbbbbbbbbbbbb"<<endl;
    }

};

int main()
{
    B b;
    return 0;
}
```

8.6.7.练习

下面代码输出什么？

```
#include <iostream>

using namespace std;

class Base
{
public:

    void foo()
    {
        func();
    }

    virtual void func()
    {
        cout<<"Base"<<endl;
    }

};

class Derive:public Base
{
private:
    void func()
    {
        cout<<"Derive"<<endl;
    }
};

int main()
{
    Derive d;
    d.foo();
    return 0;
}
```

9.模板(Templates)

泛型(Generic Programming)即是指具有在多种数据类型上皆可操作的含意。泛型编程的代表作品 STL 是一种高效、泛型、可交互操作的软件组件。

泛型编程最初诞生于 C++中，目的是为了实现 C++的 STL（标准模板库）。其语言支持机制就是模板（Templates）。模板的精神其实很简单：参数化类型。换句话说，把一个原本特定于某个类型的算法或类当中的类型信息抽掉，抽出来做成模板参数 T。

9.1.函数模板

9.1.1.函数重载实现的泛型

```
void Swap(int & a, int &b)
{
    int t = a;
    a = b;
    b = t;
}

void Swap(double & a, double &b)
{
    double t = a;
    a = b;
    b = t;
}


int main()
{
    long a = 2; long b = 3;
    Swap(a,b);
    cout<<a<<b<<endl;

    return 0;
}
```

9.1.2.函数模板的引入

◆语法格式

```
template<typename/class 类型参数表>
返回类型 函数模板名(函数参数列表)
{
    函数模板定义体
}
```

template 是语义是模板的意思，尖括号中先写关键字 typename 或是 class ，后面跟一个类型 T,此类即是虚拟的类型。至于为什么用 T,用的人多了，也就是 T 了。

9.1.3.函数模板的实例

调用过程是这样的，先将函数模板实再化为函数，然后再发生函数调用。

```
#include <iostream>

using namespace std;

template <typename T>
void Swap(T& a,T &b )
{
    T t = a;
    a = b;
    b = t;
}



int main()
{
    int ia = 10; int ib = 20;
    Swap(ia,ib);        //Swap<int>(ia,ib);

    cout<<ia<<ib<<endl;

    double da = 10, db = 20;
    Swap(da,db);        //Swap<double>(da,db);
    cout<<da<<db<<endl;

    string sa ="china"; string sb = "America";
    Swap(sa,sb);
    cout<<sa<<sb<<endl;
    return 0;
}
```

9.1.4.小结

函数模板，只适用于函数的**参数个数相同而类型不同，且函数体相同的情况。**如果个数不同，则不能用函数模板。

9.2.类模板

9.2.1.引例

Stack 类

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

using namespace std;

class Stack
{
```

```
public:
    Stack(int size=1024);
    ~Stack();
    bool isEmpty();
    bool isFull();
    void push(int data);
    int pop();

private:
    int* space;
    int top;
};

Stack::Stack(int size)
{
    space = new int[size];
    top = 0;
}

Stack::~Stack()
{
    delete []space;
}
bool Stack::isEmpty()
{
    return top == 0;
}
bool Stack::isFull()
{
    return top == 1024;
}

void Stack::push(int data)
{
    space[top++] = data;
}
int Stack::pop()
{
    return space[--top];
}

int main()
{

    Stack s(100);
    if(!s.isFull())
        s.push(10);
    if(!s.isFull())
        s.push(20);
    if(!s.isFull())
        s.push(30);
    if(!s.isFull())
        s.push(40);
    if(!s.isFull())
        s.push(50);

    while(!s.isEmpty())
        cout<<s.pop()<<endl;
```

```
    return 0;
}
```

上述栈，如果想模板化，可以 push 和 pop 不同的数据类型。主要由几个因素需要把控。**栈中的空间元素类型，压入元素类型，弹出元素类型**，三者保持一致即可。

9.2.2.类模板语法

9.2.2.1.类模板定义

```
template<typename T>
class Stack
{

}
```

9.2.2.2.类内定义成员函数

```
template<typename T>
class Stack
{
public:
    Stack(int size)
    {
        space = new T[size];
        top = 0;
    }
 }
```

9.2.2.3.类外定义函数

```
template<typename T>
void Stack<T>::push(T data)
{
    space[top++] = data;
}
```

9.2.2.4.类模板实例化为模板类

类模板是类的抽象，类是类模板的实例。

```
Stack<double> s(100);
```

9.2.3.类模板实例

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
using namespace std;

template<typename T>
class Stack
{
public:
    Stack(int size)
    {
        space = new T[size];
        top = 0;
    }
    ~Stack();
    bool isEmpty();
    bool isFull();
    void push(T data);
    T pop();

private:
    T* space;
    int top;
};

template<typename T>
Stack<T>::~Stack()
{
    delete []space;
}

template<typename T>
bool Stack<T>::isEmpty()
{
    return top == 0;
}

template<typename T>
bool Stack<T>::isFull()
{
    return top == 1024;
}
template<typename T>
void Stack<T>::push(T data)
{
    space[top++] = data;
}
template<typename T>
T Stack<T>::pop()
{
    return space[--top];
}

int main()
{

    Stack<double> s(100);  //Stack<string> s(100);


    if(!s.isFull())
        s.push(10.3);
```

```
    if(!s.isFull())
        s.push(20);
    if(!s.isFull())
        s.push(30);
    if(!s.isFull())
        s.push(40);
    if(!s.isFull())
        s.push(50);

    while(!s.isEmpty())
        cout<<s.pop()<<endl;


    return 0;
}
```

9.2.4.练习

模拟 STL 中 vector 的用法，自己实现之。

```
#ifndef MYVECTOR_HPP
#define MYVECTOR_HPP


#include <iostream>
using namespace std;


template <typename T> class MyVector;
template <typename T> ostream & operator<<(ostream &out, const MyVector<T> &obj);


template <typename T>
class MyVector
{
public:
    MyVector(int size = 0);
    MyVector(const MyVector<T> &obj);
    MyVector<T> & operator=( MyVector<T> &obj);
    ~MyVector();

    T& operator[] (int index);

    int getSize();
    friend ostream & operator<< <T>(ostream &out, const  MyVector<T> &ob
j);

protected:
    T *m_space;
    int m_len;
};

template <typename T>
int MyVector<T>::getSize()
{
    return m_len;
```

```
}


template <typename T>
ostream & operator<< (ostream &out, const MyVector<T> &obj)
{
    for (int i=0; i< obj.m_len; i++)
    {
        out << obj.m_space[i] << " ";
    }
    out << endl;
    return out;
}



template <typename T>
MyVector<T>::MyVector(int size)
{
    m_space = new T[size];
    m_len = size;
}


template <typename T>
MyVector<T>::MyVector(const MyVector & obj)
{
    m_len = obj.m_len;
    m_space = new T[m_len];

    for (int i=0; i<m_len; i++)
    {
        m_space[i] = obj.m_space[i];
    }

}

template <typename T>
MyVector<T>::~MyVector() //析构函数
{
    if (m_space != NULL)
    {
        delete [] m_space;
        m_space = NULL;
        m_len = 0;
    }
}

template <typename T>
T& MyVector<T>::operator[] (int index)
{
    return m_space[index];
}

template <typename T>
MyVector<T> &  MyVector<T>::operator=( MyVector<T> &obj)
{
    delete[] m_space;
    m_space = NULL;
```

```
    m_len = 0;

    m_len = obj.m_len;
    m_space = new T[m_len];

    for (int i=0; i<m_len; i++)
    {
        m_space[i] = obj[i];
    }
    return *this;
}



#endif // MYVECTOR_HPP
```

测试代码：

```
#include <iostream>
#include "myvector.hpp"
using namespace std;

int main()
{
    MyVector<int> a(10);

    for(int i=0; i<10; i++)
    {
        a[i] = i+100;
    }

    MyVector<int> b(a);
    cout<<b[2]<<b[1]<<endl;

    MyVector<int> c;
    c = a;

    cout<<b[2]<<b[1]<<endl;

    cout<<a;

    return 0;
}
```

注：

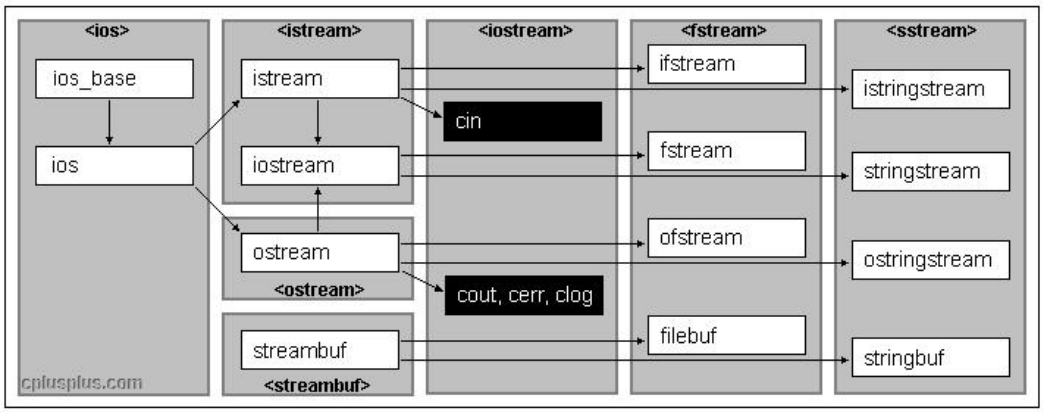
《C++编程思想》第 15 章(第 300 页)：

模板定义很特殊。由 template<...> 处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，**几乎总是在头文件中放置全部的模板声明和定义,文件后缀为.hpp。**

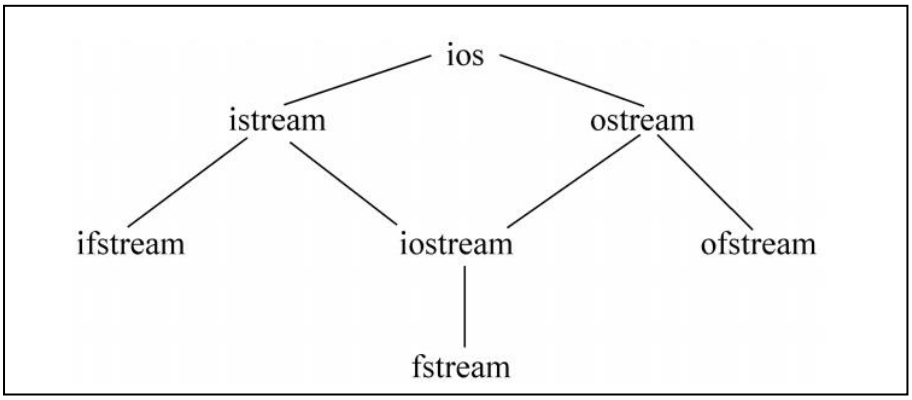
10. 输入输出 IO 流

cin 和 cout 充当了 scanf 和 printf 的功能。但他们并不是函数。而是类对象，那么我们有必要了解一下，他们是哪些类的对像。

10.1.io 类图关系



简化图和要掌握的流类。



iostream	istream reads from a stream ostream writes to a stream iostream reads and writes a stream; derived from istream and ostream,
fstream	ifstream, reads from a file; derived from istream ofstream writes to a file; derived from ostream fstream, reads and writes a file; derived from iostream

10.2.流类综述

10.2.1.IO 对象不可复制或赋值

```
#include <iostream>
#include <fstream>

fstream printf(fstream fs)
{

}

using namespace std;

int main()
{
    fstream fs1,fs2;
//    fs1 = fs2;

//    fstream fs3(fs2);
    return 0;
}
```

10.2.2.IO对象是缓冲的

```
int main()
{
    cout<<"aldksfj;lasdkfj;alsd";
    while(1);
    return 0;
}
```

下面几种情况会导致刷缓冲

1，程序正常结束，作为 main 函数结束的一部分，将清空所有缓冲区。

2，缓冲区满，则会刷缓冲。

3，endl, flush 也会刷缓冲。

4，在每次输出操作执行完后，用 unitbuf 操作符设置流的内部状态，从而清空缓冲区。

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    cout<<"hi"<<endl;
    cout<<"hi"<<flush;
    cout<<unitbuf<<"hi";
    while(1);

    return 0;
}
```

10.2.3.重载了<< 和 >>运算符

```
__ostream_type&
operator<<(long __n)
{ return _M_insert(__n); }

__ostream_type&
operator<<(unsigned long __n)
{ return _M_insert(__n); }
```

10.3.标准输出

C++提供的输出流对象有：

cout：输出基本类型数据时，不必考虑数据是什么类型，系统会自动判断，选择相应的重载函数；输出用户自己定义的类型数据时，要重载<<运算符。

cerr：是在屏幕上显示出错信息，与 cout 用法类似，不同的是只能在屏幕上，而不能在磁盘文件上输出错误信息；

clog：用法与 cerr 类似，不同点是它带有缓冲区。

10.3.1.iomanip

引用流算子的原因是，系统函数格式化并不好用。

```
int main()
{
    cout.unsetf(ios::dec);
    cout.setf(ios::hex);
    cout<<1234<<endl;
    cout.unsetf(ios::hex);
    cout<<1234<<endl;
    return 0;
}
```

输出就涉及到格式化，系统提供的方案特别的复杂。我们采用 iomanip 中的方案，分别示例：

控制符	功　能
dec	十进制数输出
hex	十六进制输出
oct	八进制数输出
setfill©	在给定的输出域宽度内填充字符 c
setprecison(n)	设显示小数精度为 n 位
setw(n)	设域宽为 n 个字符

setiosflags(ios::fixed)	固定的浮点显示
setiosflags(ios::scientific)	指数显示
setiosflags(ios::left)	左对齐
setiosflags(ios::right)	右对齐
setiosflags(ios::skipws)	忽略前导空白
setiosflags(ios::uppercase)	十六进制数大写输出
setiosflags(ios::lowercase)	十六进制数小写输出
setiosflags(ios::showbase)	当按十六进制输出数据时，前面显示前导符 **0x**；当按八进制输出数据时，前面显示前导符 **0**
endl	输入一个换行符并刷新流

1)用格式控制符控制输出

输出不同进制的数：dec(十进制)、hex(十六进制)、oct（八进制）

```
int n=20;
cout<<"设置进制:"<<endl;
cout<<"十进制  :"<<n<<endl;
cout<<"十六进制:"<<hex<<n<<endl;
cout<<"八进制  :"<<oct<<n<<endl;
cout<<"十进制  :"<<dec<<n<<endl;
```

2)设置域宽

setw(n), n 小于实际宽度时，按实际宽度输出，它一次只控制一个数值输出。

```
int m=1234;
cout<<"设置域宽: "<<endl;
cout<<setw(3)<<m<<endl;
cout<<setw(5)<<m<<endl;
cout<<setw(10)<<m<<endl;
```

3)设置填充字符

setfill(c)，需要与 setw(n)合用。

```
int x=1234;
cout<<"设置填充字符: "<<endl;
cout<<setfill('*')<<setw(5)<<x<<endl;
cout<<setw(10)<<x<<endl;
```

4)设置对齐方式：

setiosflags(ios::left)(左对齐)、setiosflags(ios::right)(右对齐)

```
int y=1234;
cout<<"设置对齐方式"<<endl;
```

```
    cout<<setfill(' ');
    cout<<setiosflags(ios::left)<<setw(10)<<y<<endl;
    cout<<setiosflags(ios::right)<<setw(10)<<y<<endl;
```

5) 强制显示

强制显示小数点和尾 0:setiosflags(ios::showpoint)

强制显示符号：setiosflags(ios::showpos)

```
    double d1=10/5,d2=22.0/7;
    cout<<"显示小数点、尾和数符："<<endl;
    cout<<d1<<endl;
    cout<<setiosflags(ios::showpoint)<<d1<<endl;
    cout<<setiosflags(ios::showpos)<<d2<<endl;
//  cout<<resetiosflags(ios::showpos);
    cout<<d2<<endl;
```

6）设置精度(有效数字个数)

setprecision(n)自动四舍五入

```
    double dd=123.4567;
    cout<<setprecision(2)<<dd<<endl;
    cout<<setprecision(3)<<dd<<endl;
    cout<<setprecision(4)<<dd<<endl;
    cout<<setprecision(5)<<dd<<endl;
```

7）设置浮点数的输出是以科学记数法还是定点数

setiosflags(ios::scientific)(科学记数法)，此时精度域表示小数位数

setiosflags(ios::fixed)(定点数)，此时精度域表示小数位数

setprecision(2)<<setiosflags(ios::fixed)设置小数的精度

```
    double dd=123.4567;
    cout<<setiosflags(ios::scientific)<<dd<<endl;
    cout<<resetiosflags(ios::scientific);
    cout<<setiosflags(ios::fixed)<<dd<<endl;
    cout<<setprecision(2)<<setiosflags(ios::fixed)<<dd<<endl;
```

8)输出十六进制数时控制英文字母的大小写

setiosflags(ios::uppercase)

resetiosflags(ios::uppercase)

```
int num=510;
cout<<"以大小写方式输出进制数："<<endl;
cout<<"16 进制数(默认：小写方式):"<<hex<<num<<endl;
cout<<"以大写方式输出进制数："<<setiosflags(ios::uppercase)<<hex<<num<<endl;
cout<<"恢复小写方式输出进制数："<<resetiosflags(ios::uppercase)<<hex<<num<<endl;
```

10.3.2.成员函数

格式:ostream put(char)

功能:输了一个字符。

```
#include <iostream>
using namespace std;
int main()
{
    char str[]="Programming with C++";
    for( int i=sizeof(str)/sizeof(str[0])-2 ; i>=0; i--)
        cout.put(*(str+i));
    cout.put('\n');

    return 0;
}
```

10.4.标准输入 cin

在 C++中，默认的标准输入设备是键盘，在 iostream 文件中定义了 cin 输入流对象。

cin 对象与提取运算符>>、变量名或数组名一起构成输入语句，形式为 C++的格式化输入。

cin>>…>>…>>…；,能够连续输入多项内容。只要是基本数据类型 ,不管是 int、double、float，还是 char、char *等，都可以写成这种形式，这给用户提供了很大的方便。如果要输入用户自己定义的类型数据，就要用友元方式重载>>运算符;

```
#include <iostream>

using namespace std;

int main()
{
    int a;
    double b;
    char buf[1024];  //string buf;

    cin>>a>>b>>buf;  //读入字符串时遇到空格则止 12 23.5 aa bb cc dd
    cout<<a<<endl;
    cout<<b<<endl;
    cout<<buf<<endl;
    return 0;
}
```

10.4.1.成员函数

流输入运算符,很大程序上提供了足够的便利,但是也有输入字符串,**遇空格则止**的问题。于是，为了弥补其不足，又提供了以下不可或缺的成员函数。

10.4.1.1.char get()

格式	char get()
功能	读入一个字符并返回(包括回车、tab、空格等空白字符)

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    while( (ch=cin.get()) !=EOF)   //EOF 为文件结束符，按 ctrl+d 输入
      cout.put(ch);

    return 0;
}
```

10.4.1.2.istream& get(char&)

格式	istream& get(char &)
功能	读入一个字符,如果读取成功则返回非 0 值(真),如失败(遇到文件结束符)，则函数返回 0 值(假)。

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    while(cin.get(ch)) //get 函数返回的是，istream & 可实现链编程。
    {
        cout.put(ch);
    }

    return 0;
}
```

10.4.1.3.istream& get(char *, int ,char)

格式	istream& get(字符数组，字符个数 n,终止字符) 或　istream & get(字符指针，字符个数 n,终止字符)

功能	从输入流中读取 n-1 字符，赋给字符数组或字符指针所指向的数组。如果在读取 n-1 个字符之前遇到终止字符，则提前结束。如果成功则返回非 0，失败　则返回 0。会清空 char*指向的空间，未读到 n-1 个字符或中止符，则会阻塞。**不会越过中止符。**

```
#include <iostream>
using namespace std;
int main()
{
    char str[80];
    cout<<"请输入一个字符串：";   //cin>>buf; aa  bb cc dd
    cin.get(str,80,'a');
    cout<<str<<endl;

    return 0;
}
```

10.4.2.istream& getline(char *, int , char)

格式	cin.getline(字符数组或字符指针，字符个数 n[,终止字符])
功能	与带三个参数的 get()功能类似，从输入流中读取 n-1 字符，赋给字符数组或字符指针所指向的空间。如果在读取 n-1 个字符之前遇到终止字符（如果不写，默认为’\n’），则提前结束。会清空 char*指向的空间，未读到 n-1 个字符或中止符，则会阻塞。**会越过中止符。**

```
int main()
{
    char buf[1024];

     // abcdiabcdi

//    cin.get(buf,1024,'i');
//    cout<<buf<<"---"<<endl;

//    cin.get(buf,1024,'i');
//    cout<<buf<<"---"<<endl;

    cin.getline(buf,1024,'i');
    cout<<buf<<"---"<<endl;

    cin.getline(buf,1024,'i');
    cout<<buf<<"---"<<endl;


    return 0;
}
```

get 和 getline 最大的区别就是，get 遇到界定符时，停止执行，但并不从流中提取界定符，再次调用会遇到同一个界定符，函数将立即返回，不会提取输入。getline 则不同，它将从输入流中提供界定符，但伤然不会把它放到缓冲区中。

10.4.3.ignore peek putback：

istream& ignore(streamsize n = 1, int delim = EOF);	跳过流中的 n 个字符 ,或遇到终止字符为止(包含) 默认参数忽略一个字符。
int peek();	窥视　当前指针未发生移动
istream& putback (char c);	回推　　插入当前指针位置

```
#include <iostream>

using namespace std;

int main()
{
    char ch[20];
    cin.get(ch,20,'/'); // i like c/ i like c++ also/
    cout<<"the first part is :"<<ch<<endl;
    cin.ignore(10, 'i');
    cin.putback('i');

    char peek = cin.peek();
    cout<<"peek is :"<<peek<<endl;

    cin.get(ch,20,'/');
    cout<<"this second part is:"<<ch<<endl;
    return 0;
}
```

11.文件 IO 流

11.1.C　IO 流

11.1.1. 数据流：

指程序与数据的交互是以流的形式进行的.进行 C 语言文件的存取时,都会先进行“打开文件”操作,这个操作就是在打开数据流,而“关闭文件”操作就是关闭数据流。

11.1.2. 缓冲区(Buffer)

指在程序执行时,所提供的额外内存,可用来暂时存放做准备执行的数据.它的设置是为了提高存取效率,因为内存的存取速度比磁盘驱动器快得多.

C++ 语言中带缓冲区的文件处理：

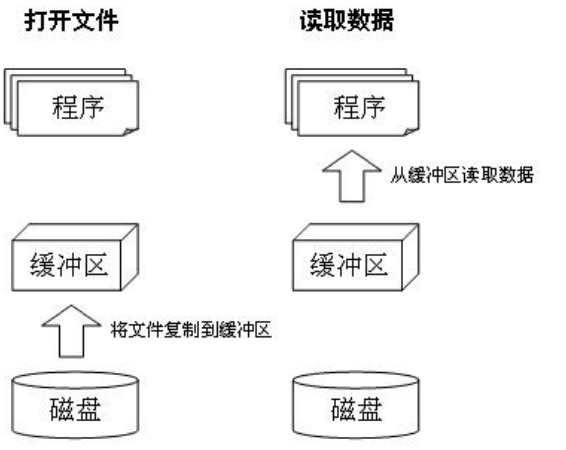
C++ 语言的文件处理功能依据系统是否设置“缓冲区”分为两种：

- 一种是设置缓冲区
- 另一种是不设置缓冲区

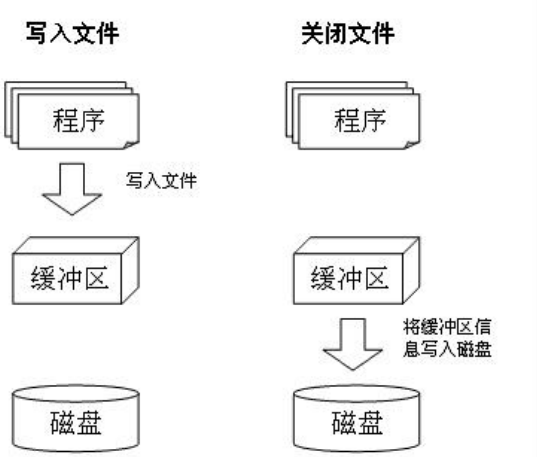
当使用标准 I/O 函数(包含在头文件 cstdio 中)时,系统会自动设置缓冲区,并通过数据流来读写文件.

当进行文件读取时,不会直接对磁盘进行读取,而是先打开数据流,将磁盘上的文件信息拷贝到缓冲区内,然后程序再从缓冲区中读取所需数据,如下图所示：

文件读：



文件写:



11.1.3.文件类型

分为文本文件和二进制文件两种.

➢文本文件:

是以字符编码的方式进行保存的,只是计算机以二进制表示数据在外部存储介质上的另一种存放形式.它所存放的每一个字节都可以转换为一个可读字符当向文件中写入数据时,windows 中一旦遇到"换行"字符(ASCII 码为 10)则会转换成"回车-换行"(ASCII 值为 13,10).在读取数据的时候,一遇到"回车-换行"的组合 ASCII 值为 13,10),则会转换成为"换行"字符(ASCII 码为 10)

➢二进制文件:

将内存中数据原封不动的读取和写入文件中。

二进制文件适用于非字符为主的数据.如果以记事本打开,只会看到一堆乱码.

当按照文本方式其实,除了文本文件外,所有的数据都可以算是二进制文件.二进制文件的优点在于存取速度快,占用空间小,以及可随机存取数据.

在文件操作中因为文本打开方式和二进制文件打开方式会导致数据读取和写入换行时候的不同所以在进行文件操作时候要注意写入和读取的方式要保持一致,如果采用文本方式写入,应该采用文本方式读取,如果采用二进制方式写入,就应该用二进制方式读取,但是不管是文本文件还是二进制文件,如果采用统一的二进制方式写入和读取数据都是不会出错的,不管是文本文件还是二进制文件,都可以采用二进制方式或者文本方式打开,然后进行写入或读取.但是对于二进制文件来说,如果以文本凡是读取数据时候可能会出现一些问题。

11.1.4.文件存取方式

包括顺序存取方式和随机存取方式两种.

■顺序读取：

也就是从上往下,一笔一笔读取文件的内容.保存数据时,将数据附加在文件的末尾.这种存取方式常用于文本文件,而被存取的文件则称为顺序文件.

■随机存取：

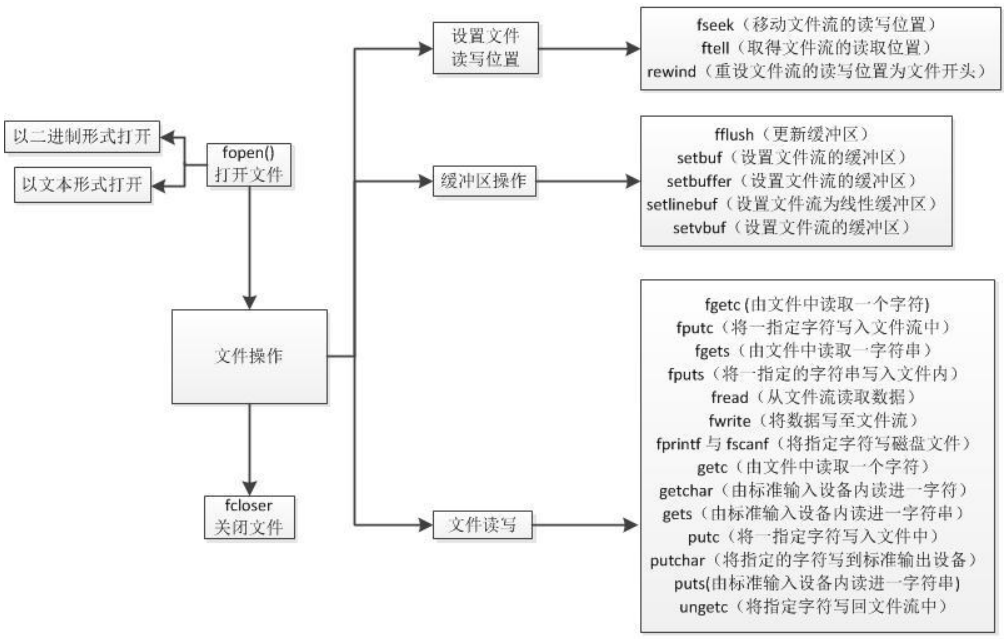
多半以二进制文件为主.它会以一个完整的单位来进行数据的读取和写入,通常以结构为单位.

11.1.5.借助文件指针读写文件

我们如果要访问文件,要借助于文本变量,即文件指针 FILE *才可以完成。

文件在进行读写操作之前要先打开,使用完毕要关闭.所谓打开文件,实际上是建立文件的各种有关信息,并使文件指针指向该文件,以便进行其它操作.关闭文件则断开指针与文件之间的联系,也就禁止再对该文件进行操作。

11.1.6.操作流图



11.2.C++ 文件 IO 流

11.2.1.引例

11.2.2.文件流类与文件流对象

对文件的操作是由文件流类完成的。文件流类在流与文件间建立连接。由于文件流分为三种 :文件输入流、文件输出流、文件输入/输出流 ,所以相应的必须将文件流说明为 ifstream、ofstream 和 fstream 类的对象，然后利用文件流的对象对文件进行操作。

对文件的操作过程可按照一下四步进行：即定义文件流类的对象、打开文件、对文件进行读写操作、关闭文件，下面分别进行介绍。

11.2.3.文件的打开和关闭

11.2.3.1.定义流对象

```
//流类 流对象;
ifstream ifile; //定义一个文件输入流对象
ofstream ofile; //定义一个文件输出流对象
fstream iofile; //定义一个文件输出/输入流对象
```

11.2.3.2.打开文件

定义了文件流对象后，就可以利用其成员函数 open ( ) 打开需要操作的文件，该成员函数的函数原型为：

```
void open ( const unsigned char *filename,int mode,int access=filebuf:openprot ) ;
```

其中：filename 是一个字符型指针，指定了要打开的文件名；mode 指定了文件的打开方式，其值如下表所示；access 指定了文件的系统属性，取默认即可：

mode ：在 ios 类中定义的文件打开方式

文件打开方式	值	含 义
ios::in	0x01	以输入（读）方式打开文件，若文件不存在则报错。
ios::out	0x02	以输出（写）方式打开文件， 若文件不存则创建。
ios::app	0x08	打开一个文件使新的内容始终添加在文件的末尾，若文件不存在，则报错。
ios::trunc	0x10	若文件存在，则清除文件所有内容；若文件不存在，则创建新文件。
ios::binary	0x80	以二进制方式打开文件，缺省时以文本方式打开文件。
ios::nocreate	0x20	打开一个已有文件，若该文件不存在，则打开失败。
ios::noreplace	0x40	若打开的文件已经存在，则打开失败。

说明：

1) 在实际使用过程中，可以根据需要将以上打开文件的方式用“|”组合起来。如：

ios::in|ios::out	表示以读/写方式打开文件
ios::in|ios:: binary	表示以二进制读方式打开文件
ios::out|ios:: binary	表示以二进制写方式打开文件
ios::in|ios::out|ios::binary	表示以二进制读/写方式打开文件

2) 如果未指明以二进制方式打开文件，则默认是以文本方式打开文件。

3) 对于 ifstream 流， mode 参数的默认值为 ios::in，
对于 ofstream 流，mode 的默 认值为 ios::out|ios::trunc,
对于 fstream 流， mode 的默认值为 ios::int|ios::out|ios::app

4) 也可以通过，构造函数打开文件。

5) 出错处理是通过，对类对象进行判断的。若文件打开成功，返回 1，否则返回 0

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream  ifs("xxx.txt",ios::in);
    if(!ifs)
        cout<<"open error1"<<endl;

    char buf[100];
    if(ifs>>buf)
        cout<<buf<<endl;


    ofstream ofs("yyy.txt",ios::out|ios::app);

    if(!ofs)
        cout<<"open error2"<<endl;

    ofs<<"abcefldkj"<<endl;


fstream  fs("zzz.txt",ios::in|ios::out|ios::app);
//app 有创建文件的功能 trunc 也有，但是清空
    if(!fs)
        cout<<"open error3"<<endl;

    fs<<"abcdefg";
    char buf[1024];

    fs.seekg(0,ios::beg);

    fs>>buf;
```

```
    if(fs)
        cout<<buf<<endl;




    return 0;
}
```

11.2.3.3.文件的关闭

在文件操作结束（即读、写完毕）时应及时调用成员函数 close()来关闭文件。该函数比较简单，没有参数和返回值。

11.2.4.流文件状态与判断

系统为了标识当前文件操作的状态，提供了标识位和检查标识位的函数。

11.2.4.1.标识位

ios_base.h

```
/// Indicates a loss of integrity in an input or output sequence (such
/// as an irrecoverable read error from a file).
    static const iostate badbit =_S_badbit;

/// Indicates that an input operation reached the end of an input sequence.
    static const iostate eofbit =_S_eofbit;

/// Indicates that an input operation failed to read the expected
/// characters, or that an output operation failed to generate the
/// desired characters.
    static const iostate failbit =_S_failbit;

/// Indicates all is well.
    static const iostate goodbit =_S_goodbit;
```

11.2.4.2.函数

●eof()

如果读文件到达文件末尾，返回 true。

●bad()

如果在读写过程中出错，返回 true 。例如：当我们要对一个不是打开为写状态的文件进行写入时，或者我们要写入的设备没有剩余空间的时候。

●fail()

除了与 bad() 同样的情况下会返回 true 以外，加上格式错误时也返回 true ，例如当想要读入一个整数，而获得了一个字母的时候。或是遇到 eof。

●good()

这是最通用的：如果调用以上任何一个函数返回 true 的话，此函数返回 false 。

●clear()

标识位一旦被置位 ,这些标志将不会被改变 ,要想重置以上成员函数所检查的状态标志，你可以使用成员函数 clear() ,没有参数。比如:通过函数移动文件指针 ,并不会使 eofbit 自动重置。

```
#include <iostream>
using namespace std;
int main()
{
    int integerVal;

    cout << "Before a bad input operation:"
         << "\n cin.eof(): " <<cin.eof()
         << "\n cin.fail(): " <<cin.fail()
         << "\n cin.bad(): " <<cin.bad()
         << "\n cin.good(): " <<cin.good()<<endl;

    cin>>integerVal;  // control + D/Z

    cout << "After a bad input operation:"
         << "\n cin.eof(): " <<cin.eof()
         << "\n cin.fail(): " <<cin.fail()
         << "\n cin.bad(): " <<cin.bad()
         << "\n cin.good(): " <<cin.good()<<endl;

    cin.clear();
    cout<< "\n cin.eof(): " <<cin.eof()
        << "\n cin.fail(): " <<cin.fail()
        << "\n cin.bad(): " <<cin.bad()
        << "\n cin.good(): " <<cin.good()<<endl;
}
```

11.2.5.(cin)和(!cin)的原理分析

在判断文件打开成功与否或是连续从流中读取数据时，就要用到对流对像的操作，比如 if(!cin) 或是 whie(cin) 等等。

代码 while(cin>>val)，我们都知道 cin 是一个流对象，而>>运算符返回左边的流对象，也就是说 cin>>val 返回 cin，于是 while(cin>>val)就变成了 while(cin)，问题就变成了一个流对象在判断语句中的合法性。

不管是 while(cin)还是 if(cin)，都是合法的，为什么呢？我们自己定义一个类，然后定义该类的对象，然后使用 if 语句来判断它是不合法的。这说明，**流对象具有某种转换函数，可以将一个流对象转换成判断语句可以识别的类型。**

打开 iostream.h 文件，找到 cin 的定义，发现是来自于 istream.h，其中的模板类 basic_istream 继承自 basic_ios，打开 basic_ios 的定义,发现它有两个重载函数。**operator void *() const** 和 **bool operator!() const**。这两个函数使得流对象可作为判断语句的内容。

```
/**
*  @brief  The quick-and-easy status check.
```

```
*
*  This allows you to write constructs such as
*  <code>if (!a_stream) ...</code> and <code>while(a_stream) ...</code>
*/
operator void*() const  //转化函数
{
    return this->fail() ? 0 : const_cast<basic_ios*>(this);
}

bool operator!() const
{
    return this->fail();
}
```

结论：

operator void *() const; 函数在 while(cin)或是 if(cin)时被调用，将流对象转换成 void *类型。
bool operator!() const; 函数在 while(!cin)或是 if(!cin)时被调用，将流对象转换成 bool 类型。

需要指出的是，上述两个类型转换都是**隐式**的。

因此，可以简单的理解调用过程为：

```
while(cin)  =====> while(!cin.fail())    //while the stream is OK
while(!cin) =====> while(cin.fail())     //while the stream is NOT OK
```

简单测试：

```
#include<iostream>
using namespace std;
class A
{
public:
    A(){}
    ~A(){}
    operator void* ()const
    {
        cout<<"cast to void*; ";
        return (void *)this;
    }
     bool operator!() const
    {
        cout<<"cast to bool; ";
        return true;
    }
};

int main()
{
    A a;
    if(a) cout<<"first"<<endl;
    if(!a) cout<<"second"<<endl;
    return 0;
}
```

11.2.6.文件的读写操作

在打开文件后就可以对文件进行读写操作了。

从一个文件中读出数据，可以使用文件流类的 get、getline、read 成员函数以及运算符“>>”；

而向一个文件写入数据，可以使用其 put、write 函数以及插入符“<<”，如下表所示：

11.2.6.1.流状态的查询和控制

We might manage an input operation as follows:

可以如下管理输入操作

```
int ival;
// read cin and test only for EOF;
//loop is executed even if there are other IO failures
while (cin >> ival, !cin.eof()) {
    if (cin.bad())        // input stream is corrupted; bail out
        throw runtime_error("IO stream corrupted");
    if (cin.fail()) {                      // bad input
        cerr<< "bad data, try again";       // warn the user
        cin.clear(istream::failbit);        // reset the stream
        continue;                          // get next input
    }
    // ok to process ival
}
```

This loop reads cin until end-of-file or an unrecoverable read error occurs. The condition uses a comma operator . Recall that the comma operator executes by evaluating each operand and returns its rightmost operand as its result. The condition, therefore, reads cin and ignores its result. The result of the condition is the result of !cin.eof(). If cin hit end-of-file, the condition is false and we fall out of the loop. If cin did not hit end-of-file, we enter the loop, regardless of any other error the read might have encountered.

这个循环不断读入 cin，直到到达文件结束符或者发生不可恢复的读取错误为止。循环条件使用了逗号操作符。回顾逗号操作符的求解过程：首先计算它的每一个操作数，然后返回最右边操作数作为整个操作的结果。因此，循环条件只读入 cin 而忽略了其结果。该条件的结果是 !cin.eof() 的值。如果 cin 到达文件结束符，条件则为假，退出循环。如果 cin 没有到达文件结束符，则不管在读取时是否发生了其他可能遇到的错误，都进入循环。

Inside the loop, we first check whether the stream is corrupted. If so, we exit by throwing an exception . If the input was invalid, we print a warning, and clear the failbit state. In this case, we execute a continue to return to the start of the while to read another value into ival. If there were no errors, the rest of the loop can safely

use ival.

在循环中，首先检查流是否已破坏。如果是的放，抛出异常并退出循环。如果输入无效，则输出警告并清除 failbit 状态。在本例中，执行 continue 语句,回到 while 的开头，读入另一个值 ival。如果没有出现任何错误，那么循环体中余下的部分则可以很安全地使用 ival。

11.2.6.2.读写文件本文件

读出：

- operator>>
- int get();
- istream& get(int);
- istream & get(char*,int n, char deli)
- istream& getline(char * ,int n);

写入:

- operator<<
- osream put(int)

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream ifs("src.txt",ios::in);
    if(!ifs)
    {
        cout<<"open error"<<endl;
        return -1;
    }

    fstream ofs("dest.txt",ios::out|ios::trunc);
    if(!ofs)
    {
        cout<<"open error"<<endl;
        return -1;
    }


//    int data;
//    while(ifs>>data,!ifs.eof())  // 只能以 空格 table 回车 作为标志
//    {
//        cout<<"x"<<endl;
//        ofs<<data<<" ";
//    }
```

```
//    char ch;
//    while(ifs.get(ch),!ifs.eof())
//    {
//        ofs.put(ch);
//    }
//    ifs.close();
//    ofs.close();

    char buf[1024];
    while(ifs.get(buf,1024,'\n'))
    {
        while(ifs.peek() == '\n')
            ifs.ignore();
        ofs<<buf<<endl;
    }
    ifs.close();
    ofs.close();



    return 0;
}
```

11.2.6.3.读写二进制文件

```
ostream & write(const char * buffer,int len);
istream & read(char * buff, int len);
```

```
#include <iostream>
#include <fstream>
using namespace std;

struct Student
{
    char name[100];
    int num;
    int age;
    char sex;
};

int main()
{
//    Student s[3] = {
//        {"li",1001,18,'f'},
//        {"Fun",1002,19,'m'},
//        {"Wang",1004,17,'f'}
//    };
```

```
//    ofstream ofs("student.data",ios::out|ios::trunc|ios::binary);
//    if(!ofs){
//        cout<<"open error"<<endl;
//    }

//    for(int i=0; i<3; i++)
//    {
//        ofs.write((char*)&s[i],sizeof(s[i]));
//    }

//    ofs.close();

    Student s;

    ifstream  ifs("student.data",ios::int|ios::binary);

    if(!ifs)
        cout<<"open error"<<endl;

 ifs.seekg(sizeof(stu),ios::beg);



    while(ifs.read((char*)&s,sizeof(Student)),!ifs.eof())
    {
        cout<<"Name "<<s.name<<endl;
        cout<<"Num  "<<s.num<<endl;
        cout<<"Age  "<<s.age<<endl;
        cout<<"Sex  "<<s.sex<<endl;
        cout<<"---------------"<<endl;
    }
    ifs.close();

    return 0;

}
```

11.2.7.随机读写函数

与文件指针相关的函数如下：

成员函数	作用

tellg();
seekg(绝对位置);
seekg(相对位置,参照位置);
seekp(绝对位置);
seekp(相对位置,参照位置);
tellp();
	返回当前指针位置　//输入流操作
	绝对移动，
	相对操作
	绝对移动，　　//输出流操作
	相对操作
	返回当前指针位置

意思用于输入的函数。p 代表 put 的意思，用于输出函数。如果是既可输入又可输出的文件，则任意使用。

参照位置

成员	意义
ios::beg = 0 ios::cur = 1 ios::end = 2	相对于文件头 相对于当前位置 相对于文件尾

代码示例：

infile.seekg(100);　//输入文件中的指针向前移到 100 个字节的位置
infile.seekg(-50,ios::cur); //输入文件中的指针从当前位置后移 50 个字节
outfile.seekp(-75,iso::end); //输出文件中指针从文件尾后移 50 个字节

11.2.8.综合练习：

有 5 个学生的数据，要求
1，把它们存到磁盘文件中；
2，将磁盘文件中的第 1 3 5 个学生数据读入程序，并显示出来；
3，将第 3 个学生的数据修改后存回磁盘文件中的原有位置;
4，从磁盘文件读入修改后的 5 个学生数据并显示出来。

提示: 要实现以上要求，需要解决 3 个问题
1,由于同一磁盘文件在程序中需要频繁地进行输入和输出，因此可将文件的工　作方式指定为输入输出文件，ios::in|ios::out|ios::binary。
2,正确计算好每次访问时指针的定位，即正确使 seekg 或 seekp 函数。
3 正确进行文件中数据的重写（更新)。

代码:

```
#include <fstream>
using namespace std;
```

```
struct student
{int num;
    char name[20];
    float score;
};
int main()
{
    int i;
    student stud[5]={1001,"Li",85,
                     1002,"Fun",97.5,
                     1004,"Wang",54,
                     1006,"Tan",76.5,
                     1010,"ling",96};
    fstream iofile("stud.dat",ios::in|ios::out|ios::binary);
    if(!iofile)
    {
        cerr<<"open error!"<<endl;
        abort();
    }
    for(i=0;i<5;i++)
        iofile.write((char *)&stud[i],sizeof(stud[i]));
    student stud1[5];
    for(i=0;i<5;i=i+2)
    {
        iofile.seekg(i*sizeof(stud[i]),ios::beg);
        iofile.read((char *)&stud1[i/2],sizeof(stud1[i]));
        cout<<stud1[i/2].num<<" "<<stud1[i/2].name<<" "<<stud1[i/2].score<<endl;
    }
    cout<<endl;
    stud[2].num=1012;
    strcpy(stud[2].name,"Wu");
    stud[2].score=60;
    iofile.seekp(2*sizeof(stud[0]),ios::beg);
    iofile.write((char *)&stud[2],sizeof(stud[2]));
    iofile.seekg(0,ios::beg);
    for(i=0;i<5;i++)
    {
        iofile.read((char *)&stud[i],sizeof(stud[i]));
        cout<<stud[i].num<<" "<<stud[i].name<<" "<<stud[i].score<<endl;
    }
    iofile.close();
    return 0;
}
```

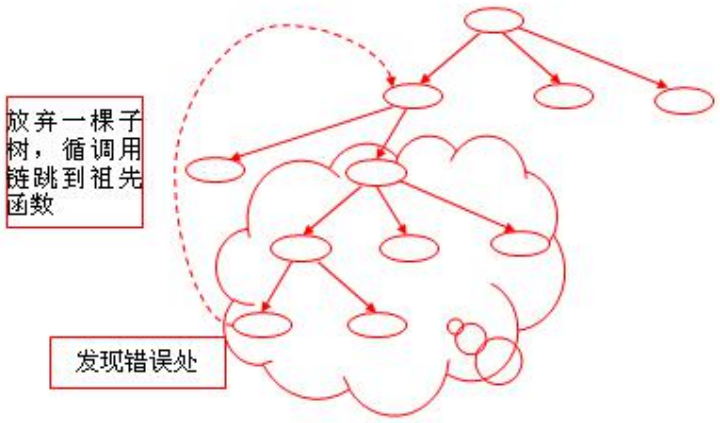
12.异常(Exception)

c 语言中错误的处理 ,通常采用返回值的方式或是置位全局变量的方式。这就存在两个问题。如果返回值正是我们需要的数据，且返回数据同出错数据容错差不高。全局变量，在多线程中易引发竞争。而且，当错误发生时，上级函数要出错处理，层层上报，造成过多的出错处理代码，且传递的效率低下。为此 c++提供了异常。

12.1.引入异常的意义

1）C++的异常处理机制使得**异常的引发和异常的处理不必在同一个函数**中，这样底层的函数可以着重解决具体问题，而不必过多的考虑异常的处理。上层调用者可以再适当的位置设计对不同类型异常的处理。

2）异常是专门针对抽象编程中的一系列错误处理的，C++中不能借助函数机制，因为栈结构的本质是先进后出，依次访问，**无法进行跳跃**，但错误处理的特征却是遇到错误信息就想要转到若干级之上进行重新尝试，如图



3）异常**超脱于函数机制**，决定了其对函数的跨越式回跳。异常跨越函数。

12.2.引例

12.2.1.求三角形的面积

```
#include <iostream>
#include <cmath>

using namespace std;

double triangle(double x, double y, double z)
{
    double area;
    double s = (x+y+z)/2;

        area = sqrt(s*(s-x)*(s-y)*(s-z));


    return area;
}
```

```
int main()
{
    int a, b,c;
    cin>>a>>b>>c;

    while(a>0&&b>0&&c>0)
    {
        cout<<triangle(a,b,c)<<endl;
        cin>>a>>b>>c;
    }
    return 0;
}
```

12.2.2.引入异常

```
#include <iostream>
#include <cmath>

using namespace std;

double triangle(double x, double y, double z)
{
    double area;
    double s = (x+y+z)/2;

    if(x+y>z && y+z>x && x+z >y)
        area = sqrt(s*(s-x)*(s-y)*(s-z));
    else
        throw 4;  //throw 数据为任意类型

    return area;
}

int main()
{
    int a, b,c;
    cin>>a>>b>>c;

    try{
        while(a>0&&b>0&&c>0)
        {
            cout<<triangle(a,b,c)<<endl;
            cin>>a>>b>>c;
        }

    }catch(int)
    {
        cout<<a<<","<<b<<","<<c<<" "<<"构不成三角形"<<endl;
    }
    cout<<"end"<<endl;

    return 0;
}
```

分析：

1，把可能发生异常的语句放在 try 语句声当中。try 原有语句的执行流程。

2，若未发生异常，catch 子语句并不起作用。程序会流转到 catch 子句的后面执行。

3，若 try 块中发生异常，则通过 throw 抛出异常。throw 抛出异常后，程序立即离开本函数，转到上一级函数。所以 triangle 函数中的 return 语句不会执行。

4，throw 抛出数据，类型不限。既可以是基本数据类型，也可以是构造数据类型。

5，程序流转到 main 函数以后，try 语句块中 抛出进行匹配。匹配成功，执行 catch 语句，catch 语句执行完毕后。继续执行后面的语句。如无匹配，系统调用 terminate 终止程序。

12.2.3.语法格式

```
try
{
被检查的语句
}
catch(异常信息类型 [变量名])
{
进行异常处理的语句
}
```

语法分析：

1，被检语句必须放在 try 块中，否则不起作用。

2，try catch 中花括号不可省。

3，一个 try-catch 结构中，只能有一个 try 块，catch 块却可以有多个。以便与不同的类型信息匹配。

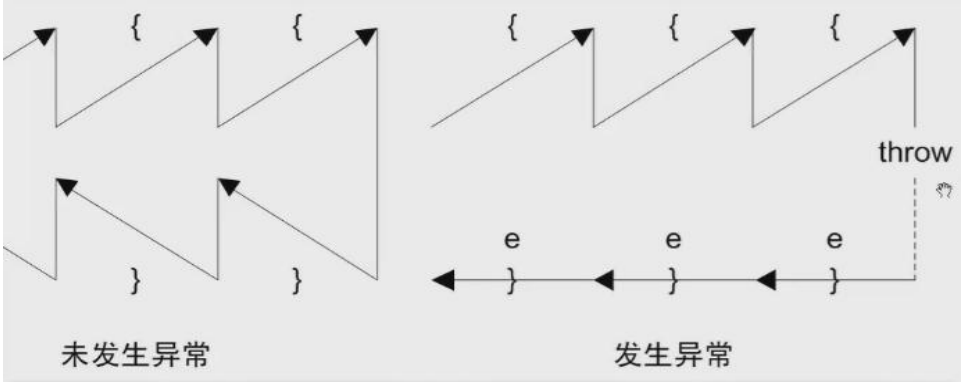
```
try{}
catch(double){}
catch(int){}
catch(char){}
catch(float){}
```

4,throw 抛出的类型，既可以是系统预定义的标准类型也可以是自定义类型。从抛出到 catch 是一次复制拷贝的过程。如果有自定义类型，要考虑自定义类型的拷贝问题。

5.如果 catch 语句没有匹配异常类型信息，就可以用(...)表示可以捕获任何异常类型的信息。

```
catch(...)
{
    cout<<"catch a unknow exception"<<endl;
}
```

6.try-catch 结构可以与 throw 在同一个函中数中，也可以不在同一个函数中，throw 抛出异常后，先在本函数内寻找与之匹配的 catch 块，找不到与之匹配的就转到上一层，**如果上一层也没有，则转到更上一层的 catch 块**。如果最终找不到与之匹配的 catch 块，系统则会调有生个系统函数，terminate,使程序终止。




```
#include <iostream>

using namespace std;


void g()
{
    double a;
    try
    {
        throw a;
    }catch(double)
    {
        cout<<"catch g()"<<endl;  //throw
    }
    cout<<"end g()"<<endl;
    return ;
}


void h()
{
    try{
        g();
    }catch(int)
    {
        cout<<"catch h()"<<endl;
    }
    cout<<"end h()"<<endl;

}

void f()
{
```

```
    try{
        h();
    }catch(char)
    {
        cout<<"catch f()"<<endl;
    }
    cout<<"end f()"<<endl;


}


int main()
{
    try{
        f();
    }catch(double)
    {
        cout<<"catch main"<<endl;
    }
    cout<<"end main"<<endl;
    return 0;
}
```

12.3.抛出类型声明

1) 为了加强程序的可读性，可以在函数声明中列出可能抛出的所有异常类型。

例如：

void func() throw (A, B, C , D); //这个函数 func（ ）能够且只能抛出类型 A B C D 及其子类型的异常。

2) 如果在函数声明中没有包含异常接口声明，则函数可以抛掷任何类型的异常，例如：

void func();

3) 一个不抛掷任何类型异常的函数可以声明为：

void func()　throw();

4) 如果一个函数抛出了它的异常接口声明所不允许抛出的异常 ,unexpected 函数会被 调用，该函数默认行为调用 terminate 函数中止程序。

12.4.栈自旋

异常被抛出后，从进入 try 块起，到异常被抛掷前，这期间在栈上的构造的所有对象，都会被自动析构。析构的顺序与构造的顺序相反。这一过程称为栈的解旋(unwinding)。

而堆上的空间，则会泄漏。

```
#include <iostream>

using namespace std;

class A
```

```
{
public:
    A()
    {
        cout<<"A constructor"<<endl;
    }
    ~A()
    {
        cout<<"~A destructor"<<endl;
    }
};

int divide(int x, int y)
{
    A a;
    if(y == 0)
        throw('a');
    return x/y;
}

int myDivide(int x, int y)
{
    A b;
    divide(x,y) ;
}

int main()
{
    try{
        myDivide(4,0);
    }catch(int x){
       cout<<"x"<<endl;
       cout<<x<<endl;
    }catch(double y){
        cout<<"y"<<endl;
        cout<<y<<endl;
    }catch(...){
        cout<<"no x, no y"<<endl;
    }

    return 0;
}
```

12.4.1.返回类对象(引用，实例)

```
#include <iostream>

using namespace std;

class A
{
public:
    A()
    {
        cout<<"A constructor"<<endl;
```

```
    }
    A(const A & )
    {
        cout<<"A copy constructor"<<endl;
    }

    ~A()
    {
        cout<<"~A destructor"<<endl;
    }
};

int divide(int x, int y)
{
    A a;
    if(y == 0)
        throw(a);
    return x/y;
}

int myDivide(int x, int y)
{

    divide(x,y) ;
}

int main()
{
    try{
        myDivide(4,0);
    }catch(int x){
       cout<<"x"<<endl;
       cout<<x<<endl;
    }catch(double y){
        cout<<"y"<<endl;
        cout<<y<<endl;
    }catch(const A &a){
        cout<<"no x, no y"<<endl;
    }

    return 0;
}
```

13.STL

14.C11

15.Boost

16.附录

16.1.运算符优先级

Precedence	Operator	Description	Example	Associativity

1	()
	[]
	->
	.
	::
	++
	--
		Grouping operator
		Array access
		Member access from a pointer
		Member access from an object
		Scoping operator
		Post-increment
		Post-decrement
			(a + b) / 4;
			array[4] = 2;
			ptr->age = 34;
			obj.age = 34;
			Class::age = 2;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
				left to right

2	!
	~
	++
	--
	-
	+
	*
	&
	(type)
	sizeof
		Logical negation
		Bitwise complement
		Pre-increment
		Pre-decrement
		Unary minus
		Unary plus
		Dereference
		Address of
		Cast to a given type
		Return size in bytes
			if(!done) ...
			flags = ~flags;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			int i = -1;
			int i = +1;
			data = *ptr;
			address = &obj;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				right to left

3	->* .*	Member pointer selector Member pointer selector	ptr->*var = 24; obj.*var = 24;	left to right

4	*
	/
	%
		Multiplication
		Division
		Modulus
			int i = 2 * 4;
			float f = 10 / 3;
			int rem = 4 % 3;
				left to right

5	+ -	Addition Subtraction	int i = 2 + 3; int i = 5 - 1;	left to right
6	<< >>	Bitwise shift left Bitwise shift right	int flags = 33 << 1; int flags = 33 >> 1;	left to right

7	<
	<=
	>
	>=
		Comparison less-than
		Comparison less-than-or-equal-to
		Comparison greater-than
		Comparison geater-than-or-equal-to
			if(i < 42) ...
			if(i <= 42) ...
			if(i > 42) ...
			if(i >= 42) ...
				left to right

8	== !=	Comparison equal-to Comparison not-equal-to	if(i == 42) ... if(i != 42) ...	left to right
9	&	Bitwise AND	flags = flags & 42;	left to right
10	^	Bitwise exclusive OR	flags = flags ^ 42;	left to right
11	|	Bitwise inclusive (normal) OR	flags = flags | 42;	left to right
12	&&	Logical AND	if(conditionA && conditionB) ...	left to right
13	||	Logical OR	if(conditionA || conditionB) ...	left to right
14	? :	Ternary conditional (if-then-else)	int i = (a > b) ? a : b;	right to left

15	=
	+=
	-=
	*=
	/=
	%=
	&=
	^=
	|=
	<<=
	>>=
		Assignment operator
		Increment and assign
		Decrement and assign
		Multiply and assign
		Divide and assign
		Modulo and assign
		Bitwise AND and assign
		Bitwise exclusive OR and assign
		Bitwise inclusive (normal) OR and assign
		Bitwise shift left and assign
		Bitwise shift right and assign
			int a = b;
			a += 3;
			b -= 4;
			a *= 5;
			a /= 2;
			a %= 3;
			flags &= new_flags;
			flags ^= new_flags;
			flags |= new_flags;
			flags <<= 2;
			flags >>= 2;
				right to left

16	,	Sequential evaluation operator	for(i = 0, j = 0; i < 10; i++, j++) ...	left to right

16.2.ASCII 码