第一章　认识C＋＋的对象

§1.1 初识C＋＋的函数和对象

1、混合型函数

C++以 .cpp 为文件扩展名，有且只有一个名为main的主函数，因保留了这个面向过程的主函数，所以被称为混合型语言。

2、注释方式

①从“/*”开始，到“*/”结束，如：/* …… */

②从“//”开始到本行结束，如：　　//……

3、输入输出对象

①提取操作：用提取操作符“>>”从cin输入流中提取字符，如：　cin >> a.x;

②插入操作：用插入操作符“<<”向cout输出流中插入字符，如：cout <<“we”; cout << endl;

③使用标准输入（键盘输入）cin及标准输出（屏幕输出）cout前，要在主函数前使用＃include <iostream>将C＋＋标准输入输出库头文件iostream将其包括。

④换行操作：用语句cout<<endl；或cout<<“\n”；实现，其中endl可以插在流的中间。

如：cout<<a.x<<endl<<a.y<<endl;

4、使用命名空间

C＋＋相比C而言，可以省略“ .h”标识头文件，但必须使用语句using namespace std；来让命名空间中的对象名称曝光。因此一般的程序都要具有下面的两条语句：

```
＃include <iostream>          //包含头文件
using namespace std;          //使用命名空间
```

注意C＋＋库中代替C库中头文件的正确名称,例如下面两个语句效果一样：

①＃include <math.h>

②＃include <cmath>

```
using namespace std;
```

5、对象的定义和初始化

定义对象包括为它命名并赋予它数据类型，一般即使初值只用来表示该对象尚未具有真正意义的值，也应将每个对象初始化。

C＋＋中利用构造函数语法实现初始化，如：

```
Int z(0);          //等同于 int z＝0;
```

6、函数原型及其返回值

①C＋＋使用变量和函数的基本规则都是：先声明，后使用。变量有时也可边声明边使用，但必须声明，否则出错。

比如对函数的调用，要在主函数之前先对调用的函数进行原型声明，如：int result （int，int)； 它向编译系统声明，后面有一个 result 函数，该函数有两个整型类型的参数，函数返回整型值。

函数声明时，除了默认参数（需给出默认参数的默认值）和内联函数（需给出函数体及其内语句）外，不需给出参数的变量名称，如果给出，效果也一样，如：int result （int a，int b)； 和上面的声明效果一样。

②除构造函数与析构函数外，函数都需要有类型声明。

如 int main（） ，指出 main 是整数类型，返回值由 return 后面的表达式决定，且表达式的值必须与声明函数的类型一致。

如果函数确实不需要返回值，还可用 void 标识，一旦使用 void 标识，函数体内就不再需要使用 return 语句，否则会编译出错，但可使用 return；语句。

③C＋＋函数有库函数（标准函数，引用时函数名外加<>）和自定义函数（引用时函数名外加“ ”）两类。

7、const（常量）修饰符及预处理程序

①const 修饰符：用于定义符号常量。

C 中一般使用宏定义“＃define”定义常量，而 C＋＋中除此外，建议使用 const 代替宏定义，用关键字 const 修饰的标识符称为符号常量。

因 const 是放在语句定义之前的，因此可以进行类型判别，这比用宏定义更安全一些。如下面两个语句是等同的，但是后者可以比前者避免一些很难发现的

错误。

＃difine BOFSIZE 100

const int BUFSIZE 100;

常量定义也可使用构造函数的初始化方法，如：

const int k（2）;　　//等同于 const int k＝2;

因被 const 修饰的变量的值在程序中不能被改变，所以在声明符号常量时，必须对符号常量进行初始化，除非这个变量是用 extern 修饰的外部变量，如：

const int d; ×　　const int d=2; √　　extern const int d; √

const 的用处不仅是在常量表达式中代替宏定义，如果一个变量在生存期内的值不会改变，就应该用 const 来修饰这个变量，以提高程序安全性。

②预处理程序

C＋＋的预处理程序不是 C＋＋编译程序的一部分，它负责在编译程序的其他部分之前分析处理预处理语句，为与一般的 C＋＋语句区别，所有预处理语句都以位于行首的符号“＃”开始，作用是把所有出现的、被定义的名字全部替换成对应的“字符序列”。

预处理语句有三种：宏定义、文件包含（也成嵌入指令）和条件编译。

文件包含是指一个程序把另一个指定文件的内容包含进来，书写时可以使用引号也可以使用尖括号，前者引用自己定义的包含文件，如：＃include “E:\prog\myfile.h”，后者引用系统提供的包含文件，如标准输入输出是定义在标准库 iostream 中的，引用时要包括以下两条语句：

＃include <iostream>　　//包含头文件

using namespace std;　　//使用命名空间

8、程序运行结果

9、程序书写格式

C＋＋的格式和 C 一样，都很自由，一行可以写几条语句，但也要注意以下规则，增加可读性：

①括号紧跟函数名后面，但在 for 和 while 后面，应用一个空格与左括号隔开；

②数学运算符左右各留一个空格，以与表达式区别；

③在表示参数时，逗号后面留一个空格；

④在 for、do…while 和 while 语句中，合理使用缩进、一对花括号和空行；

⑤适当增加空行和程序注释以增加可读性；

⑥太长的程序分为两行或几行，并注意选取合适的分行和缩进位置。

§1.2 认识 C＋＋面向过程编译的特点

一、使用函数重载

C＋＋允许为同一个函数定义几个版本，从而使一个函数名具有多种功能，这称为函数重载。

假设有一个函数 max，分别具有以下函数原型：

```
int max（int，int）;       //2 个整型参数的函数原型
int max（int，int，int）; //3 个整型参数的函数原型
```

只要分别为不同参数的 max 编制相应的函数体，就可以实现各自的功能。

二、新的基本数据类型及其注意事项

1、void 是无类型标识符，只能声明函数的返回值类型，不能声明变量。

2、C＋＋还比 C 多了 bool（布尔）型。

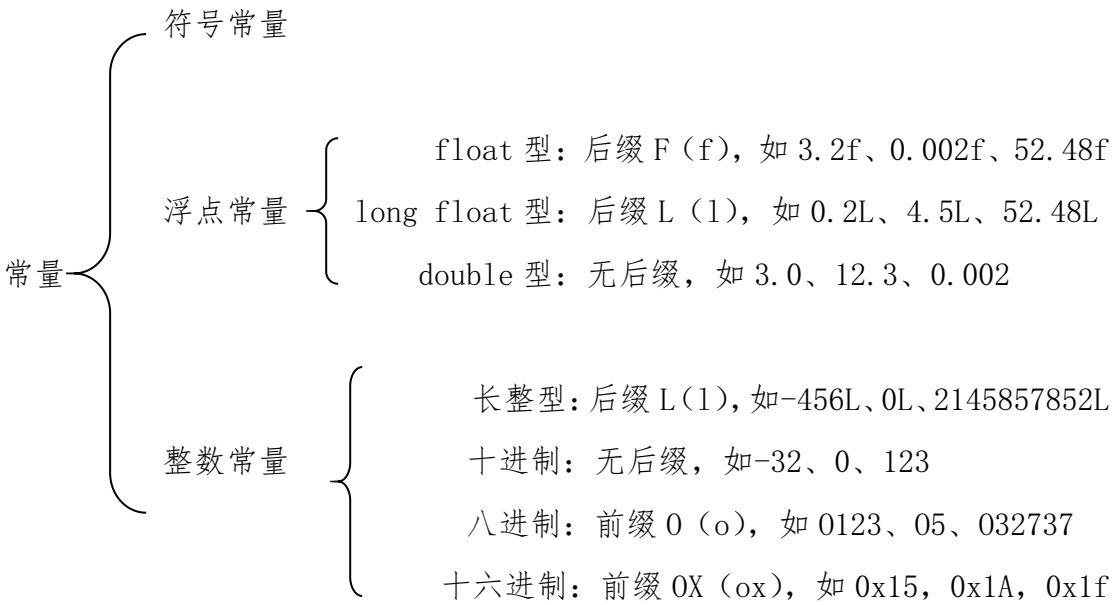
3、C＋＋只限定 int 和 short 至少要有 16 位，而 long 至少 32 位，short 不得长于 int，int 不能长于 long，VC＋＋6.0 规定 int 使用 4 字节，这与 C 使用 2 字节不同。

4、地址运算符“&”用来取对象存储的首地址，对于数组，则数组名就是数组的首地址。

如：int x＝56；定义 x，VC＋＋6.0 使用 4 个字节存储对象 56，假设存放的内存首地址用十六进制表示为 006AFDEC，则语句 cout <<&x；自动使用十六进制输出存储的首地址 006AFDEC。

5、C＋＋中的常量分三种，第一种为符号常量；第二种为整数常量，有 4 种类型，分别为十进制、长整型、八进制、十六进制，并用前缀和后缀进行分类标识；第三种为浮点常量，有三种类型，分别为 float 型、long float 型、double 型，

并用后缀进行分类识别。



16、C＋＋与 C 一样，也使用转义序列。如：'\0'表示 ASCII 码值为零的空字符（NULL），'\101'表示字符 A。

三、动态分配内存

1、在使用指针时，如果不使用对象地址初始化指针，可以自己给它分配地址。

对于只存储一个基本类型数据的指针，申请方式如下：

```
new 类型名[size]        //申请可以存储 size 个该数据类型的对象
```

不再使用时，必须使用 delete 指针名；来释放已经申请的存储空间。

如：……

```
double *p;              //声明 double 型指针
p＝new double[3]        //分配 3 个 double 型数据的存储空间
  ……
  delete p;             //释放已申请的存储空间
  ……
```

2、C 必须在可执行语句之前集中声明变量，而 C＋＋可以在使用对象时再声明或定义。

3、C＋＋为结构动态分配内存一般格式为：

```
指针名＝new 结构名;      //分配
```

delete 指针名; //释放

例如给书中例 1.1 的 Point 结构指针分配内存:

p=new Point;

当不再使用这个空间时，必须使用 delete p; 释放空间。

四、引用

1、引用简单的说，就是为现有的对象起个别名，别名的地址与引用对象的地址是一样的。

引用的声明方式为： 数据类型 & 别名=对象名; ，注意对象在引用前必须先初始化，另外声明中符号“&”的位置无关紧要，比如 int& a=x; 、int & a=x; 和 int &a=x; 等效。

例：……

```
int x=56;      //定义并初始化 x
int & a=x;     //声明 a 是 x 的引用，二者地址相同
int &r=a;      //声明 r 是 a 的引用，二者地址相同
……
r=25;          //改变 r，则 a 和 x 都同步变化
……
```

2、所谓“引用”，就是将一个新标识符和一块已经存在的存储区域相关联。因此，使用引用时没有分配新的存储区域，它本身不是新的数据类型。

可以通过修改引用来修改原对象，但是不能有空引用，在程序中必须确保引用是和一块正确的存储区域关联。

引用通常用于函数的参数表中或作为函数的返回值。前者因为使用引用作为函数参数不产生临时对象，可提高程序执行效率和安全性（§4.4.3），后者则是因为引用作为函数返回值可用于赋值运算符的左边。

3、引用实际上就是变量的别名，使用引用就如同直接使用变量一样，引用与变量名在使用的形式上完全一样，引用只是作为一种标识对象的手段。

但要注意，可以声明指向变量或引用的指针，如：int *p=&x; √ int &a=x; int *p=&a; √；也可以声明指针对指针的引用，如：int *& p2=p1; √（式中 p1、p2 是指针，* 声明 p2 是指针，&声明 p2 是 p1 的引用）;

但不能声明指针对变量的引用，如：int *&P=&x; × ；不能声明引用的引用，如：int & & r=x; ×；也不能直接声明对数组的引用。

4、引用的作用与指针有相似之处，它会对内存地址上存在的变量进行修改，但它不占用新的地址，从而节省开销。二者除使用形式不同，本质也不同：指针是低级的直接操作内存地址的机制，可由整型数强制类型转换得到，功能强大但易出错；引用则是较高级的封装了指针的特性，不直接操作内存地址，不可由强制类型转换而得，安全性较高。

5、虽然不能直接定义对数组的引用，但可以通过 typedef 来间接的建立对数组的引用。如：

```
……
typedef int array[10];   //定义 int 型数组类型 array
……
array a={12, 34, ……};      //定义 array 型数组 a
array & b=a;                //定义数组 a 的引用 b
……
```

五、对指针使用 const 限定符

1、变量的指针、指向变量的指针变量、指针变量指向的变量

变量的指针就是变量的地址，存放变量地址的变量是指针变量，为了表示指针变量和它所指向的变量之间的联系，在程序中用“*”符号表示“指向”。例如用 p 代表指针变量，来存放变量 a 所在的内存地址，则*p 代表指针变量指向的变量，也就是变量 a，且下面等式成立：

```
p=&a      *p=*&a=a      &*p=&a      (*p)++=a++
```

2、左值和右值

左值是指某个对象的表达式，必须是可变的。左值表达式在赋值语句中即可作为左操作数，也可作为右操作数，如：x=56；和 y=x; ，而右值 56 就只能作为右操作数，不能作为左操作数。

某些运算符如指针运算符“*”和取首地址运算符“&” 也可产生左值，例如 p 是一个指针类型的表达式，则“*p”是左值表达式，代表由 p 指向的对象，

且可通过“*p＝”改变这个对象的值;“&p”也是左值表达式，代表由 p 指向的对象的首地址，且可通过“&p＝”改变这个指针的指向。

3、指向常量的指针（const int *p＝&x; “*p＝”的操作不成立）

指向常量的指针是在非常量指针声明前面使用 const，如：const int *p; ，它告诉编译器，*p 是常量，不能将*p 作为左值进行操作，即限定了“*p＝”的操作，所以称为指向常量的指针。如：

```
const int y＝58;
const int *p1＝&y;   //指向常量的指针指向常量 y，y 不能作为左值
int x＝45;
const int *p＝&x;      //只能通过左值 x 间接改变*p 的值
```

上式中*p 不能作为左值，但可以通过“x＝”改变 x 的值，间接改变*p 的值，即 const 仅是限制使用*p 的方式，*p 仍然可以作为右值使用，还可以通过运算符&改变指针所指向的地址，但不能改变指针所指向的内存地址中的内容。

4、常量指针（int * const p＝&x; “p＝”的操作不成立）

把 const 限定符放在*号的右边，就可使指针本身成为一个 const，即常量指针。如：

```
int x＝5;
int * const p＝&x;
```

式中的指针本身是常量，编译器要求给它一个初始化值，这个值在指针的整个生存期中都不会改变，编译器把 p 看作常量地址，所以不能作为左值，即“p＝”不成立，也就是说不能改变指针 p 所指向的地址。但这个内存地址里的内容可以使用间接引用运算符*改变其值，例如语句 *p＝56; 将上面的 x 的值改变为 56。

4、指向常量的常量指针

也可以声明指针本身和所指向的对象都不能改变的“指向常量的常量指针”，这时必须要初始化指针。如：

```
int x＝2;
const int * const p＝&x;
```

语句告诉编译器，*p 和 p 都是常量，都不能作为左值，即“*p＝”和“p＝”

两操作均不成立，这种指针限制“&”和“*”运算符，在实际中很少用。

六、泛型算法应用于普通数组

1、数组中元素及位置的关系

如 int a[ ]＝{5，6，7，8}；

则数组中各元素分别为：a[0]＝5，a[1]＝6，a[2]＝7，a[3]＝8。a 为数组的起始地址，各元素的位置分别是：a＋1 位置为 5，a＋2 位置为 6，a＋3 位置为 7，a＋4 位置为 8。对数组按元素位置进行操作时，不包括起始位置，如：语句 sort （a＋1，a＋4）；，只对从 a＋1 位置（不含 a＋1 位置的元素）起到 a＋4 位置（含 a＋4 位置的元素）为止的各元素进行操作，即 a＋2，a＋3，a＋4 三个位置上的三个元素，而不是 a＋1～a＋4 四个位置上的所有 4 个元素。

注意式中的 a＋1 并不是地址 a 加上一个字节后的地址，而是 a＋1×d 得到的地址，其中 d 是元素类型占用的字节数，比如 C＋＋中整型数占用 4 个字节，则 a＋1 位置上元素的地址就是地址 a 加上 4 个字节后得到的地址。

2、数组不能作为整体输出，C＋＋引入 STL 库提供的泛型算法，大大简化数组操作。所谓泛型算法，就是提供的操作与元素的类型无关。

3、对数组内容进行升幂、输出、反转和复制等操作需要包含头文件<algorithm>;对数组内容进行降幂和检索等操作需要包含头文件<functional>。

4、假设一维数组 a 和 b 的长度均为 Len，数据类型为 Type，则对数组内容的相关操作和语句如下：

①数据内容反转：

reverse（a，a＋Len）；　　//数组元素反转排列

②复制数组内容：

copy（a，a＋Len，b）；　//将数组 a 的内容原样复制到数组 b

reverse_copy（a，a＋Len，b）；　//逆向复制数组 a 中内容到数组 b

③数组升幂排序：

sort （a，a＋Len）；　//默认排序方式是升幂

④数组降幂排序：

sort （b，b＋Len，greater<Type>（））；　//数组降幂排序

⑤检索查找数组内容：

find（a，a＋Len，value）；　//查找数组 a 中是否存在值为 value 的元素

find 函数返回的是位置指针，一般使用判别语句输出查找的内容，如：

Type *x＝find（a，a＋Len，value）；　//x 是类型为 type 的指针

if（x＝＝a＋Len）　cout<<“没有值为 value 的数组元素”;

else　　cout<<“有值为 value 的数组元素”;

⑥输出数组的内容

copy（a，a＋Len，ostream_iterator<Type>（cout，“字符串”））;

可将 ostream_iterator 简单理解为输出流操作符，<Type>表示数组元素的数据类型，本语句将数组内容按正向送往屏幕，输出方式是将每个元素与“字符串”的内容组合在一起连续输出。如果使用空格“ ”或换行符“\n”，可以按格式输出。也可将数组内容按逆向方式送往屏幕，语句为：

reverse_copy（a，a＋Len，ostream_iterator<Type>（cout，“字符串”））;

关键字
- 反转：reverse
- 复制：copy，reverse_copy（逆向复制）
- 排序：sort（默认升幂，尾部加 greater<Type>（） 为降幂）
- 检索：find
- 输出：copy(尾部必须加 ostream_iterator<Yype>(cout,“字符串”))

七、数据的简单输入输出格式

1、C＋＋提供了两种格式控制方式，一种是使用 iso_base 类提供的接口，另一种是使用一种称为操控符的特殊函数,操控符的特点是可直接包含在输入和输出表达式中，因此更为方便，不带形式参数的操控符定义在头文件<iostream>中，带形式参数的操控符定义在头文件<iomanip>中。

在使用操控符时，一是要正确包含它们，二是只有与符号“<<”或“>>”连接时才起作用，三是无参数的操控符函数不能带有“()”号。

2、常用操控符及其作用

格式	含义	作用
dec	设置转换基数为十进制	输入/输出
oct	设置转换基数为八进制	输入/输出

hex	设置转换技术为十六进制	输入/输出
endl	输出一个换行符并刷新流	输出
Setprecision（n）	设置浮点数输出精度 n	输出
Setw（n）	设置输出数据字段宽度	输出
Setfill（'字符'）	设置 ch 为填充字符	输出
Setiosflags（flag）	设置 flag 指定的标志位	输出
resetiosflags（flag）	清除 flag 指定的标志位	输出

上表中操控符使用时，后四个操控符必须包含头文件<iomanip>，其中后两个操控符的参数 flag 是引用 C＋＋的类 ios_base 里定义的枚举常量，要使用限定符“::”，下面的表中是几个常用的 ios_base 定义的枚举常量，另外 flag 可由多个常量“或”起来使用，如：setiosflags（ios_base::showpoint ｜ ios_base::fixed)。

参数 flag 常引用的枚举常量及其含义

常量名	含义
ios_base::left	输出数据按输出域左边对齐输出
ios_base::right	输出数据按输出域右边对齐输出
ios_base::showpos	在正数前添加一个“＋”号
ios_base::showpoint	浮点输出时必须带有一个小数点
ios_base::scientific	使用科学计数法表示浮点数
ios_base::fixed	使用定点形式表示浮点数

3、操控符使用实例

①使用 setw 设置输出宽度

```
#include <iostream>
#include <iomanip>
using namespace std;
void main(){
```

```
cout<<setfill('*')
    <<setw(0)<<15<<endl
    <<setw(1)<<15<<endl
    <<setw(2)<<15<<endl
    <<setw(3)<<15<<endl
    <<setw(4)<<15<<endl;
 cout<<setw(16)<<setfill('*')<<" "<<endl;

 cout<<setiosflags(ios_base::right)     //设置标志位
     <<setw(5)<<1
     <<setw(5)<<2
     <<setw(5)<<3;
cout<<resetiosflags(ios_base::right);     //清除标志位
cout<<setiosflags(ios_base::left)
    <<setw(5)<<1
    <<setw(5)<<2
    <<setw(5)<<3<<endl; }
```

程序输出结果为：

```
15
15
15
*15
**15
***************

****1****2****3
1****2****3****
```

如上所示，setw（n）只对在后面紧接着的那个元素有效；当域宽 n 比后面

要显示元素的位数少时，则不起作用，即不影响显示；使用填充字符时，n比后面要显示元素的位数大1时，才发生填充作用；要显示15个“*”号，必须取n＝16，同时setfill后面使用“ ”才能全部填充为设定字符“*”，否则将全部填充为空格；如果在程序中使用了设置标志，只有在清除设置标志之后，才能进行新的设置。

②使用setprecition设置浮点数输出精度

……

```
const double PI＝3.141592;
void main（）{
cout<<setprecition(0)<<PI<<endl
    <<setprecition(1)<<PI<<endl
    <<setprecition(2)<<PI<<endl
    <<setprecition(3)<<PI<<endl
    <<setprecition(7)<<PI<<endl;
    ……
    }
```

程序输出结果为：

```
3.14159
3
3.1
3.14
3.141592
```

注意使用setprecition(int n)设定显示小数位数时，小数点本身也占1位，0等于不设，由系统决定（默认为最多输出5位小数），1代表显示整数数字，2才显示小数点后面的一位数。因系统只输出5位小数，所以为了将6位小数全部输出，最后一行必须设置7位才行。

③使用dec、oct、hex设置转换基数为不同进制

……

```
int b＝100;
```

```
cout<<"Dec:"<<dec<<b<<endl      //十进制格式输出
   <<"Hex:"<<hex<<b<<endl      //十六进制格式输出
   <<"Oct:"<<oct<<b<<endl;      //八进制格式输出
  cout<<b<<endl
  <<"Input b =";
  cin>>b;          //输入 100
  cout<<b<<endl;
```

```
cout<<dec<<setiosflags(ios_base::showpos)<<b<<endl;
```

```
cout<<"Input b =";
cin>>b;          //输入 100
cout<<b<<endl;
cout<<resetiosflags(ios_base::showpos);
cout<<b<<endl;
……
```

程序输出结果为：

Dec：100

Hex：164

Oct：144

144

Input b=100　　//输入 100

144　　//因尚未转换基数，100 仍按八进制输出

＋100

Input b＝100　　//输入 100

＋100　　//已转换基数，将 100 按十进制输出

100　　//清除 flag 指定的标志位和正数前显示的＋号

如上面程序，程序执行 cout<<oct 语句后，将保持八进制格式输出，虽然输入 100，但输出仍按八进制，只有使用 cout<<dec 语句将其恢复为十进制。使用

语句 cout<<dec<<setiosflags(ios_base::showpos)<<b<<endl;除了将进制恢复为十进制，还将输出设置为在正的数字前面显示“＋”号，直到后面的语句使用清除该设置标志的语句 cout<<resetiosflags(ios_base::showpos);执行，另外如果单独使用语句“resetiosflags(ios_base::showpos);”，则不起作用。

§1.3 程序的编辑、编译和运行的基本概念

1、C＋＋程序编制过程

①先使用编辑器编辑一个C＋＋程序mycpp.cpp,又称其为C＋＋的源程序；

②然后使用C＋＋编译器对这个C＋＋程序进行编译,产生文件mycpp.obj；

③再使用连接程序（又称 Link），将 mycpp.obj 变成 mycpp.exe 文件。

2、C＋＋程序编制环境及使用方法

现在 C＋＋的编制一般都使用集成环境，如 Visual C＋＋6.0 等，所谓集成环境，就是将 C＋＋语言的编辑、编译、连接和运行程序都集成到一个综合环境中。

利用 VC 编制 C＋＋程序源文件的步骤如下：

①启动 VC6.0；

②File 菜单－New 对话框－Project 选项卡－Win32 Console Application 选项,在右边的 Project name 输入框中输入项目名称 myfile,在右边的 Location 输入框中输入存储目录，然后单击 OK 按键，进入 Win32 Console Application 制作向导的第一步，编辑 C＋＋程序文件是选择 An empty project 选项，单击 Finish 按钮，完成设置；

③选中 File View 选项卡，进入空项目，单击它展开树形结构，选中 myfile files 节点；选中 Source File 标记，再从 File 菜单中选 new 命令，弹出 new 对话框；选中 C＋＋Source File 选项，在右方的 File 输入框中输入 C＋＋程序文件名（mycpp），系统默认的文件扩展名为.cpp，单击 OK 按钮，返回集成环境，并在 Source File 项下面产生空文件 mycpp.cpp；在右边的源代码编辑框中输入源文件；

④部分 Build 菜单项描述

菜单项	描　　述
Compile	编译源代码窗口中的活动源文件
Build	查看工程中所有文件，并对最近修改过的文件进行编译和链接
Rebuild All	对工程中的所有文件全部进行重新编译和连接
Clean	删除项目的中间文件和输出文件
Start Debug	弹出级联菜单，主要包括有关程序调试的选项
Execute	运行应用程序

第二章　从结构到类的演变

§2.1　结构的演化

一、结构发生质的演变

1、函数与数据共存

C＋＋中首先允许结构中可以定义函数，这些函数称为成员函数，形式如下：

```
struct 结构名{
        数据成员
        成员函数
};
```

可以像C语言中结构变量使用结构成员的方法那样，通过C＋＋的结构对象使用成员函数，形式如下：

结构对象.成员函数

2、封装性

如果在定义结构时，将数据成员使用private关键字定义，则产生封装性，没有使用private定义的成员函数，默认为public。

要注意，私有的数据成员，必须通过公有的成员函数才能使用，而不能不通过公有的成员函数直接来使用，否则就会出错，这就称为数据的封装性。

二、使用构造函数初始化结构的对象

函数名与结构同名，称为构造函数，专门用于初始化结构的对象，构造函数使用的一般形式为：

构造函数名 对象名（初始化参数）；

程序在运行时，会自动完成初始化任务。

§2.2　从结构演变一个简单的类

1、用关键字class代替struct，就是一个标准的类。

实例：

```
#include <iostream>
using namespace std;
class Point{        //定义类 Point
private:
    double x,y;  //类 Point 的数据成员
public:
    Point( ){ };  //类 Point 的无参数构造函数
    Point(double a,double b) {x=a;y=b;}   //具有两个参数的构造函数
    void Setxy(double a,double b) {x=a;y=b;} //成员函数，用于重新设置数据成员
    void Display( ){cout<<x<<"\t"<<y<<endl;} //成员函数，按指定格式输出数据成员
    };
void main(){
    Point a;  //定义类 Point 的对象 a
    Point b(18.5,10.6);  //定义类 Point 的对象 b 并初始化
    a.Setxy(10.6,18.5);  //为对象 a 的数据成员赋值
    a.Display();  //显示对象 a 的数据成员
    b.Display();  //显示对象 b 的数据成员
}
```

程序运行结果：

```
10.6  18.5
18.5  10.6
```

2、类的示意图

上例中的 Point 类可以看作直角坐标系中的点类，其结构示意图如右：

类名 Point
具有的属性 x 和 y
提供的操作 构造函数 Point Setxy 用来给对象赋值 Display 用来输出 x 和 y

第一个方框中是类名；第二个方框中是坐标点的数据，称为属性（或称数据成员）；第三个方框中表示类所提供的具体操作方法，实际上是如何使用数据 x 和 y，以实现预定功能的函数，这里称为成员函数。

§2.3　面向过程与面向对象

1、面向过程的方法

所谓面向过程，就是不必了解计算机的内部逻辑，而把精力集中在对如何求解问题的算法逻辑和过程的描述上，通过编写程序把解决问题的步骤告诉计算机。

C 语言就是面向过程的结构化程序设计语言，其程序设计特点就是通过函数设计，实现程序功能的模块化、结构化。但实际工作中，尽管结构化程序设计中的分而治之的想法非常好，但在结构化程序设计语言和结构化程序设计方法下却难以贯彻到底，特别是在软件规模在三四万行以上时，开发和维护就十分困难。

2、面向对象的方法

为了解决面向过程的方法存在的问题，人们提出了面向对象的方法。所谓对象，就是现实世界中客观存在的事务。相对于过程，对象是稳定的，复杂的对象可由简单的对象组成，对象各自完成特定的功能。

在面向对象程序设计中，可以将一组密切相关的函数统一封装在一个对象中，从而合理而又有效的避免全局变量的使用，更彻底的实现了结构化程序设计的思想。

结构化程序设计使用的是功能抽象，而面向对象程序设计不仅能进行功能抽象，还能进行数据抽象，“对象”实际上是功能抽象和数据抽象的统一。

面向对象的程序设计方法不是以函数过程和数据结构为中心，而是以对象代表来求解问题的中心环节，追求的是现实问题空间与软件系统解空间的近似和直接模拟，从而使人们对复杂系统的认识过程与系统的程序设计实现过程尽可能的一致。

3、软件开发过程及发展趋势

软件开发者将被开发的整个业务范围称为问题域，软件开发是对给定问题求解的过程，分为两项主要内容：认识和描述。“认识”就是在所要处理的问题域范围内，通过人的思维，对该问题域客观存在的事务以及对所要解决的问题产生正确的认识和理解。“描述”就是用一种语言把人们对问题域中事务的认识、对问题及解决方法的认识描述出来，最终的描述必须使用一种能够被机器读懂的语

言，即编程语言。

人们习惯使用的自然语言和计算机能够理解并执行的编程语言之间存在很大的差距，称为“语言的鸿沟”。由于人的认识差距，问题域和自然语言之间也有缝隙，机器语言和自然语言之间鸿沟最宽，程序设计语言的发展趋势则是为了鸿沟变窄。

§2.4 C＋＋面向对象程序设计的特点

面向对象的程序设计具有抽象、封装、继承和多态性等关键要素。

1、对象

C＋＋中的对象是系统中用来描述客观事物的一个实体，是构成系统的一个基本单位，C＋＋中使用对象名、属性和操作三要素来描述对象，如右所示：对象名用来标识一个具体对象；用数据来表示对象的属性，一个属性就是描述对象静态特征的一个数据项；操作是描述对象动态特征（行为）的一个函数序列（使用函数实现操作），也称方法或服务。数据称为数据成员，函数称为成员函数，一个对象由一组属性和对这组属性进行操作的成员函数构成。

对象名
属性 1 属性 2 …… 属性 n
操作 1 操作 2 …… 操作 n

例：用简单对象表示平面上的 A（3.5，6.4）和 B（8.5，8.9）两个坐标点。

可用如下的对象结构图表示 A 和 B 的对象结构：

A
x（3.5） y（6.4）
Display（）; Setxy（）; Move（）;

B
x（8.5） y（8.9）
Display（）; Setxy（）; Move（）;

2、抽象和类

抽象是一种从一般的观点看待事物的方法，即集中于事物的本质特征，而不

是具体细节或具体实现。面向对象的方法把程序看作是由一组抽象的对象组成的，如果把一组对象的共同特征进一步抽象出来，就形成了“类”的概念。

对于一个具体的类，它有许多具体的个体，这些个体叫做“对象”，同一类的不同对象具有相同的行为方式。一个对象是由一些属性和操作构成的，对象的属性和操作描述了对象的内部细节，类是具有相同的属性和操作的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和操作两个主要部分。

类的作用是定义对象，类和对象的关系如同一个模具和其铸造出来的铸造件的关系，对象之间就像是同一模具铸出的零件，模样相同，铸造材料可能不同。类给出了属于该类的全部对象的抽象定义，而对象则是符合这种定义的实体。所谓一个类的所有对象具有相同属性和操作，是指它们的定义形式(即属性的个数、名称、数据类型）相同，而不是说每个对象的属性值都相同。

3、封装

为了保护类的安全，即限制使用类的属性和操作，需要将类封装起来。封装就像用同一个模具铸造零件，各零件使用的材料（数据成员）和铸造工艺（成员函数）均不同，每一种材料都有其对应的铸造工艺，材料与铸造工艺是成套使用（封装）的，虽然铸出零件的模样一样，但如果用错了铸造工艺，就可能出废品。

所谓“封装”，就是把对象的属性和操作结合成一个独立的系统单位，并尽可能隐蔽对象的内部细节。按照面向对象的封装原则，一个对象的属性和操作是紧密结合的，对象的属性只能由这个对象的操作来存取。

对象的操作分为内部操作和外部操作，前者只供对象内部的其他操作使用，不对外提供；后者对外提供一个信息接口，通过这个接口接受对象外部的消息并为之提供操作（服务)。对象内部数据结构的这种不可访问性称为信息（数据）隐藏。

数据封装给数据提供了与外界联讯的标准接口，只有通过这些接口，使用规范的方式，才能访问这些数据，同时程序员也只需要和接口打交道，而不必了解数据的具体细节。

由此可见，封装要求一个对象应具备明确的功能，并具有接口以便和其他对象相互作用，同时，对象的内部实现（代码和数据）是受保护的，外界不能访问它们，这样封装一方面使得程序员在设计程序时能专注于自己的对象，同时也切断了不同模块之间数据的非法使用，减少了出错。

在类中，封装是通过存取权限实现的，例如每个类的属性和操作分为私有和公有两种类型，对象的外部职能访问对象的公有部分，而不能直接访问对象的私有部分。

4、继承

继承是一个类可以获得另一个类的特性的机制，支持层次概念，通过继承，低层的类只需定义特定于它的特征，而共享高层的类中的特征。继承具有重要的实际意义，它简化了人们对事物的认识和描述。

继承就像铸造中母模与子模的关系。

5、多态性

不同的对象可以调用相同名称的函数，但可导致完全不同的行为的现象称为多态性。利用多态性，程序中只需进行一般形式的函数调用，函数的实现细节留给接受函数调用的对象，这大大提高了解决人们复杂问题的能力。

多态性就像是铸造时不同的零件、不同材料所铸的同一款零件虽然可以使用相同的铸造工艺，但铸出的零件用途、使用寿命和使用方法都不一样。

§2.5 使用类和对象

1、使用 string 对象

实际上 string 类很复杂，如右的 string 类的简化图中只给出了下例中涉及的属性和部分操作。

string
str
string find size substr

由图，类 string 的属性是一个字符串 str，同名函数 string 是构造函数，用来初始化字符串，另外三个成员函数用来对属性 str 进行操作，其中 find 成员函数用来在 str 字符串中检索需要的子串；size 成员函数计算并输出 str 存储的单

词长度；substr 成员函数用来返回 str 字符串中的子串。

在程序中可以使用 string 类定义存储字符串的对象，这些对象属于 string 类，因此还要使用#include <string>来包含这个类的头文件。

因为 string 需要的是字符串,所以 string 类的对象不能用单引号括起来的单个字符常量初始化，即：

string str ='A'; × //同理 string str('A'); ×

但可以使用双引号括起来的单个字符常量初始化，即：

string str = "A"; √ //同理 string str("A"); √

如果 string 的对象 str 的内容为"ab"，则 str[0]='a', str[1]='b' 。

例：使用 string 对象及初始化

```
#include <iostream>
#include <string>          //在程序中包含 string 类的头文件
using namespace std;
void main( ){
    string str1("We are here!"); //用构造函数 string 定义对象 str1 并赋值
    string str2="Where are you?";//用构造函数 string 定义对象 str2 并赋值
    (或 string str1("We are here!"), string str2="Where are you?";)
    cout<<str1[0]<<str1[11]<<","<<str1<<endl;
    cout<<str2[0]<<str2[13]<<","<<str2<<endl;
    cout<<"please input a word:";
    cin>>str1;     //输入 good
    cout<<"lenght of the "<<str1<<" is "<<str1.size( )<<"."<<endl;
}
```

程序运行结果为：

```
W! We are here!
W? Where are you?
Please input a word: good
```

Length of the good is 4.

程序中使用了两种方法给 string 类的两个对象初始化，也可将它们定义在一行中：string str1("We are here!"), string str2="Where are you?";

string 类还提供将两个字符串连接起来组成一个新字符串的能力，用“＋”号将后面的字符串连接到前一个字符串的后面,也可以与单个字符常量连接,如：

```
str1＝str1＋“ ”＋str1;
```

将原来的两个 str1 的内容用空格连接起来。

2、使用 string 类的典型成员函数

string 对象是通过调用成员函数实现操作，从而提供对象的行为或消息传递的，**对象调用成员函数的语法为：**

对象名称.成员函数（参数（可供选择的消息内容））

常用的 string 类成员函数：

substr：
```
string newstr＝str1.substr(3,3);
```

find：
```
int i ＝str1.find(“are”,0);
```

getline：
```
getline (cin, InputLline, ’\n’);
```

swap：
```
str1.swap(str2);＝str2.swap(str1);
```

begin 和 end：
```
copy(str1.begin( ),str1.end( ),str2.begin( ));
```

①成员函数 substr 用来返回给定字符串的子串，格式为：

对象名称.substr（要截取子串的起始位置，截取的长度);

如：string str1("We are here!"); 语句中要从对象 str1 中截取字符串 are，可用下面的语句实现：

```
string newstr＝str1.substr(3,3);
```

此时 newstr 的内容为 are,括号中的第一个 3 是因为 **C＋＋规定字符串的计数从 0 开始，**所以 a 处于位置 3；第二个 3 是因为 are 是 3 个字符，所以截取子串的长度为 3。

注意给出的要截取子串的起始位置必须位于字符串中，否则出错。如果截取长度大于字符串长度，则是可以的，程序将自动截取到字符串末尾，如语句 string strnew ＝ newstr.substr(2,8); 和语句 string strnew ＝

newstr.substr(2,1);等效，都是截取字符 e。

②成员函数 find 用来在主串中检索所需字符串，格式为：

对象名称.find（要查找的字符串，开始查找的位置）；

函数返回查找到的字符串在主串中的位置，如：

```
int i =str1.find(“are”,0);
```

表示从 str1 字符串的位置 0 开始查找 are 出现的位置，结果为 3。

如果不给位置参数，默认位置参数为 0。

③string 类还提供一个辅助功能，以便使用 getline 从流 cin 中读出输入的一行给 string 类的对象，如：

```
string InputLine;
getline (cin, InputLline, ’\n’);
cout<<”your input:”<<InputLine<<endl;
```

如果输入“I am here!”，则得到如下结果：

```
your input:I am here!
```

例：将美国格式的日期转换为国际格式

```
#include <iostream>
#include <string>
using namespace std;
void main(){
   cout<<"Enter the date in American format "
      <<"(例如：December 25, 2002):";  //注意逗号后面有一个空格
   string Date;
   getline(cin,Date,'\n');        //输入时间：May 28, 2002
   int i = Date.find(" ");         //i＝3
   string Month=Date.substr(0,i);   //截取月份赋值给字符串 Month
   int k = Date.find(",");          //k＝6
   string Day=Date.substr(i+1,k-i-1);  //截取日期赋值给字符串 Day
```

```
    string Year=Date.substr(k+2,4);      //截取年份赋值给字符串 Year
    string NewDate=Day+" "+Month+" "+Year;
    cout<<"Original date:"<<Date<<endl;
    cout<<"Converted date:"<<NewDate<<endl;
}
```

程序运行实例：

```
Enter the date in American format(例如：December 25, 2002):May 28, 2002
Original date:May 28,  2002
Converted date:28 May 2002
```

3、使用 complex 对象

C＋＋标准库提供 complex 类定义复数对象，在程序中包含这个类的头文件为：#include <complex>

复数（complex number）类需要两个初始值：实部和虚部，complex 是一个模板类，利用构造函数初始化的格式为：

comlex <数据类型> 对象名（实部值，虚部值);

如：

```
complex <int> num1(2,3);   //定义复数 2＋3i
complex <float> num2(2.5,3.6);   //定义复数 2.5＋3.6i
```

complex 的 real 和 imag 成员函数用来输出对象的实部和虚部的值，如：

```
cout<<num2.real( )<<","<<num2.imag( )<<endl;
```

4、使用对象小结

标准库提供的类都是经过抽象，代表了一类对象，例如 string 类描述的是字符串特性，具有一个用来描述对象静态性质的字符串，字符串的值可以区分不同的对象;通过一系列的操作方法对这个字符串进行操作,用函数实现这些方法，又称这些函数为成员函数；数据成员（属性）和成员函数（方法）代表了字符串一类事物的特征。

String 类的使用方法一般如下：

```
#include <string>
```

```
string str1;  //定义这个类的一个对象 str1
str1＝“this is a string”;   //给 str1 赋值
string str2 (“this is a str2”); //定义 str2 并赋值
cout<<str1<<str2.size( );
```

可以先定义对象，然后给它赋值（如上式 str1），也可以在定义对象的同时初始化对象（如上式 str2），如果要使用对象的成员函数，则用“.”运算符（如最后一句）。

同理，复数类可以抽象为具有实部和虚部的数据成员，以及能对复数进行基本操作的成员函数，与 string 不同的是，定义复数类时与数据成员的类型无关，当定义复数类的对象时才指定实部和虚部的数据类型。

注意，类是抽象出一类物质的共同特征，模板则是归纳出不同类型事物的共同操作。

§2.6 string 对象数组与泛型算法

1、第 1.2.6 节介绍的泛型算法同样适合 string 类，但要注意不要将该节介绍的 find 函数与 string 本身的 find 函数混淆。

另外，string 类还有一个 swap 成员函数，用来交换两个对象的属性。假设有两个 string 类的对象 str1 和 str2，下面两种调用方式是等效的：

```
str1.swap(str2);
str2.swap(str1);
```

例：演示 string 对象

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
void main( ){
    string str1=”we are here!”,str2=str1;
```

```
reverse(&str1[0],&str1[0]+12);
copy(&str1[0],&str1[0]+12,&str2[0]);
cout<<str1<<str2<<endl;
reverse_copy(&str2[0],&str2[0]+12,ostream_iterator<char>(cout));
}
```

程序运行结果:! ereh era ew

! ereh era ew

we are here!

该例中用 str1 初始化 str2，是为了保证在复制时 str2 能有足够的空间存储 str1，另外要注意 ostream_iterator 的数据类型是 char，不是 string。

2、string 类的成员函数 begin 和 end

二者都用来指示元素位置，前者指示第一个元素，后者指示最后一个元素后面的字符串结束位置。如果 begin 不等于 end，算法首先作用于 begin 所指元素，并将 begin 前进一个位置，然后作用于当前的 begin 所指元素，如此继续前进，直到 begin 等于 end 为止，所以它们是元素存在的半开区间 [begin，end）。

在实例中，例如 str2 值为“wrrheeeea!”，则 reverse(str2.begin( )+2，str2.begin()+8)；执行后 str2 值为“wreeeehra!”，这是因为 begin（）+2～begin+8 是一个半开区间，区间包含的元素不是 7 个，而是上面阴影中包含的 6 个，即处于 begin（）+8 位置的元素 a 并不被包含在这个半开区间中。同理，上例中的&str1[0]～&str1[12]也是这样的一个半开区间，包含的元素是 12 个而不是 13 个。

有了这两个成员函数，就可以用它们表示元素存储区间，使用 string 定义的字符串中不用字符'\0'为结束符，而使用 char 定义的字符串则自动在尾部加入'0'作为结束符。

例：演示 string 对象使用成员函数表示存储空间

```
#include <iostream>
#include <string>
#include <algorithm>
```

```
#include <functional>
using namespace std;
void main( ){
    string str1="wearehere!",str2(str1);
    reverse(str1.begin( ),str1.end( ));  //str1 逆向
    cout<<str1<<endl;      //输出 str1=“!ereheraew”

    copy(str1.begin( ),str1.end( ),str2.begin( ));
    sort (str1.begin( ),str1.end( ));  //按默认升幂排序 str1
    cout<<str1<<endl;              //输出 str1＝“!aeeeehrrw”
    cout<<str2<<endl;              //输出 str2＝“!ereheraew”

    reverse_copy(str1.begin( ),str1.end( ),str2.begin( ));
    cout<<str2<<endl;              //输出 str2＝“wrrheeeea!”

    reverse(str2.begin( )+2,str2.begin( )+8); //此时 str2=“wreeeehra!”
    copy(str2.begin( )+2,str2.begin( )+8,ostream_iterator<char>(cout));
                                   //输出“eeeehr”

    sort(str1.begin( ),str1.end( ),greater<char>( ));  //str1 降幂排列
    cout<<str1<<endl;              //输出 str1＝“wrrheeeea!”

    str1.swap(str2);               //互换内容
    cout<<str1<<” ”<<str2<<endl;
                                   //输出 wreeeehra!(str1)  wrrheeeea!(str2)

    cout<<(*find(str1.begin( ),str1.end( ),'e')= ='e')<<” ”
        (*find(str1.begin( ),str1.end( ),'e')= ='o')<<endl;
    }                              //输出 1 0 ，注意上面的 find 不是成员函数 find
```

3、虽然可以声明 string 类的数组，但只能对数组分量使用这些操作，不能对整

个数组使用这些操作，swap 成员函数可以用来交换两个数组分量。

例：演示 string 对象数组

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
void main( ){
    string str[ ]={"we are here!","where are you?","welcome"};
    for(int i = 0;i<3;i++){
        copy(str[i].begin( ),str[i].end( ),ostream_iterator<char>(cout));
        cout<<endl;
    }    //for 循环，换行分别输出 we are here!  Where are you?  Welcome!
    str[0].swap(str[2]);
                //互换，str[0]＝“Welcome!” str[2]＝“we are here!”
    str[0].swap(str[1]);
                //互换，str[0]＝“Where are you?” str[1]＝“Welcome!”
    for(i=0,i<3,i++)
        cout<<str[i]<<endl;      //for 循环,换行分别输出 Where are you?
                                          Welcome!     We are here!
}
```

程序输出结果为：

```
we are here!
Where are you?
Welcome!
Where are you?
Welcome!
We are here!
```

注意除非输出部分内容，否则使用 cout<<str[i]的方法更方便。

第三章　函数和函数模板

C 语言补充资料：

一、函数的参数：函数的参数分为形参和实参两种。形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是做数据传送，发生函数调用时，主调函数把实参的值传送给被调函数的形参，从而实现主调函数向被调函数的数据传送。

二、函数的形参和实参具有以下特点：

1、形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只有在函数内部有效，函数调用结束返回主调函数后，则不能再使用该形参变量。

2、实参可是是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此，应预先用赋值、输入等办法使实参获得确定值。

3、实参和形参在数量上、类型上、顺序上应严格一致，否则会发生“类型不匹配”的错误。

4、函数调用中发生的数据传送是单向的，即只能把实参的值传送给形参，而不能把形参的值反向的传给实参，因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

§3.1 函数的参数及其传递方式

一、C＋＋的函数参数传递方式

C 语言函数参数的传递方式只有传值一种，又分为传变量值和传变量地址值两种情况，而 C＋＋的函数参数传递方式有两种：第一种和 C 语言一样，是传值；第二种是传引用，即传对象的地址，所以也称传地址方式。

注意传地址值传递的是值，是以对象指针作为参数；而传地址传递的是地址，是以对象引用作为参数。

所以在设计函数参数时，可以使用“对象”、“对象指针”和“对象引用”作为参数。

二、对象作为函数参数

使用对象作为函数参数,是将实参对象的值传递给形参对象,传递是单向的,形参具有实参的备份，当在函数中改变形参的值时，改变的是这个备份中的值，不影响原来实参的值。

例：传对象不改变原来对象数据成员值的例子。

```
①  #include <iostream>
   #include <string>
   using namespace std;
   void swap(string,string);     //定义函数 swap，使用 string 类的对象作为函数形参
   void main( ) {
       string str1("现在"),str2("过去");    //定义对象 str1 和 str2
②      swap(str1,str2);   //使用传值方式传递函数实参 str1 和 str2 的数据成员值
④      cout<<"返回后：str1＝"<<str1<<" str2="<<str2<<endl;
   }
③  void swap(string s1, string s2)   //定义 string 类的对象 s1 和 s2 作为函数形参
   {
       string temp=s1; s1=s2; s2=temp;
       cout<<"交换为：str1＝"<<s1<<" str2="<<s2<<endl;
   }
```

程序执行步骤：

第一步：先执行①所包含的语句；

第二步：执行②语句，发生函数调用，主调函数 swap（str1，str2）将实参 str1 和 str2 的值分别传递给被调函数 swap（s1，s2）的形参 s1 和 s2。

第三步：执行③所包含语句，即函数跳转到被调函数的函数体执行，直至被调函数结束。

第四步：执行④语句，即函数调用结束，函数返回发生调用语句的下一个语句继续执行。

程序输出结果：交换为：str1＝过去 str2＝现在

返回后：str1＝现在 str2＝过去

三、对象指针作为函数参数

使用指向对象的指针作为函数参数，形参是对象指针（指针可以指向对象的地址），实参可以是对象的地址值，而不一定非得是指针。虽然参数传递方式仍然是传值方式，但因为形参传递的就是实参本身，所以当在函数中改变形参的值时，改变的就是原来实参的值。

传对象地址值要用到对象的指针，而对于数组，因数组名就是数组的指针名，所以数组也能用传数组地址值的方式。

例：使用对象指针作为函数参数的例子。

①
```
#include <iostream>
#include <string>
using namespace std;
void swap(string *,string *); //定义函数 swap，使用 string 类的指针作为函数形参
void main( ) {
    string str1("现在"),str2("过去");    //定义对象 str1 和 str2
```

②
```
    swap(&str1,&str2);  // 因函数原型中参数的类型是指针，所以 string *s1=&str1;
```
是完全正确的，即在主调函数中可将对象 str1 和 str2 的首地址值&str1 和&str2 作为实参，并传递给形参

④
```
    cout<<"返回后：str1＝"<<str1<<" str2="<<str2<<endl;
}
```

③
```
void swap(string *s1, string *s2) //string 类的对象指针 s1 和 s2 作为函数形参
{
    string temp=*s1; *s1=*s2; *s2=temp;
    cout<<"交换为：str1＝"<<*s1<<" str2="<<*s2<<endl;
}
```

因为采用的是传地址值的方式，实参与形参的地址相同，所以改变形参的值的同时也改变了实参的值。

程序输出结果：交换为：str1＝过去 str2＝现在

返回后：str1＝过去 str2＝现在

在上例中，因为函数原型参数的类型是指针，可以直接指向对象地址，即string *s1=&str1；是完全正确的，所以主函数中，实参的产生并不是非先要在主程序中产生指针，然后再使用指针作为实参，而是直接使用&str1 和&str2 作为了实参，这是因为使用&str1 和&str2 标识的是取对象 str1 的首地址值，所以可以直接作为实参。

另外在以数组作为函数参数时，因数组名即是数组指针名，指向数组首地址（例如 a 为数组名,则 a 同时也是数组指针名,a+1 则指向数组 a 的第一个元素），所以也能采用传地址值的方式，当交换后，因形参的改变，实参也会同时改变。

例：传递数组名实例。

```
#include <iostream>
using namespace std;
void swap(int[ ]);
void main( ){
    int a[ ]={3,8};
    swap(a);
    cout<<"返回后：a="<<a[0]<<" b="<<a[1]<<endl;
}
void swap(int a[ ])
{
    int temp= a[0]; a[0]=a[1]; a[1]=temp;
    cout<<"交换为：a="<< a[0]<<" b="<< a[1]<<endl;
}
```

程序运行结果为：交换为：a＝8 b＝3

返回后：a＝8 b＝3

四、引用作为函数参数

使用引用作为函数参数，函数调用时，把实参对象的地址传给形参对象，使形参对象的地址取实参对象的地址，从而使形参对象和实参对象共享同一个单元，（也可形象的记为实参对象名传给形参对象名，形参对象名成为实参对象名的别名）所以改变形参对象的值就是改变实参对象的值。

例：使用引用作为函数参数的例子。

```
①  #include <iostream>
   #include <string>
   using namespace std;
   void swap(string &,string &); //函数 swap，使用 string 类的引用对象作为函数形参
   void main( ) {
       string str1("现在"),str2("过去");    //定义对象 str1 和 str2
②      swap(str1,str2);  // 传递对象的名字 str1 和 str2
④      cout<<"返回后：str1＝"<<str1<<" str2="<<str2<<endl;
   }
③  void swap(string &s1, string &s2) //string 类的引用对象 s1 和 s2 作为函数形参
   {
       string temp=s1; s1=s2; s2=temp;
       cout<<"交换为：str1＝"<<s1<<" str2="<<s2<<endl;
   }
```

程序输出结果：

交换为：str1＝过去 str2＝现在

返回后：str1＝过去 str2＝现在

在程序中调用函数 swap 时，实参 str1 和 str2 分别初始化引用 s1 和 s2，在函数 swap 中，s1 和 s2 分别是 str1 和 str2 的别名，形参对象 s1 和 s2 和实参对象 str1 和 str2 共享同一个单元(系统向形参传递的是实参的地址而不是实参的值)，对 s1 和 s2 的访问就是对 str1 和 str2 的访问，所以函数 swap 改变了 main 函数中变量 str1 和 str2 的值。

因为引用对象不是一个独立的对象，不单独占用内存单元，而对象指针要另外开辟内存单元（其内容是所指向对象的地址），所以传引用要比传指针更好。但要注意，虽然系统向形参传递的是实参的地址而不是实参的值，但实参必须使用对象名，例如：“swap（str1，str2);”正确，而“swap（&str1，&str2);”错误。

五、可以通过间接引用的方法使用数组，如下例：

```
#include <iostream>
using namespace std;
typedef double array[12];          //自定义数组标识符 array
void avecount(array& b,int n)      //定义函数 avecount，其形参一个使用引用，
{                                                    一个使用对象
    double ave(0);
    int count(0);  //累加器初始化 0
    for(int j=0;j<n-2;j++){
        ave=ave+b[j];
        if(b[j]<60) count++;
    }
    b[n-2]=ave/(n-2);   //平均成绩
    b[n-1]=count;       //不及格人数
}
void main( ){
array b={12,34,56,78,90,98,76,85,64,43};
    array &a=b;
    avecount(a,12);
    ccout<<"平均成绩为"<<a[10]<<"分，不及格人数为"<<int(a[11])<<"人。"<<endl;
}
```

```
For(表达式 1；表达式 2；表达式 3) {
    语句 1；
    语句 2；
}
语句 3；
语句 4；

上面 for 循环的执行步骤为：
1、执行表达式 1；
2、执行语句 1、2，并返回到表达式 2；
3、如表达式 2 值为真，执行表达式 3，然后执行语句 1、2，并返回到表达式 2，直到表达式 2 值为假；
4、执行语句 3、4
```

程序输出结果：

平均成绩为 63.6 分，不及格人数为 4 人。

六、默认参数

默认参数就是不要求程序员设定该参数， 而由编译器在需要时给该参数赋默认值。当程序员需要传递特殊值时，必须显式的指明。默认参数必须在函数原型中说明，默认参数可以多于 1 个，但必须放在参数序列的后部。例如：

int SaveName(char *first,char *second=" ",char *third=" ",char *fourth=" ");

表明在实际调用函数 SaveName 时，如不给出参数 second、third 和 fourth，则取默认值“ ”。

另外，如果某个默认参数需要指明一个特定值，则在此之前的所有参数都必须赋值，如上例中，如果需要给出参数 third 的值，则必须同时也给 first 和 second 赋值。

例：设计一个根据参数数量输出信息的函数

```
#include <iostream>
#include <string>
using namespace std;
void Display(string s1,string s2=" ",string s3=" ");    //语句①
void main( ){
    string str1("现在"), str2("过去"), str3("未来");   //语句②
    Display(str1);                  //语句③，输出结果为现在
    Display(str1,str2,str3);        //语句④，输出结果为现在、过去、未来
    Display(str3,str1);             //语句⑤，输出结果为未来、现在
    Display(str2,str3);             //语句⑥，输出结果为过去、未来
}
void Display(string s1,string s2,string s3)
{
    if(s2= =" "&s3= =" ")          cout<<s1<<endl;
    else if(s3= =" "&s2!=" ")    cout<<s1<<"、"<<s2<<endl;
    else                         cout<<s1<<"、"<<s2<<"、"<<s3<<endl;
```

语句⑦

}

上面程序主要执行步骤为：

第一步，执行语句①，声明函数 Display，该函数有三个 string 类的形参，其中形参 s2 和 s3 设定为默认参数“ ”，即程序中 s2 和 s3 如不取特定值，则由程序取默认值“ ” ；

第二步，执行语句②，定义 string 类对象 str1、str2 和 str3 并初始化；

第三步，执行语句③，调用语句⑦，将实参 str1 的值赋给语句⑦中函数 Display 的形参 s1，同时因为 str2 和 str3 设定了默认参数而又未取特定值，所以都取默认值“ ”，执行语句⑦中的 if 条件语句，输出结果“现在”后，返回至语句④位置；

第四步，执行语句④，调用语句⑦，将实参 str1、str2、str3 的值赋给语句⑦中函数 Display 的形参 s1、s2、s3，执行语句⑦中的 if 条件语句，输出结果“现在、过去、未来”后，返回至语句⑤位置；

第五步，执行语句⑤，调用语句⑦，将实参 str3、str1 的值赋给语句⑦中函数 Display 的形参 s1、s2，执行语句⑦中的 if 条件语句，输出结果“未来、现在”后，返回至语句⑥位置；（注意是实参 str3 对应形参 s1，实参 str1 对应形参 s2，而不是实参 str3 对应形参 s3，实参 str1 对应形参 s1）

第六步，执行语句⑥，调用语句⑦，将实参 str2、str3 的值赋给语句⑦中函数 Display 的形参 s1、s2，执行语句⑦中的 if 条件语句，输出结果“过去、未来”后，返回至语句⑥位置；（注意是实参 str2 对应形参 s1，实参 str3 对应形参 s2，而不是实参 str2 对应形参 s2，实参 str3 对应形参 s3）

七、使用 const 保护数据

可以用 const 修饰要传递的参数，意思是通知函数，它只能使用参数而无权修改参数，以提高系统的自身安全，C＋＋中普遍使用这种方法。

例：不允许改变作为参数传递的字符串内容的实例

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

```
void change(const string&);                    //语句①
void main( ){
    string str(“can you change it?”)     //语句②
    change(str);                                 //语句③
    cout<<str<<endl;                             //语句④
}
void change(const string&s)
{
    string s2=s+”no!”;
    cout<<s2<<endl;
}
```

语句⑤

上面程序的主要执行步骤为：

第一步，执行语句①，声明函数 change 使用不允许改变的 string 类的引用对象作为函数形参；

第二步，执行语句②，定义 string 的对象 str 并赋初值；

第三步，执行语句③，调用语句⑤，将实参 str 的值赋给语句⑤中函数 change 的形参 s，执行语句⑤中的语句，输出结果“can you change it? no!”后，返回至语句④位置；注意：因用 const 限定了 change 函数的形参 string 类的引用对象 s 不能改变，所以在语句⑤中又新定义了一个 string 类的对象 s2，然后将 s＋”no!”的值赋给了 s2，而不是像以前的例子那样用语句 s＝s+”no!”。

第四步，执行语句④，输出结果“can you change it?”。

§3.2 深入讨论函数返回值

C＋＋函数的返回值类型可以是除数组和函数以外的任何类型，非 void 类型的函数必须向调用者返回一个值，数组只能返回地址。当返回值是指针或引用对象时，需要特别注意：函数返回所指的对象必须继续存在，因此不能将函数内部的局部对象作为函数的返回值。

一、返回引用的函数

函数可以返回一个引用，这样的目的是为了将该函数用在赋值运算符的左边，因为其他情况下，一个函数是不能直接用在赋值运算符左边的。

返回引用的函数原型的声明方式为：

数据类型 & 函数名（参数列表);

例：返回引用的函数实例

```
#include <iostream>
using namespace std;
int a[ ]={2,4,6,8,10,12};      //语句①，定义全局数组 a
int & index(int i);            //语句②，返回引用的函数 index 的原型声明
void main( ){
    index(3)=16;               //语句③，将 a[3]的值改为 16
    cout<<index(3)<<endl;      //语句④，输出 a[3]的值 16
}
int & index(int i)             //定义返回引用的函数 index
{return a[i];}                 //返回指定下标的整型数组内容
```

语句⑤

上面函数的主要执行步骤为：

第一步，执行语句①，定义全局数组 a;

第二步，执行语句②，声明函数 index 返回值为引用

第三步，执行语句③，调用语句⑤，将实参的值 3 赋给形参 i，执行语句⑤中的语句，返回 a{3}的值，（数组内容计数是从 0 开始的，所以 a[3]的值为 8）并将其值改为 16;

注意：因函数 index 不是 void 型的，所以必须有返回值，即必须有 return……;语句，否则编译将出错。

第四步，执行语句④，输出改后的 a[3]的值 16。

二、返回指针的函数

指针函数：返回值是存储某种类型数据的内存地址的函数。

返回指针的函数原型的声明方式为：

数据类型 *函数名（参数列表);

例：使用函数 input 输入一组数并返回一个指针，然后由 main 显示这组数

```
#include <iostream>
using namespace std;
float * input(int&);          //语句①，声明返回指针的函数 input
void main( ){
    int num;
    float *data;              //语句②，声明与 input 类型一致的指针变量 data
    data=input(num);          //语句③，调用函数，返回指针赋给 data
    if(data){                 //data 不空，返回所指内容
    for(int i=0;i<num;i++)    //使用指针的下标形式
        cout<<data[i]<<" ";   //循环输出 data 内容
    delete data;              //释放指针占用的内存
    }
}
float *input(int& n)                  //语句⑤，定义返回指针的函数 input
{   cout<<"Input number:";            //询问输入数据的个数
    cin>>n;
    if(n<=0) return NULL;             //输入的个数不合理则退出
    float * buf=new float[n];         //根据输入数据的个数申请所需空间
    if(buf= =0) return NULL;          //申请不到空间则退出
    for(int i=0;i<n; i++)
        cin>>buf[n];
```

语句④

语句⑥

语句⑦

```
    return buf;                          //返回指针
}
```

上面程序的主要执行步骤为：

第一步，执行语句①，声明返回值为指针的函数 input，该函数使用 int 型的引用作为形参；

第二步，执行语句②，声明返回值与 input 函数一样的指针变量 data；

第三步，执行语句③，调用 input 函数，执行语句⑤，定义返回值为指针的函数 input，该函数使用 int 型的引用 n 作为形参，用实参 num 初始化形参 n，n 作为 num 的引用；执行语句⑥，询问输入数据的个数 n，并输入 n，然后声明 float 型的指针变量 buf，并分配 n 个 float 型数据的存储空间，在 n<=0 或申请不到空间时退出；执行语句⑦，逐个将输入的 buf[i]的值存入已分配的空间中，然后返回指针 buf 赋给指针 data；

注意虽然系统向形参传递的是实参的地址而不是实参的值,但实参必须使用对象名，而形参的定义形式必须使用地址值的形式，如“data=input(num);”正确，而“data=input(&num);”错误；“float *input(int& n)”正确，而“float *input(int n)”错误；

第四步，执行语句④，如 data 内容不空，则将 data[i]逐个输出，输出完成后，删除指针 data，释放已动态分配的空间。

注意：调用 input 函数时，使用了和该函数返回类型一致的指针变量 data 保存 input 函数返回的指针，因是主程序调用 input 函数，所以由主程序负责释放由 input 函数分配的内存空间。

三、返回对象的函数

返回对象的函数原型的声明方式为：

数据类型 函数名（参数列表）

例：函数返回对象的实例

```
#include <iostream>
```

```
#include <string>
using namespace std;
string input(const int)        //语句①，声明返回 string 类的对象的函数
void main( ){
    int n;
    cout<<"Input n=";                                          语句②
    cin>>n;                          //输入要处理的字符串个数 n
    string str=input(n);             //语句③，将函数返回的对象赋给对象 str
    cout<<str<<endl;
}
string input(const int n)
{   string s1,s2;              //建立两个 string 类的对象（均为空串）
    for(int i=0;i<n;i++)       //逐次接收 n 个字符串                    语句④
      {cin>>s1;s2=s2+s1+" ";}  //逐次将接收的字符串相连
    return s2;                 //返回字符串相连的最后结果
}
```

上面程序的主要执行步骤为：

第一步，执行语句①，声明返回 string 类的对象的函数 input，该函数使用被 const 保护的整型数作为形参；

第二步，执行语句②，定义整型数 n 并输入 n 的值；

第三步，执行语句③，调用语句④，用构造函数初始化两个 string 类的对象（均为空串），逐次输入 n 个字符串并用空格相连，返回字符串相连的最后结果（string 类的对象 s2），并将其值赋给 string 类对象 str，然后输出 str。

程序运行结果为：

Input n＝3

Zhang san feng（分三次输入，一次输入一个字符串）

Zhang san feng

四、函数返回值作为函数的参数

如果函数返回值作为另一个函数的参数,那么这个返回值必须与另一个函数的参数的类型一致。

例：函数返回值作为函数的参数的实例

```
#include <iostream>
using namespace std;
int max(int,int);    //声明含有两个整型参数的函数原型
void main( )
{   cout <<max(55,max(25,39))<<endl;}
int max(int m1,int m2)    //定义含有两个整型参数的函数原型
{return (m1>m2)?m1:m2;}
```

注：上面语句“(m1>m2)?m1:m2”为条件运算符,相当于语句“if(m1>m2) max=m1; else max=m2;”

§3.3 内联函数

使用关键字 inline 说明的函数称为内联函数，内联函数必须在程序中第一次调用此函数的语句出现之前定义，在C＋＋中，除具有循环语句、switch 语句的函数不能说明为内联函数外，其他函数都可以说明为内联函数。使用内联函数可以提高程序执行速度，但如果函数体语句多，则会增加程序代码的大小。

例：使用函数 isnumber 判定一个输入字符是否是数字。

1、不使用内联函数形式时的代码：

```
#include <iostream>
using namespace std;
int isnumber(char c){return (c>='0'&&c<='9')?1:0;}
void main( ) {
    char c;
    cin>>c;
    if(isnumber( c )) cout<<"你输入了一个数字";
    else cout<<"你输入的不是一个数字";
```

}

说明：如果在程序中多次调用函数 isnumber 的话，将会大大降低使用效率，为提高效率，常将函数 main 中对函数 isnumber 的调用替换成表达式，即将 mian 函数中语句“if(isnumber(c))”转换为“if((c>='0'&&c<='9')?1:0);”，这种替换手工来作很麻烦，可以让 C＋＋编译程序来作，方法就是将函数 isnumber 的定义前面加上关键字 inline，变为内联函数的形式即可，这样，C＋＋编译器在遇到对函数 isnumber 调用的地方都会用这个函数体替换该调用表达式。

2、使用内联函数形式时的代码：

```
#include <iostream>
using namespace std;
inline int isnumber(char c){return (c>='0'&&c<='9')?1:0;}
void main( ){ …… }
```

§3.4 函数重载和默认参数

一、函数重载

函数重载就是为同一个函数定义几个版本，从而使一个函数名具有多种功能，也称函数多态性。这些函数的多种版本之间的区别有两个，一是参数类型不同，二是参数个数不同，所以仅凭返回值不同或仅凭参数个数不同，均不能区分重载函数。在函数重载时，源代码只指明函数调用，而不说明调用具体调用哪个函数，直到程序运行时才确定调用哪个函数，编译器的这种连接方式称为动态联编或迟后联编。

例：函数重载产生多态性的例子

```
#include <iostream>
using namespace std;
double max(double,double);
int max(int,int);
char max(char,char);
int max(int,int,int);
```

```
void main( ){
    cout<<max(2.5,17.54)<<" "<<max(56,8)<<" "<<max('w','p')<<endl;
    cout<<"max(5,9,4)="<<max(5,9,4)<<" max(5,4,9)="<<max(5,4,9)<<endl;
}
double max(double m1,double m2){return (m1>m2)?m1:m2;}
int max(int m1,int m2){return (m1>m2)?m1:m2;}
char max(char m1,char m2){return (m1>m2)?m1:m2;}
int max(int m1,int m2,int m3)
{   int t=max(m1,m2);
    return max(t,m3);
}
```

C++能够自动正确调用相应函数，程序输出结果如下：

```
17.54 56 w
max(5,9,4)=9 max(5,4,9)=9
```

二、函数重载与默认参数相结合

当函数重载与默认参数相结合时,能够有效减少函数个数及形态,缩减代码规模。

例：设计一个求 4（不多于 4）个整数的和的程序

1、不使用函数重载，也不使用默认参数时的代码

```
#include <iostream>
using namespace std;
int add1(int,int);                    //声明两个整数相加的函数 add1
int add2(int,int,int);                //声明三个整数相加的函数 add2
int add3(int,int,int,int);            //声明四个整数相加的函数 add3
void main( )
{cout<<add1(1,3)<<","<<add2(1,3,5)<<","<<add3(1,3,5,7)<<endl;}
int add1(int m1,int m2){return m1+m2;}
int add2(int m1,int m2,int m3){return m1+m2+m3;}
int add3(int m1,int m2,int m3,int m4){return m1+m2+m3+m4;}
```

2、使用函数重载，但不使用默认参数时的代码

```
#include <iostream>
using namespace std;
int add(int,int);
int add(int,int,int);
int add(int,int,int,int);
void main( )
{cout<<add(1,3)<<","<<add(1,3,5)<<","<<add(1,3,5,7)<<endl;}
int add(int m1,int m2){return m1+m2;}
int add(int m1,int m2,int m3){return m1+m2+m3;}
int add(int m1,int m2,int m3,int m4){return m1+m2+m3+m4;}
```

3、即使用函数重载，又使用默认参数时的代码

```
#include <iostream>
using namespace std;
int add(int m1=0,int m2=0,int m3=0,int m4=0)
{return m1 + m2 + m3 + m4;}
void main( )
{cout<<add(1,3)<<","<<add(1,3,5)<<","<<add(1,3,5,7)<<endl;}
```

程序输出结果为：4，9，16

比较：相对程序 1 而言，程序 2 的函数名只有一个，不像程序 1，需要使用三个函数名；相对程序 2 而言，程序 3 只需定义一种函数形态，不像程序 2，需要声明和定义三种函数形态。

4、如果使用默认参数，就不能对参数个数少于默认参数个数的函数形态进行重载，只能对于多于默认参数个数的函数形态进行重载。

例如如下代码，默认参数个数为 4 个，则重载形态 1 的 add 函数时会出错，因为编译器决定不了是使用形态 1、还是形态 2 的 add 函数。

```
#include <iostream>
using namespace std;
int add(int,int,int);                                    //形态 1
int add(int m1=0,int m2=0,int m3=0,int m4=0)
```

```
{return m1 + m2 + m3 + m4;}                                    //形态 2
void main( )
{cout<<add(1,3)<<","<<add(1,3,5)<<","<<add(1,3,5,7)<<endl;}
int add(int m1,int m2,int m3){return m1+m2+m3;}
```

这段代码在执行到主函数 main 中语句“cout<<add(1,3,5)”时，因编译器不知道应该重载 add 函数的哪个形态，将会出错。

§3.5 函数模板

一、函数模板

在程序设计时并不给出相应数据的实际类型，而是将类型参数化，在实际编译时，才由编译器利用实际的类型给予实例化，使它满足需要，就像按照模板来制造新的函数一样，这样的函数就是函数模板。

将函数模板与某个具体数据类型连用，就产生了模板函数，又称这个过程为函数模板实例化。在调用函数模板时，函数参数的类型决定到底使用模板的哪个版本，即模板的参数是由函数的参数推断出来的。

C++中规定模板以关键字 template 和一个形参表开头，形参表中以 class 表示“用户定义的或固有的类型”，一般选用 T 作为标识符来标识类型参数。

1、编制求两个数据最大值的函数模板程序

```
#include <iostream>
using namespace std;
template <class T>
T max(T m1,T m2) {return (m1>m2)?m1:m2;}
void main( ){
    cout<<max(2,5)<<"/t"<<max(2.0,5.)<<"/t"
        <<max('w','a')<<"/t"<<max("ABC","ABD")<<endl;    //"/t"输出空格
}
```

程序输出结果为：5 5 w ABD

2、没有使用函数模板的复数 complex 重载函数实例

```
#include <iostream>
#include <complex>
#include <string>
using namespace std;
void printer(complex <int> );
void printer(complex <double> );
void main( ){
    int i(0);
    complex <int> num1(2,3);
    complex <double> num2(3.5,4.5);//用构造函数 complex 初始化 num2 并赋值
    printer(num1);
    printer(num2);
}
void printer(complex <int> a)
{
    string str1("real is "),str2="image is ";
    cout<<str1<<a.real( )<<','<<str2<<a.image( )<<endl;
}
void printer(complex <double> a)
{
    string str1("real is "),str2="image is ";
    cout<<str1<<a.real( )<<','<<str2<<a.image( )<<endl;
}
```

语句①，函数声明

语句②，函数定义

程序运行结果为：real ia 2,image is 3

real is 3.5,image is 4.5

3、同 2 求解问题一样，但使用类模板作为函数模板参数的实例

```
#include <iostream>
#include <complex>
```

```
#include <string>
using namespace std;
template <class T>
void printer(complex <T> a);
{
    string str1("real is "),str2="image is ";
    cout<<str1<<a.real( )<<','<<str2<<a.image( )<<endl;
}
void main( ){
    int i(0);
    complex <int> num1(2,3);
    complex <double> num2(3.5,4.5);
    printer(num1);                    //语句②
    printer(num2);                    //语句③
}
```

语句④，类模板作为函数模板参数

语句①

上面程序主要执行步骤为：

第一步，执行语句①，用同名构造函数 complex 初始化 complex 对象 num1 和 num2 并赋值，其中 num1 用整数作为参数，num2 用浮点数作为参数；

第二步，执行语句②，调用语句④，因为使用了类模板作为函数模板参数，所以程序从函数的参数为整型推断出模板参数也为整型，从而使用模板的整形数模板执行，输出结果“real ia 2,image is 3”；

第三步，执行语句③，调用语句④，因为使用了类模板作为函数模板参数，所以程序从函数的参数为浮点型推断出模板参数也为浮点型，从而使用模板的浮点数模板执行，输出结果“real is 3.5,image is 4.5”；

虽然 2 例和 3 例结果一样，但是因 3 例中 complex 的参数已经类型参数化，所以程序功能比 2 例中的函数重载强的多，例如语句“complex <float> num3(2.5,3.6); printer(num3);”，2 例就不能执行，而 3 例中因参数已经类型参数化，就可以执行。

同时，就精简函数代码而言，3 例中只使用了语句④，便不仅达到而且超过 2 例中使用语句①和语句②的效果，使得函数代码规模变小，效率和功能提高。

二、函数模板的参数及显式规则

例如语句“cout<<max<int> (2,5)<<endl;”，属于显式的给出了模板参数的比较准则，函数模板参数的显式规则主要用于特殊场合。

显示比较准则形式为：函数模板名<模板参数> （参数列表）

每次调用都显式的给出比较准则的话，会使人感到很厌烦，一般可使用下面的默认方式：函数模板名（参数列表），但这样的前提是这个调用的函数参数列表能够唯一的标识出模板参数所属的集合，否则，仍要显式的给出比较准则。

如语句“cout<<max (6.5, 8)<<endl;”，在没有定义 max(double,int)形式的情况下，语句是无法执行的，这时就要显式的给出比较准则“cout<<max<double> (6.5,8)<<endl;”，或者对参数表中的参数进行强制转换，如“cout<<max (6.5,（double）8)<<endl;”，以便正确的由参数列表推断出模板参数。

三、关键字 typename 和使用显式规则

C＋＋中，关键字 typename 仅用于模板中，其用途之一就是代替类模板 template 参数列表中的关键字 class。

例：使用关键字 typename 和显式规则的实例

```
#include <iostream>
using namespace std;
template<typename T>                          //使用 typename 代替 class
T max(T m1,T m2) {return (m1>m2)?m1:m2;}      //求二者最大值
template <typename T>                         //必须重写
T min(T m1,T m2) {return (m1<m2)?m1:m2;}      //求二者最小值
Void main( ) {
    cout<<max("ABC","ABD")<<","<<min("ABC","ABD")<<","
        <<min ('W','T')<<","<<min(2.0,5.); //输出 ABD, ABC, T, 2
```

```
    cout<<min<double>(8.5,6)<<","<<min(8.5,(double)6)
        <<","<<max((int)8.5,6);  //输出 6，6，8
    cout<<min<int>(2.3,5.8)<<","<<max<int>('a','y')
        <<","<<max<char>(95,121)<<endl;  //输出 2，y，121
}
```

此程序需要注意的地方主要是：

第一，定义函数模板时，函数模板一定要包含语句“template <typename T>”，不能因为前面有过该语句，在定义后面的函数模板时就将该句省略；

第二，如果参数列表中的参数类型不同，或参数类型相同，但需要转化参数类型时，一定要显式的给出比较准则，如“min<double>(8.5,6)”、“min<int>(2.3,5.8)”、“max<int>('a','y')”、“max<char>(95,121)”，或者对参数表中的参数进行强制转换，如“cout<<max (8.5,（double） 6)”、“max((int)8.5,6)”，以便程序能够正确的由参数列表推断出模板参数。

第四章　类和对象

§4.1　类及其实例化

将一组对象的共同特征抽象出来，就形成了类的概念，如从直角坐标系的点抽象出点的概念，这就是 point 类，point 类只是表示类名是 point，它的点位置有两个坐标属性，只有当产生一个具体的坐标点 A，A 的两个属性具有了具体的值，才能代表实际的点，即实例化了。

一、类的定义

类是对一组性质相同的对象的程序描述，也属于一种用户自己构造的数据类型，也要遵循 C＋＋的规定，如先声明后使用、是具有唯一标识符的实体、在类中声明的任何成员不能使用 extern、auto 和 register 关键字修饰、类中声明的变量属于该类等。

与其他数据类型不同的是，类除了数据，还可以有对数据进行操作的函数，分别称为类的数据成员和成员函数，而且不能在类的声明中对数据成员使用表达式进行初始化。

1、声明类

C＋＋中声明类的一般形式为：

```
class 类名{
        private:
                私有数据和函数
        public:
                公有数据和函数
        protected:
                保护数据和函数
};
```

类的声明以关键字 class 开始，其后跟类名，类所声明的内容用花括号括起来，称为类体，右花括号后的分号作为类的声明语句的结束标志。

类中定义的数据和函数称为这个类的成员，无论是数据成员还是成员函数，

均具有一个访问权限，分别是私有、公有和保护，通过前加关键字 private、public 和 protected 来定义，如无关键字，则所有成员默认声明为 private 权限。

2、定义成员函数

成员函数在声明后还必须在程序中通过定义来实现对数据成员的操作。

定义成员函数的一般形式为：

返回类型 类名:: 成员函数名（形参列表）

{

成员函数的函数体　　//内部实现

}

其中返回类型是指这个成员函数返回值的类型，类名是指成员函数所属类的名字，“::”是作用域运算符，用于表示其后的成员函数属于这个特定的类。

如：void Point:: Setxy(int a, int b)

{x＝a; y＝b; }

语句表示定义属于 Point 类的成员函数 Setxy，该成员函数带有两个整型形参，函数没有返回值。

可以使用关键字 inline 将成员函数定义为**内联函数**，如：inline int Point:: Getx（）

{return x; }

如果在声明类的同时，在类体内给出成员函数的定义，也默认为内联函数，例如将声明 Getx 的语句“int Getx（）;”改为“int Getx（）{return x; }”，则 Getx 为内联函数。

3、数据成员的赋值和初始化

赋值和初始化是两个不同的概念。

赋值是在有了对象 A 之后，对象 A 调用成员函数 Setxy 实现的，如 A.Setxy（25, 56）;，不能在类体内或类体外面给数据成员赋值，例如用语句“int x=25,y=56;”给类的数据成员赋值，那么不论是在类体内还是类体外，都是错误

的，类的数据成员的赋值，除通过例如语句"B=A;"实现，就只能通过调用类的赋值成员函数或构造函数才能实现。

初始化是指产生对象时就使对象的数据成员具有指定值,它是通过使用与类同名的构造函数实现的，如在类体中声明构造函数 Point（int，int）后，在主函数中利用构造函数产生对象 B 并初始化：Point B（15，25）。

二、使用类的对象

例如对 Point 类，使用 Point 在程序中声明变量，具有 Point 类的类型的变量称为 Point 的对象,只有产生类的对象后,才能使用类的数据成员和成员函数。

类 Point 不仅可以声明对象，还可以定义对象的引用和对象的指针，其中对象和引用都使用运算符“.”访问对象的数据成员和成员函数，指针则使用“->”运算符，语法如下：

```
Point A, B;     //定义 Point 类型的对象 A 和 B
Point *p=&A;    //定义指向对象 A 的 Point 类型的指针 p
Point &R=B;     //定义 R 为 Point 类型对象 B 的引用
A.Setxy (25, 88);  //为对象 A 赋初值
R.Display ();   //显示对象 B 的数据成员之值
p->Display ();  //显示指针 p 所指对象的数据成员之值
```

例：使用内联函数定义 Point 类及使用 Point 类指针和引用

```
#include <iostream>
using namespace std;
class Point {                  //使用内联函数定义类 point
private:
   int x,y;                    //私有数据成员
public:
   void Setxy(int a,int b)  {x=a;y=b;}
   void Move(int a,int b)   {x=x+a;y=y+b;}
   void Display( )  {cout<<x<<","<<y<<endl;}
   int Getx( ) {return x;}
   int Gety( ) {return y;}
```

无返回值的内联公有成员函数

返回值类型为 int 的内联公有成员函数

```
};          //类定义以分号结束
void print(Point *a){a->Display( );}
void print(Point&a){a.Display( );}
```

分别以类指针和类引用作为print函数的参数定义重载函数

```
void main( ){
   Point A,B,*p;  //声明对象和指针
   Point &RA=A;  //声明对象 RA 为对象 A 的引用
   A.Setxy(25,55);              //使用成员函数为对象 A 赋值
   B=A;          //例如通过 int x＝25, y＝55; 对类的私有数据赋值是错误的
   p=&B;
   p->Setxy(112,115);  //使用指针调用函数 setxy 重设 B 的值
   print (p);          //传递指针显示对象 B 的属性
   p->Display();       //使用指针调用 display 函数显示对象 B 的属性
   RA.Move(-80,23);
   print(A);           //使用对象和对象指针的效果一样
   print(RA);
}
```

程序运行结果为：

112，115

112，115

－55，78

－55，78

如果在上例中的 print 函数或主函数使用如下语句，将产生错误：cout<<A.x<<","<<A.y<<endl;　　这是因为 x 和 y 都是对象 A 的私有数据成员，不能在类的外面直接使用它们，而只能通过类中的公有成员函数来使用它们。

通过上例，可归纳出有关类的一些规律如下：

①类的成员函数可以直接使用自己类的私有成员（数据成员和成员函数）；

②类外面的函数不能直接访问类的私有成员，而只能通过类的对象使用该类的公有成员函数；

③对象 A 和 B 的成员函数的代码一样，两个对象的区别是属性的取值。

三、数据封装

面向对象的程序设计是通过为数据和代码建立分块的内存区域，以便提供对程序进行模块化的一种程序设计方法，这些模块可以被用作样板，在需要时再建立副本。由此，对象是计算机内存中的一块区域，通过将内存分块，每个模块在功能上保持相对独立。

面向对象是消息处理机制，对象之间只能通过成员函数调用实现相互通信。当对象的一个函数被调用时，对象执行其内部的代码来响应这个调用，使对象呈现出一定的行为，对象被视为能作出动作的实体，对象使用这些动作完成相互之间的作用。

C＋＋通过类实现数据封装，即通过指定各成员的访问权限来实现。一般情况下将数据说明为私有的，以便隐藏数据；而将部分成员函数说明为公有的，用于提供外界和这个类的对象相互作用的接口（界面），从而使其他函数也可以访问和处理该类的对象。

公用的成员函数是外界所能观察到（访问到）的对象界面，它们所表达的功能构成对象的功能，使同一个对象的功能能够在不同的软件系统中保持不变，这样当数据结构发生变化时，只需要修改少量的类的成员函数的实现代码，就可以保证对象的功能不变。只要对象的功能保持不变，则公有的成员函数所形成的接口就不会发生变化，这样，对象内部实现所作的修改就不会影响使用该对象的软件系统，这就是数据封装的益处。

§4.2　构造函数

构造函数是用来实现对象初始化的特殊成员函数。

一、默认构造函数

当没有为一个类定义任何构造函数的情况下，C＋＋编译器会自动建立一个不带参数的、函数体为空的构造函数，以 Point 类为例，默认构造函数的形式为：

```
Point:: Point（）{ }
```

一旦程序定义了自己的构造函数，系统就不再提供默认构造函数，这时如果没有再定义一个无参数的构造函数，但又声明了一个没有初始化的对象（如

Point A;)，则因系统已不再提供默认构造函数而造成编译错误。同理，如果程序中定义了有参数的构造函数,又存在需要先建立对象数组后进行赋值操作的情况,则因每个元素对象均需调用一次无参数构造函数来为自己初始化,此时就要求必须为数组定义一个无参数的默认构造函数。

二、定义构造函数

1、构造函数的声明与定义

为了提高安全性和效率，构造函数的名字必须和类名相同，并在定义构造函数时不能指定返回类型，即不要返回值，即使是 void 类型也不可以，另外类可有多个构造函数。

①构造函数在类体里的声明形式：

类名（形参 1，形参 2，…形参 n）；　//也可没有形参

②构造函数的定义形式：

假设数据成员为 x1、x2，则有以下两种形式：

类名:: 类名（形参 1，形参 2）: x1（形参 1），x2（形参 2）{ }

```
类名:: 类名（形参 1，形参 2）
  {
    x1＝形参 1;
    x2＝形参 2;
    }
```

构造函数的参数在排列时无顺序要求，只要保证相互对应即可，可以使用默认参数或重载。在程序中说明一个对象时，程序自动调用构造函数来初始化该对象。

2、构造函数的使用方法：自动调用

构造函数不能在程序中显式调用，它是由系统自动调用的。例如构造 Point 类的对象 a，不能写成“Point a.Point（x，y);”，只能写成“Point a（x，y)”。编译系统会自动调用构造函数 Point（x，y）产生对象 a 并使用 x 和 y 将其正确的初始化。

在产生对象 a 之后，也不能使用 a.Point（x，y）改变对象的属性值，而是

要设计专门的成员函数如 Setxy 函数改变对象的属性值。

可以设计多个构造函数,编译系统根据对象产生的方法自动调用相应的构造函数，构造函数将在产生对象的同时初始化对象。

三、构造函数和运算符 new

运算符 new 用于建立生存期可控的动态对象，new 返回这个对象的指针，语法以 Point 类为例，见下面例子。

New 和构造函数一同起作用，当使用 new 建立一个动态对象时，new 首先分配足以保存 Point 类的一个对象所需要的内存,然后自动调用构造函数来初始化这块内存，再返回这个动态对象的地址。

使用运算符 new 建立的动态对象只能用运算符 delete 删除，所以使用完毕后，一定要记得使用 delete 释放空间。

例：构造函数的定义、使用过程及运算符 new 的使用方法

```
#include <iostream>
using namespace std;
class Point {
private:
   int x,y;
public:
   Point( );  //声明不带参数的构造函数
   Point(int,int);  //声明带两个参数的构造函数
};
Point::Point( ):x(0),y(0)  {cout<<"Initializing default"<<endl;}
                                   //定义不带参数的构造函数
Point::Point(int a,int b):x(a),y(b)  //定义带两个参数的构造函数
{cout<<"Initializing"<<a<<","<<b<<endl;}
void main( ){
   Point A;  //使用不带参数的构造函数产生对象 A
```

```
    Point B(15,25);  //使用带参数的构造函数产生对象 B
    Point C[2];  //使用不带参数的构造函数产生对象数组 C
    Point D[2]={Point(5,7),Point(8,12)};  //使用带参数的构造函数产生对象数组 D
    Point *ptr1=new Point;  //使用不带参数的构造函数产生动态对象 ptr1
    Point *ptr2=new Point(5,7);  //使用带参数的构造函数产生动态对象 ptr2
    delete ptr1;  //删除动态对象 ptr1，释放内存空间
    delete ptr2;  //删除动态对象 ptr1，释放内存空间
}
```

程序输出结果为：

```
Initializing default
Initializing 15, 25
Initializing default
Initializing default
Initializing 5, 7
Initializing 8, 12
Initializing default
Initializing 5, 7
```

四、构造函数的默认参数

如果定义了有参数的构造函数，同时又想使用无参数的构造函数，可以通过将相应的构造函数全部使用默认参数设计来实现。

例：设计构造函数的默认参数

```
#include <iostream>
using namespace std;
class Point {
private:
    int x,y;
```

```
public:
    Point(int=0,int=0);  //声明两个参数均为默认参数
};
Point::Point(int a,int b):x(a),y(b)  //定义构造函数
{cout<<"Initializing"<<a<<","<<b<<endl;}
void main( ){
    Point A;           //构造函数产生对象 A
    Point B(15,25);    //构造函数产生对象 B
    Point C[2];        //构造函数产生对象数组 C
}
```

程序输出结果如下：

```
Initializing 0, 0
Initializing 15, 25
Initializing 0, 0
Initializing 0, 0
```

五、复制构造函数

复制构造函数的作用，就是通过拷贝方式使用一个类已有的对象来建立该类的一个新对象。通常情况下，由编译器建立一个默认复制构造函数，其原形为：类名:: 类名（const 类名&），或不加 const 修饰，但不论使用哪种形式的原型声明，为了提高程序的效率和安全性，都必须使用对象的引用作为形参。

如构造函数一样，复制构造函数也可自定义，如果自定义了，则编译器只调用自定义的复制构造函数，而不再调用默认的复制构造函数。

以 Point 类为例，复制构造函数的原型声明为：Point（Point&）；　　它的定义如下：Point:: Point（Point& t）｛x＝t.x; y＝t.y;｝

§4.3　析构函数

析构函数和构造函数、复制构造函数都是构造型成员函数的基本成员，它的作用是在对象消失时，释放由构造函数分配的内存。

一、定义析构函数

和构造函数一样，为了提高安全性和效率，析构函数的名字也必须和类名相同，只是在析构函数前加一个“～”号，并且在定义析构函数时也不能指定返回类型，即使是void类型也不可以，同时析构函数也不能指定参数，但可以显式的说明参数为void，如A::～A(void)，不同的是，类只能定义一个析构函数，且不能指明参数，以便编译系统在对象的生存期结束时自动调用

①析构函数在类体里的声明形式：

～类名（）;

②析构函数的定义形式：

类名::～类名（）{ }

析构函数在对象的生存期结束时被编译系统自动调用，然后对象占用的内存被回收。全局对象和静态对象的析构函数在程序运行结束之前调用。类的对象数组的每个元素调用一次析构函数，全局对象数组的析构函数在程序结束之前被调用。

二、析构函数和运算符delete

运算符delete与析构函数一起工作，当使用运算符delete删除一个动态对象时，它首先为这个对象调用析构函数，然后再释放这个动态对象占用的内存，这和使用运算符new建立动态对象的过程刚好相反。

例：使用Point类建立和释放一个动态对象数组

```
#include <iostream>
using namespace std;
class Point {
private:
   int x,y;
public:
   Point(int=0,int=0);  //声明两个参数均为默认参数
   ～Point ();  //声明析构函数
```

```
};
Point::Point(int a,int b):x(a),y(b)  //定义构造函数
{cout<<"Initializing"<<a<<","<<b<<endl;}
Point:: ~Point ()  {cout<<"Destructor is active"<<endl;}
                                    //定义析构函数
void main( ){
   Point *ptr=new Point[2];
   delete [ ]ptr;
}
```

程序输出结果为:

```
Initializing 0, 0
Initializing 0, 0
Destructor is active
Destructor is active
```

表达式 new Point[2]首先分配 2 个 Point 类的对象所需的内存，然后分别为这 2 个对象调用一次构造函数。当使用 delete 释放动态对象数组时，用语句 delete [ ]ptr;　　使运算符 delete 知道 ptr 指向的是动态对象数组，delete 将为动态对象数组的每个对象调用一次析构函数，然后释放 ptr 所指向的内存。

当程序先后创建几个对象时，系统按照后建先析构的原则析构对象，当使用 delete 调用析构函数时，则按 delete 的顺序析构。

三、默认析构函数

如果在定义类时没有定义析构函数,则编译器将自动为类产生一个函数体为空的默认析构函数，以 Point 类为例，默认析构函数形式如下:

```
Point:: ~Point () { }
```

§4.4　调用复制构造函数的综合实例

一、程序代码

```
#include <iostream>
using namespace std;
class Point {
    private:
        int X,Y;
    public:
        Point(int a=0,int b=0){X=a;Y=b;cout<<"初始化中"<<endl;}
            //语句①,定义有默认参数的构造函数(且为内联公有成员函数)
        Point(const Point &p);        //语句②, 声明复制构造函数
        int GetX( ){return X;}
        int GetY( ){return Y;}        定义内联公有成员函数
        Void Show( ){cout<<"X="<<X<<",Y="<<Y<<endl;}  //语句③
        ~Point( ){cout<<"删除…"<<X<<","<<Y<<endl;}  //语句④
};
Point::Point(const Point &p){X=p.X;Y=p.Y;Cout<<"拷贝初始化中"<<endl;}
      //语句⑤, 定义必须使用对象的引用做形参的复制构造函数
void display(Point p){p.show( );}     //语句⑥, 点类对象做函数形参
void disp(Point &p){p.show( );}     //语句⑦, 点类对象的引用做函数形参
Point fun( ){Point A(101,202);return A;}  //语句⑧, 函数返回值为点类对象
void main( ){
    Point A(42,35);     //代码①, 定义点类对象 A 并赋值
    Point B(A);         //代码②, 定义点类对象 B, 调用复制构造函数用 A 初始化 B
    Point C(58,94);     //代码③, 定义点类对象 C 并赋值
    Cout<<"called display(B)"<<endl;
    Display(B);                          //代码④
    Cout<<"下一个…"<<endl;
```

```
    Cout<<"called disp(B)"<<endl;
    Disp(B);                                    //代码⑤
    Cout<<"call C = fun( )"<<endl;
    C=fun( );                                   //代码⑥
    Cout<<"called disp( C )"<<endl;
    Disp(C);                                    //代码⑦
    Cout<<"out…"<<endl;
}
```

二、上面程序的主要执行步骤：

第一，执行代码①，调用语句①，创建点类对象 A 并初始化，然后将实参 42、35 的值分别赋给形参 a、b，最后执行函数体内语句，将 a、b 的值分别赋给点 A 的私有数据成员 X、Y，并输出“初始化中”;（在这个构造函数的实现中，它访问并修改了对象 A 的私有数据成员 X、Y，这是允许的，因为类的成员函数可以直接使用自己类的私有成员，包括数据成员和成员函数）

第二，执行代码②，调用语句⑤，创建点类对象 C，然后将实参 A 的地址值传递给形参 p（采取传引用的方式时，虽然系统向形参传递的是实参的地址，但实参必须使用对象名，形象的说，就是把实参的名传给了形参），p 作为 A 的引用，最后执行函数体内语句，将 p.x 和 p.Y 的值分别赋给点 C 的私有数据成员 X、Y，并输出“拷贝初始化中”;（在这个复制构造函数的实现中，它以 A 的引用 p 的形式，访问了对象 A 的数据成员 A.X 和 A.Y，访问并修改了对象 C 的私有数据成员 X、Y，这是允许的，因为类的成员函数可以直接使用自己类的私有成员，包括数据成员和成员函数）

第三，执行代码③，调用语句①，具体过程分析同第一步，最后输出“初始化中”;

第四，执行代码④，先后调用语句⑥、语句⑤、语句③、语句④，先后输出“called display(B)”、“拷贝初始化中”、“X＝42，Y＝35”、“删除…42，35”、“下一个…”;这是因为函数 display 的形参是类的对象，采取的是传值方式，在把实参的值传递给形参的时候，将产生一个副本，即临时对象，这就需要调用复制构造函数来产生副本，然后才进入函数体内执行语句，最后在退出函数时，又需要调用析构函数清除临时对象;

第五，执行代码⑤，调用语句⑦，先后输出“called disp（B)”、“X＝42，Y＝35”;比起第四步,输出内容少了很多,这是因为函数 disp 的形参是对象的引用,采用的是传地址的方式，进行实参和形参结合时，不产生临时对象，所以不需调用复制构造函数和析构函数，而是直接进入函数体内执行语句即可;

第六，执行代码⑥，先输出“call c＝fun（)”，然后调用语句⑧，调用的过程中先后执行：调用语句①，创建对象 A，输出“初始化中”，此时 A 的生命期与函数 fun 同在；调用语句⑤，产生临时对象，输出“拷贝初始化中”；由临时对象完成将函数返回值 A 赋给 C，然后调用语句④，析构这个临时对象，输出“删除…101，202”；调用语句④，析构对象 A，输出“删除…101，202”;

第七，执行代码⑦，调用语句⑦，先后输出“called disp（C)”、“X＝101，Y＝202”、“out…”，具体过程分析同第五步;

第八,按照对象定义的顺序序析构,即先调用语句④,析构对象 C,输出“删除…101，202”；然后调用语句④，析构对象 B，输出“删除…42，35”；最后调用语句④，析构对象 A，输出“删除…42，35”。

三、调用复制构造函数的三种情况的小结

1、当用一个类的对象去初始化另一个对象时，需要调用复制构造函数。如例中：

```
Point A（14，25）;   //调用构造函数初始化对象 A
Point B（A）;   //用对象 A 初始化对象 B，调用复制构造函数
```

2、如果函数的形参是类的对象，调用函数时，进行形参与实参的结合时，将产生临时对象，需要调用复制构造函数和析构函数，如例中：

```
display（B）;   //对象 B 作为 display（）的实参，调用复制构造函数
```

因为函数 display（Point p）是以对象作为形参，采用的是传对象的传值方式，所以需要产生一个副本，这就是临时对象，调用复制构造函数产生副本。在退出函数时，系统再自动调用析构函数析构临时对象。

而对于函数 disp（Point&）而言，是以对象的引用作为形参，采用的是传引用的方式，所以不产生副本，即程序不需调用复制构造函数产生临时对象，而是直接进入 disp（）的函数体执行语句。因没有临时对象产生，所以当退出 disp（）函数时，也不需要调用析构函数来析构临时对象。

由此可见，函数参数使用对象的引用不会产生副本，从而不需调用复制构造函数和析构函数，方便安全，这也是推荐使用对象的引用而不使用对象的原因。

3、如果函数的返回值是对象，当函数调用完成返回时，需要调用复制构造函数，产生临时对象，并在执行完返回值赋值语句后，析构临时对象和对象。如例中： C＝fun（）； //fun 的返回值赋给对象 C，调用复制构造函数

程序执行过程为：调用函数 fun（）{Point A（101，202）； return A；}，在其内调用构造函数 Point（int，int）创建对象 A，然后执行 return A；目的是使 C＝A，但这个赋值过程需要调用复制构造函数创建一个临时对象，用这个临时对象完成 C＝A，然后再析构这个临时对象，析构临时对象后，结束 fun（）的函数调用，退出 fun（）的函数体，此时再调用析构函数析构对象 A。

4、上述实例的输出结果证实了调用构造函数和析构函数是按相反的顺序执行的，如果调用的函数程序里也产生了对象，则在函数程序里也遵循这一规律。

§4.5 成员函数重载及默认参数

成员函数可重载或使用默认参数，下面例子就演示使用私有成员函数封装函数，并使用成员函数重载和默认参数。

例：构造一个求 4 个正整数中最大者的类 Max，并在主程序中验证其功能

```
#include <iostream>
using namespace std;
class Max{  //声明类
private:  //封装数据成员和成员函数
    int a,b,c,d;  //数据成员
    int Maxi(int,int);  //声明 Maxi 只允许类内部的成员函数调用
public:  //对外部的接口
    void Set(int,int,int,int);  //公有成员函数 Set 的原型声明
    int Maxi( );  //求最大值
}A[3];  //声明类的对象数组，作用与主程序中语句“Max A[3];”相同
//类中成员函数的实现
```

```
int Max::Maxi(int x,int y) {return(x>y)?x:y;}  //求两个数的最大值
void Max::Set(int x1,int x2,int x3=0,int x4=0) {a=x1;b=x2;c=x3;d=x4;}
                                                //使用两个默认参数
int Max::Maxi( ){int x=Maxi(a,b);int y=Maxi(c,d);return Maxi(x,y);}
```

//Maxi 函数重载，用于求自己类中四个数的最大值，x 和 y 为 Maxi（）函数的局部整数对象，如 x 和 y，C＋＋中允许变量便使用便定义。

```
void main(){
    A[0].Set(12,45,76,89);  //为数组对象 A[0]置初值
    A[1].Set(12,45,76);  //为数组对象 A[1]置初值
    A[2].Set(12,45);  //为数组对象 A[2]置初值
    for(int i=0;i<3;i++)
        cout<<A[i].Maxi( )<<" ";  //输出对象求值结果
}
```

程序输出结果为：89 76 45

小结：

①程序演示了可在声明类的同时也声明类的对象，这里是声明对象数组 A，作用与在主程序里使用语句“Max A[3];”相同。当然，为了提高可读性，一般不在声明类时声明对象。

②Max 类中重载了成员函数 Maxi，其中一个原型为 Maxi（ini，ini），用来求两数中的大者；另一个原型为 Maxi（），它调用两次 Maxi（ini，ini），然后再用这两次的结果作为 Maxi（ini，ini）的参数，求出四个数中的最大值。

§4.6 this 指针

一、this 指针的概念和作用

C＋＋规定，当一个成员函数被调用时，系统将自动向它传递一个隐含的参数，该参数是一个指向调用该函数的对象的指针，名为 this 指针，从而使成员

函数知道该对哪个对象进行操作。

使用 this 指针，保证了每个对象可以拥有自己的数据成员，但处理这些数据成员的代码却可以被所有的对象共享，从而提高了程序的安全性和效率。

this 指针是 C＋＋实现封装的一种机制，它将对象和该对象调用的成员函数连接在一起，从而在外部看来，每个对象都拥有自己的成员函数。

二、this 指针的实际形式

例如当执行 A.Setxy（25，55）时，成员函数 Setxy（int a，int b）实际上是如下形式：

```
void Point:: Setxy（int a，int b，(Point * ) this)
{    this->x＝a;
     this->y＝b;
}              //此时成员函数的 this 指针指向对象 A。
```

但是，一般情况下，即使在定义 Setxy 函数时使用指针，也不要给出隐含参数（Point *）this，写成下面的形式即可：

```
void Point:: Setxy（int a，int b）
{    this->x＝a;
     this->y＝b;
}
```

除非特殊需要，一般情况下都省略掉符号“this->”，而让系统进行默认设置。

§4.7　一个类的对象作为另一个类的成员的实例

因为类本身就是一种新的数据类型，所以一个类的对象可以作为另一个类的成员。例如有 A、B 两个类，可以通过在 B 类里定义 A 的对象作为 B 的数据成员，或者定义一个返回类型为 A 的函数作为 B 的成员函数。

例：使用 Point 类的对象及函数作为 Rectangle 类的数据成员及成员函数

一、程序代码

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

```
class Point{                              //定义点类 Point
        int x,y;                          //没有说明的，默认性质是 private
    public:
        void Set(int a,int b){x=a;y=b;}  //语句⑤，定义内联的公有成员函数
        int Getx( ){return x;}            //语句⑥，定义内联的公有成员函数
        int Gety( ){return y;}            //语句⑦，定义内联的公有成员函数
};
class Rectangle{                                  //定义矩形类 Rectangle
        Point Loc;                                //注释 1
        int H,W;                                  //定义矩形类的高 H 和宽 W
    public:
        void Set(int x,int y,int h,int w);
        Point * GetLoc( );      //声明返回 Point 类指针的成员函数 GetLoc
        int GetHeight( ){return H;}          //定义内联的公有成员函数
        int GetWidth( ){return W;}           //定义内联的公有成员函数
};
void Rectangle::Set(int x,int y,int h,int w){Loc.Set(x,y); H=h;W=w;}
                                                    //语句②，注释 2
Point * Rectangle::GetLoc( ){return &Loc;}          //注释 3
void main( ){
    Rectangle rect;               //定义 Rectangle 类的对象 rect
    rect.Set(10,2,25,20);         //语句①
    cout<<rect.GetHeight( )<<","<<rect.GetWidth( )<<",";
                                  //语句⑧，输出"25,20,"
    Point * p=rect.GetLoc( );                       //语句③，注释 4
    cout<<p->Getx( )<<","<<p->Gety( )<<endl;     //语句④，输出"10,2"
}
```

二、程序中主要语句的注释：

注释 1：定义 Point 类的对象 Loc 为 Rectangle 类的私有数据成员（如果类

的数据成员或成员函数没有用 private 说明，则默认性质为 private)，此时 Loc 充当的是矩形的顶点；

注释 2：定义具有四个形参的 Rectangle 类的公有成员函数 Set，该函数先利用 Point 类的对象 Loc 调用 Point 类的成员函数 Set(x,y)实现坐标顶点初始化并赋值，然后给矩形的高 H 和宽 W 赋值；

注释 3：定义返回值类型为 Point 类指针的 Rectangle 类的公有成员函数 GetLoc,该函数返回 Point 类对象 Loc 的首地址；

注释 4：定义 Point 类的指针对象 p，并用 Rectangle 类的公有成员函数 GetLoc 的返回值（Point 类对象 Loc 的首地址）给该指针赋值。

三、程序的主要执行步骤

第一，声明类 Point，该类具有两个整型私有数据成员 x、y；具有一个无返回值的公有成员函数 Set 和两个返回值类型为整型的公有成员函数 Getx、Gety；

第二，声明类 Rectangle，该类具有三个私有数据成员，其中 Loc 是 Point 类的对象作为 Rectangle 类的私有数据成员，H、W 为整型数；具有四个公有成员函数，其中 Set 无返回值，GetLoc 返回值声明为 Point 类的指针，GetHeight 和 GetWidth 为内联的公有成员函数，返回值为整形数 H、W；

第三，定义 Rectangle 类的公有成员函数 Set，该函数具有四个形参，函数体内语句执行过程为先利用 Point 类的对象 Loc 调用 Point 类的成员函数 Set(x,y)实现坐标顶点初始化并赋值，然后给矩形的高 H 和宽 W 赋值；

第四,定义 Rectangle 类的公有成员函数 GetLoc,该函数返回值类型为 Point 类的指针，函数体内语句执行后返回 Point 类的对象 Loc 的首地址；

第五，执行语句①，调用语句② ，Rectangle 类的对象 rect 执行 Rectangle 类的成员函数 Set，函数体内语句执行过程为：首先，因为 Rectangle 类的私有数据成员 Loc 为 Point 类的对象，所以它可以调用 Point 类的公有成员函数 Set 给形参 a、b 赋值（此时 a、b 的值分别变为 10、2)；然后，再执行 Point 类的成员函数 Set 的函数体内语句，将形参 a、b 的值分别赋给 Loc 的私有数据成员 x、y（这是因为 Point 类的成员函数 Set 是可以直接访问并修改 Point 类的对象 Loc 的私有数据成员的，此时 x、y 的值分别变为 10、2，Point 类的对象 Loc 的值确定为（10，2))；最后，将实参 25、20 的值赋给 Rectangle 类的成员函数

Set 的形参 h、w，执行语句“H=h;W=w;”，将 h、w 的值分别赋给 rect 的私有数据成员 H、W;

第六，执行语句⑧，Rectangle 类的对象 rect 分别调用 Rectangle 类的成员函数 GetHeight 和 Getwidth，输出 H、W 的值 25、20;

第七,执行语句③,定义 Point 类的指针对象 p,由 Rectangle 类的对象 rect 调用 Rectangle 类的成员函数 GetLoc，将返回值 Point 类的对象 Loc 的首地址赋给 Point 类的指针对象 p;

第八，执行语句④，Point 类的指针对象 p 调用 Point 类的成员函数 Getx 和 Gety，输出 Point 类的对象 Loc 的私有数据成员 x、y 的值 10、2。

说明：例中 Rectangle 类具有一个顶点，所以也具有 Point 类的属性。因为 Rectangle 类不能直接操作 Point 类的数据，所以必须通过 Point 类的对象调用 Point 类的的成员函数来实现对 Point 类的数据的操作,如例中是通过定义 Point 类的对象 Loc 作为 Rectangle 类的数据成员，然后通过 Loc 调用 Point 类的成员函数 Set（x，y）实现的，语句为 Loc.Set（x，y)。

§4.8 类和对象的性质

一、对象的性质

1、同一类的对象之间可以相互赋值，如语句：

Point A，B;　　A.Setxy（25，55);　　B＝A;

2、可以使用对象数组，如语句：Point A[3];　定义数组 A 可以存储 3 个 Point 类的对象。

3、也可以使用指向对象的指针，使用取地址运算符&将一个对象的地址置于该指针中，如语句： Point * p＝&A;　　p->Display（);

4、对象可以用作函数参数。

如果参数传递采用传对象值的方式,会在传值过程中产生副本,即临时对象,所以在被调用函数中对形参所作的改变不影响调用函数中作为实参的对象。

如果采取传对象的引用（传地址）的方式，当形参对象被修改时，相应的实参对象也将被修改。

如果采用传对象地址值的方式，则使用对象指针作为函数参数，效果与传对象的引用一样。

C＋＋推荐使用对象的引用作为参数传递，因为这样不会产生副本（即临时对象），在程序执行时不用调用构造函数，可直接进入函数体执行语句，在函数结束后，也不用调用析构函数析构临时对象。

如想避免被调用函数修改原来对象的数据成员，可使用 const 修饰符。

5、对象作为函数参数时，可以使用对象、对象引用和对象指针三种方式。以 Point 函数为例，它们的参数形式分别为：

```
void print (Point a) {a.Display; }   //对象作为函数参数
void print (Point&a) {a.Display; }   //对象引用作为函数参数
void print (Point * p) {p->Display; }   //对象指针作为参数
```

它们的原型声明分别为：print(Point),print(Point&),print(Point *)。对于对象 A，print（&A）调用的是原型为 print（Point *）的函数形式。

另外，函数重载不能同时采用如上 3 种同名函数，因为函数使用的参数为对象或对象引用时，编译系统无法区别这两个函数，重载只能选择其中的一种。

6、一个对象可以作为另一个类的成员。如上节例中，定义 Point 类的对象 Loc 为 Rectangle 类的数据成员，定义 GetLoc（）为 Rectangle 类的返回 Point 类型指针的成员函数，当 Rectangle 类创建 rect 对象后，rect 对象就将 Point 类的对象 Loc 作为自己的一个数据成员(此例中 Loc 充当的是顶点)。

二、类的性质

1、使用类的权限

①类本身的成员函数可以使用类的所有成员（私有和公有成员）。

②类的对象只能访问公有成员函数。例如输出 x 只能使用 A.Getx（），不能使用 A.x。

③其他函数不能使用类的私有成员，也不能使用公有成员函数，它们只能通

过定义类的对象为自己的数据成员，然后通过类的对象使用类的公有成员函数。

④虽然一个类 A 可以包含另外一个类 B 的对象，但类 A 也只能通过被包含的类 B 的对象使用类 B 的成员函数，通过类 B 的成员函数使用类 B 的数据成员，如 Loc.Set（x，y)。

2、不完全的类的声明

类不是内存中的物理实体，只有当使用类产生对象时，才进行内存分配，这种对象建立的过程称为实例化。

类必须在其成员使用之前先进行声明。然而，有时也可以在类没有完全定义之前就引用该类，此时将类作为一个整体来使用，如声明全局变量指针：

```
class MembersOnly;   //不完全的类声明
MembersOnly * club;   //定义全局变量类指针
Void main（）{… 函数体}    //主函数
class MembersOnly{…函数体};   //完全定义该类
```

不完全声明的类不能实例化，否则会编译出错；不完全声明仅用于类和结构，企图存取没有完全声明的类成员，也会引起编译错误。

3、空类

尽管类的目的是封装代码和数据，它也可以不包括任何声明，如：class Empty{ }; 这种空类没有任何行为，但可以产生空类对象。

在开发大的项目时，需要在一些类还没有完全定义或实现时进行先期测试，定义空类可保证代码能正确被编译，从而允许先测试其中的一部分。此时常给空类增加一个无参数构造函数，如 class Empty{public：Empty（）{ } } ，以消除强类型检查的编译器产生的警告。

4、类作用域

①声明类时所使用的一对花括号形成类作用域，在类作用域中声明的标识符只在类中可见，如：class example { int num; } ;

```
int i=num;  //定义整型数 i，并用 num 赋值；错误，因为 num 在类外并没有定义，所以在此不可见
int num;  //重新定义整型数 num；正确，因为此 num 是重新定义的一个整
```

型数，与类中说明的数据成员 num 具有不同的作用域

②如果某成员函数的实现是在类定义之外给出的,则类作用域也包含该成员函数的作用域，因此，当在该成员函数内使用一个标识符时，编译器会先在类定义域中寻找，如下例：

```
class Myclass {
      int number;     //定义属于类 Myclass 的整型数 number
   public:
      void set(int);
};
int number;   //这个 number 不属于类 Myclass
void Myclass:: set (int i) {
number=i;   //使用的是类 Myclass 中的标识符 number
}
```

③类中的一个成员名可以通过使用类名和作用域运算符来显式的指定,称为成员名限定，例如：

```
void Myclass:: set (int i) {
Myclass:: number=i;  //显示指定访问 Myclass 类中的标识符 number
}
```

④在程序中，对象的生存期由对象说明来决定；类中各数据成员的生存期由对象的生存期决定，随对象存在和消失。

⑤使用 struct 关键字设计类与 class 相反，struct 的默认控制权限是 public，一般不用 struct 设计类，而是用 struct 设计只包含数据的结构，然后用这种结构作为类的数据成员。

§4.9 面向对象的标记图

面向对象的早期，主要完善的是面向对象实现（即 OOP 阶段）阶段，后来转向面向对象分析（OOA）和面向对象设计（OOD），OOA 和 OOD 更注重于使问题的描述更接近于真实的自然，它们采用一致的概念、原则和表示法，二者之间不存在鸿沟，不需要从分析文档到设计文档的转换。

为了设计、开发和相互交流的需要，人们采用图像形式将面向对象分析、设计和实现阶段对问题的描述直观的表示出来，称为标记图。实际使用的标记图种类很多，直到 1992 年 OMG（面向对象管理组）制定的面向对象分析和设计的国际标准 UML（统一建模语言）问世后，标记图的方法和技术才真正统一。

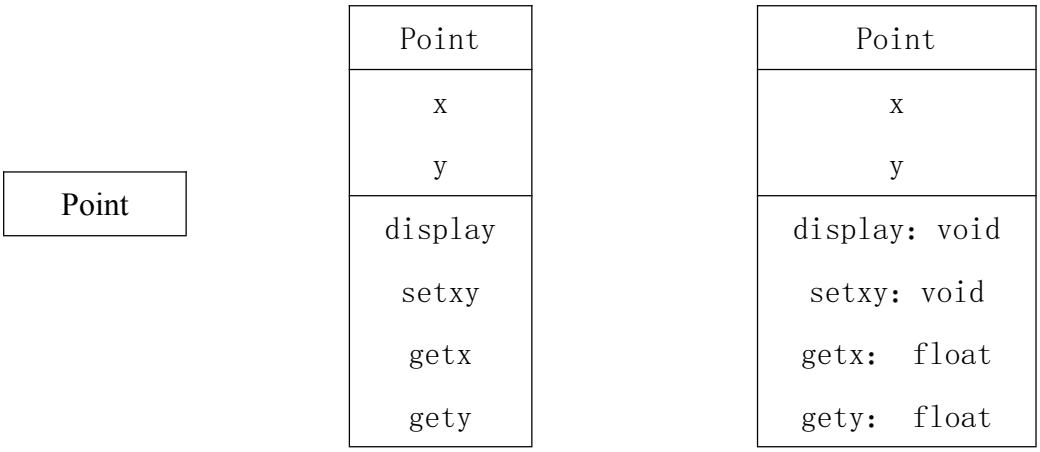
UML 是一种可视化建模语言，主要用于面向对象分析和建模。

一、类和对象的 UML 标记图

类和对象的标记符号都只能表示静态特征。

1、类的标记图

在 UML 语言中，类使用短式和长式两种方式表示。短式仅用 1 个含有类名的长方框表示，长式使用 3 个方框表示，最上面是类名，中间框填入属性（数据成员），最下面填入操作（成员函数），属性和操作可以根据需要进行细化。



短式　　长式　　进一步细化的长式

2、对象的标记图

对象的表示有 3 种方式，最简单的是只填写对象名称；完整的方式是在对象名的右边用冒号连接类名，并用下划线将它们标注出来；另外一种就是在还没有决定这个对象的名称时，可以不给出对象名，但不能省去冒号和类名。

A
x＝1.5 y＝5.4
display setxy getx gety

简单

B: Point
x＝3.5 y＝6.9
display setxy getx gety

完整

: Point
x＝1.5 y＝5.4
display setxy getx gety

未定

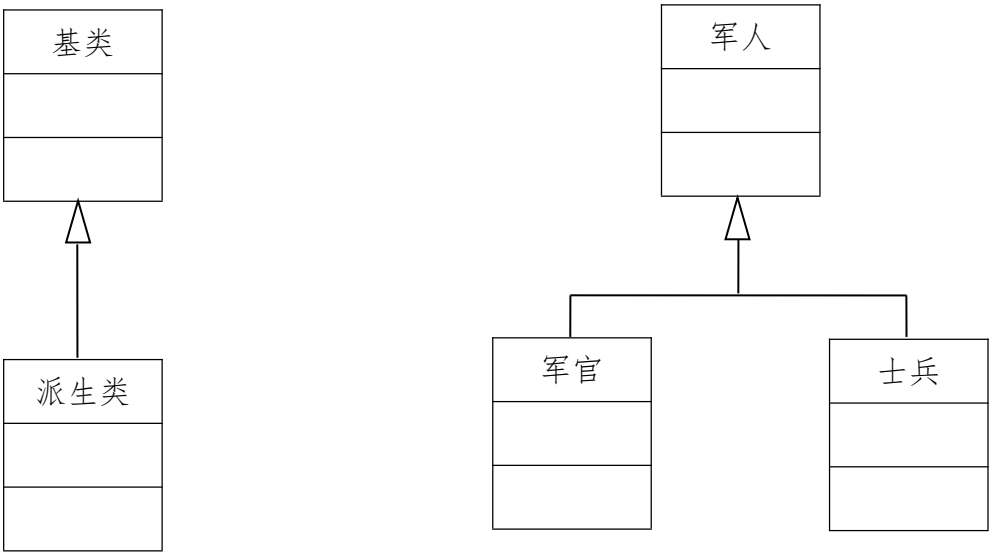
二、对象的结构与连接

对象的结构是指对象之间的分类（继承）关系和组成（聚合）关系，统称关联关系。

对象的连接是指实例连接和消息连接，其中，对象之间的静态关系是通过对象属性之间的连接反映的，称为实例连接；对象行为之间的动态关系是通过对象行为（消息）之间的依赖关系表现的，称为消息连接。

1、分类关系及其表示

C＋＋中的分类结构是指继承（基类/派生类）结构，UML 使用一个空三角形表示继承关系，三角形指向基类。



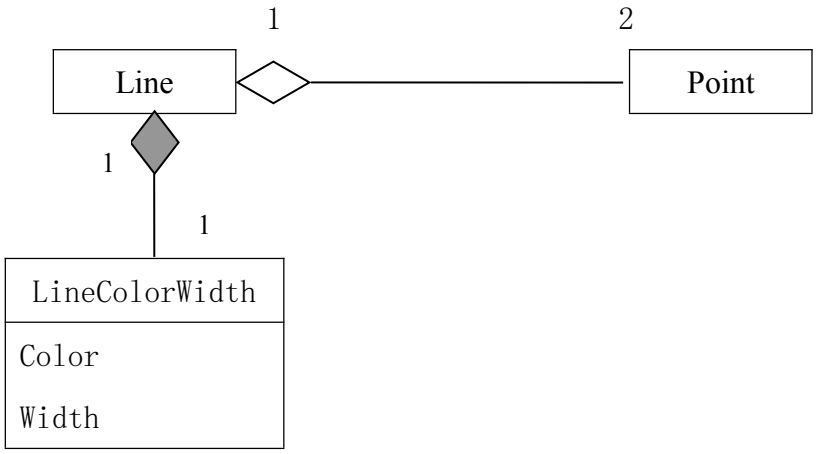
2、组合关系及其表示

组合关系说明的是整体与部分的关系，C＋＋中最简单的是包含关系，如线段由两个点组成。

C＋＋中的聚合有两种实现方式，一种是独立的定义，可以属于多个整体对象，并有不同的生存期，例如一个法律顾问可以属于几个单位，这种所属关系是可以动态变化的，称为聚集，用空心菱形表示，如下图中 Line 对象由两个 Point 对象组成。

另一种方式是用一个类的对象作为一种广义的数据类型来定义整体对象的一个属性，构成一个嵌套对象，这种情况下，这个类的对象只能隶属于唯一的整体对象并与它同生同灭，称这种情况为组合，使用实心菱形表示，如下图中 Line 的颜色和宽度由另外一个嵌套对象 LineColorWidth 提供，它们生命周期一样。

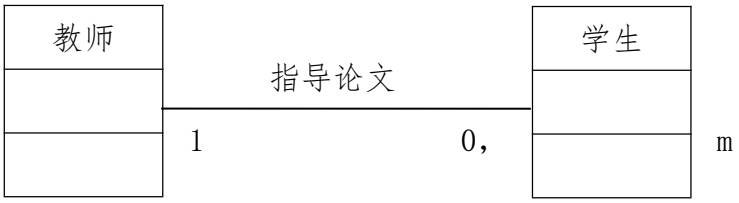
图中还给出关联时的数量关系。



3、实例连接及其表示

实例连接反映对象之间的静态关系，例如车和驾驶员的关系，实例连接有一对一、一对多和多对多 3 种连接方式，用一条实线表示连接关系。

简单的实例连接是对象实例之间的一种二元关系，如教师类的对象实例和毕业班学生类的对象实例之间有一种指导毕业论文的关系，如下图：



这种连接的意义是表明一个教师为某些学生指导毕业论文。

4、消息连接及其表示

消息连接描述对象之间的动态关系，即若一个对象在执行自己的操作时，需要通过消息请求另一个对象为它完成某种服务，则说第 1 个对象与第 2 个对象之间存在着消息连接。

消息连接是有方向的，使用一条带箭头的实线表示，从消息的发送者指向消息的接收者。

- 对象的结构（关联关系）
  - 分类（继承）关系：使用指向基类的空三角形表示
  - 组合（聚合）关系：
    - 聚集：空心菱形表示，如线由两点组成
    - 组合：实心菱形表示，如线及其颜色、宽度

- 对象的连接
  - 实例连接：反映静态关系，用实线表示
    - 一对一
    - 一对多
    - 多对多
  - 消息连接：反映动态关系，有方向，用带箭头的实线表示

三、使用实例

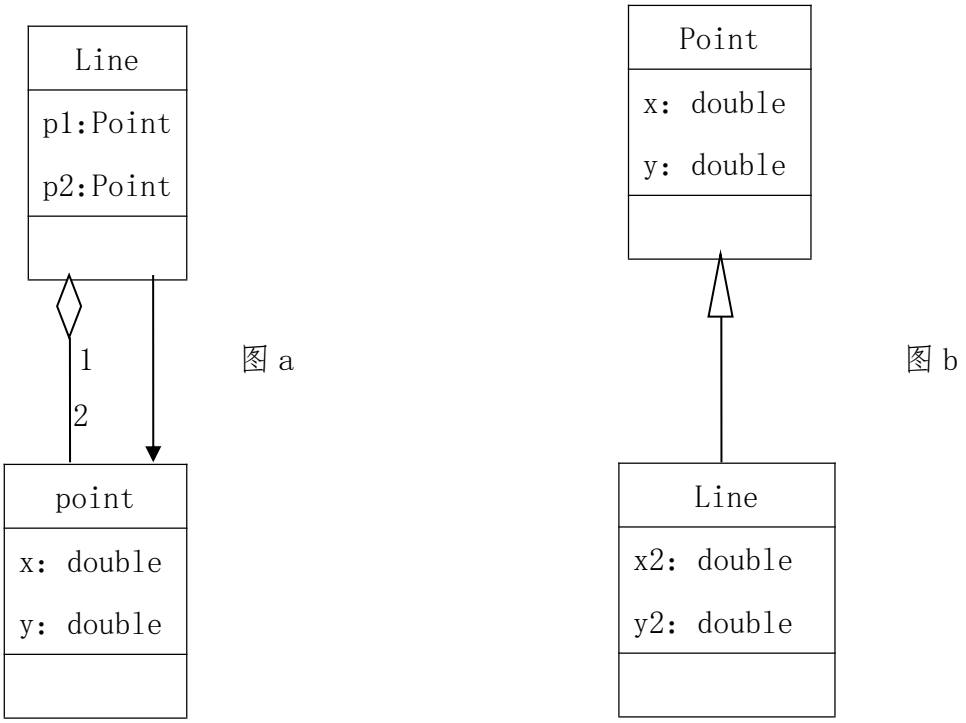
假设已经定义了类 Point，分别使用聚合和继承的方法组成 Line 类，不涉及操作，不用给出成员函数。

1、Line 类包含 Point 类的对象

假设 Point 的对象为 p1 和 p2，则它们构成 Line 类的两个数据成员，Line 类使用这两对值作为两个坐标点，构成一条线段。显然，它们具有各自的生存周期，应使用空心菱形连接，关联关系是一对二。

另外，Line 类的数据成员 p1 和 p2 需要具有确定的属性值，这由 Point 类完成，为此 Line 类必须向 Point 类发消息，请求 Point 类为自己构造两个点对

象，消息连接的箭头指向Point类。如下图a:



图a　　图b

2、Line类继承Point类

如用继承的思路构造Line类,则是Line类继承Point类的坐标点作为线段的一个坐标点，自己再定义一对属性作为另一个端点，这样就可构成一条线段。

因为基类是Point，所以三角形指向Point类，一般情况下，继承关系的消息传递规律清楚，为了保持图的清晰，可以不画它们之间的消息连接，另外，不管两者之间有多少消息，也均画出一条，如上图b。

四、对象、类和消息

由以上分析可见，对象的属性是指描述对象的数据成员，对象属性的集合又称为对象的状态。

对象的行为是定义在对象属性上的一组操作的集合，操作（成员函数）是响应消息而完成的算法，表示对象内部实现的细节，对象的操作集合体现了对象的行为能力。

对象的属性和行为是对象定义的组成要素,分别代表了对象的静态和动态特征，无论对象是简单的或是复杂的，对象一般具有以下特征:

①有一个状态，由与其相关联的属性值集合所表征；

②有唯一标识名，可以区别于其他对象；

③有一组操作方法，每个操作决定对象的一种行为；

④对象的状态只能被自己的行为所改变；

⑤对象的操作包括自身操作（施加于自身）和施加于其他对象的操作；

⑥对象之间以消息传递的方式进行通信；

⑦一个对象的成员仍可以是一个对象。

其中，前三条是对象的基本特征，后四条属于特征的进一步定义。

消息是向对象发出的服务请求,它是面向对象系统中实现对象间的通信和请求任务的操作。消息传递是系统构成的基本元素，是程序运行的基本处理活动。一个对象所能接受的消息及其所带的参数，构成该对象的外部接口。对象接收它能识别的消息，并按照自己的方式来解释和执行。一个对象可以同时向多个对象发送消息，也可以接收多个对象发来的消息。消息值反映发送者的请求，由于消息的识别和解释取决于接收者,因而同样的消息在不同对象中可解释成不同的行为。

对象传送的消息一般由 3 部分组成:接收对象名、调用操作名和必要的参数。每个消息在类描述中用一个相应的方法给出，即使用成员函数定义操作。向对象发送一个消息，就是引用一个方法的过程，即实施对象的各种操作就是访问一个或多个在类中定义的方法。

消息协议是一个对象对外提供服务的规定格式说明,外界对象能够并且只能向该对象发送协议中所提供的消息，请求该对象服务。具体实现是将消息分为私有和公有消息，前者只供内部使用，后者是对外的接口，协议则是一个对象所能接受的所有公有消息的集合。

§4.10 面向对象编程的文件规范

一般将类的说明放在头文件中，实现则放在.cpp 文件中，主程序单独使用一个文件，这就是多文件编程规范。另外，非常简单的成员函数可以在声明中定

义（即默认内联函数形式），而在.cpp 文件中，要将头文件包含进去。

对规模较大的类，一般应该为每个类设立一个头文件和一个实现文件。但函数模板和类模板比较特殊，如果使用头文件说明，则要同时完成它们的定义。

一、编译指令

1、嵌入指令

嵌入指令#include 指示编译器将一个源文件嵌入到该指令所在的位置。尖括号或双引号中的文件名可含有路径信息。如：

#include <\user\prog.h>

注意：由于编译指令不是 C＋＋的一部分，因此在这里表示反斜杠时只需使用一个反斜杠。如果在 C＋＋程序中表示上述文件名，则必须使用双反斜杠，如：

char fname [ ]=\\user\\prog.h;

2、宏定义

宏定义以＃define 开头，＃define 指令定义一个标识符及串，在源程序中每次遇到该标识符时，编译器将自动用后面的串代替它。该标识符称为宏名，替换过程称为宏替换。

宏定义的一般形式为：＃define 宏名 替换正文

其中宏名必须是一个有效的 C＋＋标识符，替换正文可为任意字符组成的字符序列。宏名和替换正文之间至少有一个空格，一般将宏名写成大写字体。

注意：宏定义由新行结束，而不以分号结束，分号视为替换正文的一部分。

当替换正文要书写在多行上时,除最后一行之外,每行的行尾要加上一个反斜线，表示宏定义继续到下一行，如：

```
＃define MAX（a，b）((a) > (b) ? \
                (a):(b))
```

因宏定义有许多不安全因素，对需要使用无参数宏的场合，应该尽量使用 const 代替宏定义。如：语句“#define ABC 2150”和语句“const int ABC 2150;”作用一样，但后者比前者要安全。

宏定义不再使用时，可用＃undef 删除。

3、条件编译指令

条件编译指令是指#if、#else、#elif 和#endif，它们构成了类似 C++的 if 选择结构，其中#endif 表示一条指令结束。

其中#if 用于控制编译器对源程序的某部分有选择的进行编译，该部分从#if 开始，到#endif 结束，如果#if 后面的常量表达值为真，则编译这部分，否则就不编译，此时，这部分代码相当于被删除。

#else 是在#if 测试失败的情况下建立另外一种选择，可以在#else 分支中使用编译指令#error 输出出错信息，形式为：#error 出错信息　　其中出错信息是一个字符序列，当遇到#error 指令时，编译器显示其后面的出错信息，并中止对程序的编译。

编译指令可嵌套，规则和编译器对其处理的方式与 if 语句嵌套情况类似。

4、defined 操作符

关键字 defined 不是指令，而是一个预处理操作符，用于判定一个标识符是否已经被#defined 定义，例如标识符 identifier 已被#defined 定义，则 defined（idertifier）为真，否则为假。

条件编译指令#ifdef 和#ifndef 用于测试其后的标识符是否被#define 定义，如果已被定义，则#ifdef 为真，#ifndef 测试为假；如果没有被定义，则#ifdef 测试为假，#ifndef 测试为真。

如：#if!defined (HEAD_H) 和 #ifndef HEAD_H 两语句效果一样。

二、在头文件中使用条件编译

在多文件设计中，由于文件包含指令可以嵌套使用，可能会出现不同文件包含了同一个头文件，这样会引起变量及类的重复定义。为了避免重复编译类的同一个头文件，可对这类头文件使用条件编译。例如 Point.h 可能会被嵌套包含，则使用如下形式：

```
#if! defined (POINT_H)
    #define POINT_H
class Point{
    …… //类体
};
#endif
```

语句①

如果 Point.h 没有被嵌套包含，则编译语句①，否则，就不编译，此时语句①相当于被从源文件中删除。

第五章　特殊函数和成员

§5.1　对象成员的初始化

一、对象成员的概念

如果在一个类 A 中说明的数据成员的类型属于另外某个类或某些类，则称这些成员为对象成员。在类 A 中说明对象成员的一般形式为：

```
class A{
    类名 1 成员名 1;
    类名 2 成员名 2;
    ……
    类名 n 成员名 n;
};
```

说明对象成员是在其所属类名之后给出对象成员的名字。

二、含对象成员的类的构造函数格式

为初始化这些对象成员，A 类的构造函数需要分别调用这些对象成员所属类的构造函数。含有对象成员的类 A 的构造函数的定义形式为：

A:: A（参数表 0）：成员 1（参数表 1），成员 2（参数表 2），…，成员 n（参数表 n）　{…… //其他操作}

冒号“:”后由逗号隔开的项组成成员初始化列表，其中的参数表给出了为调用相应成员所在类的构造函数时应提供的参数，参数列表中的参数都来自于“参数表 0”，可以使用任意复杂的表达式，其中可以有函数调用。

对象成员构造函数的调用顺序取决于这些对象成员在类 A 中说明的顺序，与它们在成员初始化列表中的顺序无关。当建立 A 类的对象时，先调用对象成员的构造函数，初始化对象成员，然后才执行 A 类的构造函数，初始化 A 类中的其他成员。析构函数的调用顺序与构造函数正好相反。

例：含对象成员的类如何调用构造函数和析构函数

```
#include <iostream>
using namespace std;
class object{
    private:
        int val;  //声明 int 型的 object 类的私有数据成员 val
    public:
        object( ):val(0) {cout<<"信息一"<<endl;}  //语句①，定义不带参数的 object 类的构造函数 object 为内联函数
        object(int i):val(i) {cout<<"信息二"<<val<<endl;}  //语句②，定义带一个参数的 object 类的构造函数 object 为内联函数
        ~object( ) {cout<<"信息三"<<val<<endl;}  //语句③，定义 object 类的析构函数为内联函数
};                                  //类 object 的声明结束
class container{
    private:
        object one;  //声明 object 类的对象 one 为 container 类的对象成员，初始化顺序为第一
        object two;  //声明 object 类的对象 two 为 container 类的对象成员，初始化顺序为第二
        int data;    //声明 int 型的 container 类的私有数据成员 data，初始化顺序为第三
    public:
        container( ):data(0) {cout<<"信息四"<<endl;}  //语句④，定义不带参数的 container 类的构造函数 container 为内联函数
        container(int i,int j,int k);  //声明带 3 个 int 型参数的 container 类的构造函数 container
        ~container( ) {cout<<"信息六"<<data<<endl;}  //语句⑥，定义 container 类的析构函数为内联函数
```

};　　//类 container 的声明结束

```
container::container(int i,int j,int k):two(i),ont(j) {data=k;cout<<"信息五"<<data<<endl;}
```

//语句⑤，定义带 3 个 int 型参数的 container 类的构造函数

```
void main(){container obj,an0bj(5,6,10);}
```

//语句⑦,生成两个 container 类的对象，其中一个对象 obj 不带参数，另一个对象 an0bj 带 3 个参数

上面程序的主要执行步骤为：

第一步：执行语句⑦中的 container obj，生成 container 类的对象 obj 并初始化，该对象具有三个私有数据成员 one、two 和 data，其中 one 和 two 都是类型为 object 类的对象成员，三者的初始化顺序应该按照在 container 类中的说明顺序，分别为 one、two 和 data，而不是按照初始化列表中的顺序 two、one 和 data。因为 one 和 two 是 object 类的对象且无参数，所以调用 object 类的无参数构造函数语句①先后对 one 和 two 进行初始化，并输出“信息一”、“信息一”；然后执行 container 类的构造函数，对除对象成员 one 和 two 外的其他的 container 类的成员 data 初始化，即调用 container 类的无参数构造函数语句④，并输出“信息四”。

第二步：执行语句⑦中的 container an0bj（5，6，10），生成 container 类的对象 an0bj 并初始化，该对象具有三个私有数据成员 one、two 和 data，其中 one 和 two 都是类型为 object 类的对象成员，三者的初始化顺序应该按照在 container 类中的说明顺序，分别为 one、two 和 data，而不是按照初始化列表中的顺序 two、one 和 data。因为 one 和 two 是 object 类的对象且实参分别为 6、5，所以调用 object 类的含一个形参的构造函数语句②，先后对 one 和 two 进行初始化，并输出“信息二 6”、“信息二 5”；然后执行 container 类的构造函数，对除对象成员 one 和 two 外的其他的 container 类的成员 data 初始化，即调用 container 类的含三个参数的构造函数语句⑤，并输出“信息五 10”。

第三步：按 container 类的对象出现的顺序反序析构对象 an0bj（5，6，10）和 obj，即先析构对象 an0bj，针对对象 an0bj 的三个数据成员，按照其在 container 类中的声明顺序反序析构，即先调用语句⑥，析构成员 data，输出“信

息六 10”; 然后析构成员 two，调用语句③，输出“信息三 5”; 最后析构成员 one，调用语句③，输出“信息三 6”。

第四步：析构 container 类的对象 obj，针对对象 obj 的三个数据成员，按照其在 container 类中的声明顺序反序析构，即先调用语句⑥，析构成员 data，输出“信息六 0”; 然后析构成员 two，调用语句③，输出“信息三 0”; 最后析构成员 one，调用语句③，输出“信息三 0”。

三、基本数据类型的成员和特殊成员的初始化

1、基本数据类型的成员的初始化

如上例中 container 类的基本数据类型成员 data 的初始化，可如上例中在函数体内通过语句“data＝k;”初始化，也可在成员初始化列表中进行，如将语句⑤改为:container::container(int i,int j,int k):two(i),ont(j),data(k){cout<<”信息五”<<data<<endl;}

2、特殊成员的初始化

特殊成员如 const 成员、引用成员的初始化，则必须通过成员初始化列表进行，如:

```
class exam{
    private:
        const int num;
        int& ret;
    public:
        exam(int n,int f):num(n),ret(f){ }
};
```

§5.2 静态成员

一、静态成员概念

简单成员函数：函数声明中不含 const、volatile、static 关键字。

静态成员：如果类的数据成员或成员函数使用关键字 static 修饰，这样的成员称为静态数据成员或静态成员函数。

例：含静态成员的程序

```
class Test {
        static int x;              //声明静态数据成员
        int n;
    public
        Test( ){ }        //定义无参数的 Test 类的构造函数
        Test(int a,int b){x=a;n=b;}  //定义含两个参数的 Test 类的构造函数 Test 为内联函数
        static int func( ){return x;}  //定义静态成员函数 func 为内联函数
        static void sfunc(Test&r,int a){r.n=a;}  //定义静态成员函数 sfunc 为内联函数,函数以 Test 类的引用 r 和整形数 a 为参数
        int Getn( ){return n;}  //定义成员函数 Getn 为内联函数
};                          //类 Test 的声明结束
int Test::x=25;     //初始化静态数据成员
#include <iostream>
using namespace std;
void main( ){
    cout<<Test::func( );     //x 在对象产生之前就存在，输出“25”
    Test b,c;          //利用无参数的构造函数产生 Test 类的对象 b 和 c
    b.sfunc(b,58);  //设置对象 b 的数据成员 n，n 值为 58，r 为 b 的引用
    cout<<” ”<<b.Getn( );  //输出“ 58”
    cout<<” ”<<b.func( );  //x 属于所有对象，输出“ 25”
    cout<<” ”<<c.func( );  //x 属于所有对象，输出“ 25”
    Test a(24,56);  //利用含两个参数的构造函数产生 Test 类的对象 a，并将 x 的值改为 24，给 a 的私有数据成员 n 赋值 56
    Cout<<” ”<<a.func( )<<” ”<<b.func( )<<” ”<<c.func( )<<endl;  //x 属于所有对象，其值已经变为 24，输出“ 24 24 24”
```

静态数据成员只能说明一次，如果在类中仅对其进行声明，则必须在文件作用域的某个地方进行定义，在进行初始化时，必须进行成员名限定，如：“int Test::x=25;”，也可直接在构造函数中使用类成员限定符对其进行初始化，如：“Test(int a,int b) {Test::x=a;n=b;}”。

由于数据隐藏的需要，静态数据成员常被说明为私有的，而通过定义公有的静态成员函数来访问它。

另外，由于 static 不是函数类型中的一部分，所以在类声明之外定义静态成员函数时，不使用 static，在类中定义的静态成员函数是内联的。

二、静态成员函数与一般成员函数的不同

1、可以不指向某个具体的对象，只与类名连用；

2、在没有建立对象之前，静态成员就已存在；

3、静态成员是类的成员，不是对象的成员；

4、静态成员为该类的所有对象共享，它们被存储于一个公用内存中；

5、没有 this 指针，只能通过对象名或指向对象的指针访问类的数据成员；

6、不能被说明为虚函数；

7、不能直接访问非静态函数；

三、类的静态对象与普通对象的对比使用实例

静态对象是使用关键字 static 声明的类的对象，它与普通对象区别如下：

1、静态对象的构造函数在代码执行过程中，在第一次遇到它的变量定义并初始化时被调用，但直到整个程序结束之前仅调用一次；而普通对象则是遇到变量定义就被调用，遇到几次调用几次。

2、静态对象的析构函数在整个程序退出之前被调用，同样也只调用一次；而普通对象则是变量被定义几次，则析构几次。

例：静态对象和普通对象的对比使用实例

```
#include <iostream>
```

```
using namespace std;
class test{
    private:
        int n;     //声明 test 类的私有数据成员 n
    public:
    test(int i){n=i;cout<<"构造:"<<i<<endl;}  //语句①，定义含一个整型参数的 test 类的内联构造函数为公有成员函数
    ~text(){cout<<"消除:"<<n<<endl;  //语句②，定义 test 类的内联析构函数
    int getn( ){return n;}      //语句③，定义 test 类的内联函数 getn
    void inc( ){++n;}     //语句④，函数作用为：使用 n 之前，使 n 的值加 1
};
void main( ){
    cout<<"循环开始:"<<endl;
    for(int i=0;i<3;i++){
        static test a(3);  //语句⑤，定义 test 类的静态对象 a 并初始化
        test b(3);  //语句⑥，定义 test 类的普通对象 b 并初始化
        a.inc( );  //语句⑦，test 类的静态对象 a 调用其同类的成员函数 inc
        b.inc( );  //语句⑧，test 类的普通对象 b 调用其同类的成员函数 inc
        cout<<"a.n="<<a.getn( )<<endl;  //语句⑨
        cout<<"b.n="<<b.getn( )<<endl;  //语句⑩
    }
    cout<<"循环结束"<<endl;
    cout<<"退出主程序"<<endl;
}
```

上述程序的主要执行步骤为：

第一步：输出信息“循环开始”。因 for 循环的 i 值为 0，for 循环开始，执行语句⑤，定义 test 类的静态对象 a，调用语句①对其初始化，将实参的值 3

赋给构造函数的形参 i，执行构造函数函数体内语句，将 i 的值 3 赋给 n，输出信息“构造 3”;

第二步，执行语句⑥，定义 test 类的普通对象 b，调用语句①对其初始化，将实参的值 3 赋给构造函数的形参 i，执行构造函数函数体内语句，将 i 的值 3 赋给 n，输出信息“构造 3”;

第三步：执行语句⑦，test 类的静态对象 a 调用其同类的成员函数 inc，使得 n 值在使用前加 1，n 值变为 4;

第四步：执行语句⑧，test 类的普通对象 b 调用其同类的成员函数 inc，使得 n 值在使用前加 1，n 值变为 4;

第五步：执行语句⑨，test 类的静态对象 a 调用其同类的成员函数 getn，输出信息“a.n=4”;

第六步：执行语句⑩，test 类的普通对象 b 调用其同类的成员函数 getn，输出信息“b.n=4”，执行 i++，将 for 循环的 i 值加 1，变为 1;

第七步：本次循环结束，因 test 类的普通对象 b 生命期与本次循环共存，调用语句②，析构对象 b，输出信息“消除 4”;

第八步：for 循环的 i 值为 1，开始第二次循环，因静态对象 a 在整个程序结束前仅调用一次构造函数，在以后的循环中将不再对 a 进行初始化，故只需重新定义 test 类的普通对象 b，调用语句①对其初始化，将实参的值 3 赋给构造函数的形参 i，执行构造函数函数体内语句，将 i 的值 3 赋给对象 b 的 n，输出信息“构造 3”;

第九步：执行语句⑦，test 类的静态对象 a 调用其同类的成员函数 inc，使得 n 值在使用前加 1，因对象 a 的 n 值在第一次循环中已经变为 4，故本次循环再加 1 后，对象 a 的 n 值变为 5;

第十步：执行语句⑧，test 类的普通对象 b 调用其同类的成员函数 inc，使得 n 值在使用前加 1，n 值变为 4;

第十一步：执行语句⑨，test 类的静态对象 a 调用其同类的成员函数 getn，输出信息“a.n=5”;

第十二步：执行语句⑩，test 类的普通对象 b 调用其同类的成员函数 getn，输出信息“b.n=4”，执行 i++，将 for 循环的 i 值加 1，变为 2;

第十三步：本次循环结束，因 test 类的普通对象 b 生命期与本次循环共存，调用语句②，析构对象 b，输出信息“消除 4”；

第十四步：for 循环的 i 值为 2，开始第三次循环，同第八步，输出信息“构造 3”；

第十五步：同第九步，因对象 a 的 n 值在第二次循环中已经变为 5，故本次循环再加 1 后，对象 a 的 n 值变为 6；

第十六步：同第十步，对象 b 的 n 值变为 4；

第十七步：同第十一步，输出信息“a.n=6”；

第十八步：同第十二步，输出信息“b.n=4”；

第十九步：同第十三步，输出信息“消除 4”，for 循环的 i 值加 1，变为 3

第二十步：因 for 循环的 i 值为 3，不满足继续循环的条件，所以退出循环，执行后面的语句，输出信息“循环结束”、“退出主程序”，整个程序结束，析构 test 类的静态对象 a，调用语句②，输出信息“消除 6”，整个程序结束。

§5.3　友元函数

概念：可以在类 A 中通过关键字 friend 声明或定义某个独立函数或另一个类 B 的某个成员函数或另一个类 B 为类 A 的友元函数，友元函数可以无限制的存取类 A 的成员（包括私有、公有和保护成员）。

定义形式：friend 函数类型 函数所在类名::函数名（参数列表）;

友元函数可在类中的私有或公有部分通过关键字 friend 说明或定义，但如在类中声明，而在类外定义，就不能再在类外使用 friend 关键字。友元函数作用域的开始点在它的说明点，结束点和类的作用域相同。

友元函数应被看作类的接口的一部分，使用它的主要目的是提高效率，因为它可以直接访问对象的私有成员，从而省去调用类的相应成员函数的开销。友元函数的另一个优点是：类的设计者不必在考虑好该类的各种可能使用情况之后再设计这个类，而是可以根据需要，通过使用友元来增加类的接口。

一、类外的独立函数作为类的友元函数

类外的独立函数作为类的友元函数，这个独立函数其实就是一个普通的函

数，仅有的不同点是：它在类中被说明为类的友元函数，可以访问该类所有对象的私有成员。

例：使用友元函数计算两点距离的实例

```
#include <iostream>
#include <cmath>
using namespace std;
class Point {
    private:
        double X,Y;
    public:
        Point( double xi,double yi){X=xi,Y=yi;}  //类 Point 的构造函数
        double GetX( ){return X;}
        double GetY( ){return Y;}
        friend double distances( Point&, Point&);  //声明友元函数
};
double distances( Point& a, Point& b)  //像普通函数一样定义友元函数
{   double dx=a.X-b.X;  //因是友元函数，所以可以直接访问对象的私有数据成员
    double dy=a.Y-b.Y;  //因是友元函数，所以可以直接访问对象的私有数据成员
    return sqrt( dx*dx + dy*dy );
}
void main( ){
    Point p1(3.5,5.5),p2(4.5,6.5);  //语句①
    Cout<<"距离是"<<distances(p1,p2)>>endl;  //语句②
}
```

程序主要执行步骤为：

第一步：执行语句①，定义 Point 类的对象 p1，并调用 Point 类的构造函数对其初始化，将实参的值 3.5 和 5.5 分别赋给形参 xi、yi，执行函数体内语句，将 xi、yi 的值赋给 p1 的私有数据成员 X 和 Y；然后定义 p2，过程同 p1。

第二步：执行语句②，输出信息“距离是1.41421”，调用函数distances，将实参p1和p2的名字（实际上是地址）传递给形参a、b，a和b作为p1和p2的引用（这样做的好处是传引用的方式不会产生临时对象），这时因为函数distances是Point类的友元函数，所以它可以直接访问Point类的对象的私有数据成员，即“double dx=a.X-b.X;”和“double dy=a.Y-b.Y;”两语句成立。否则，如果没有在类的声明中说明函数distances为类Point的友元函数的话，尽管该函数使用Point类的引用作为自己的参数，也必须使用“double dx=a.GetX-b.GetX;”和“double dy=a.GetY-b.GetY;”才能实现dx和dy的计算。

第三步，程序结束，析构Point类的对象p1和p2。

在这个例子中，函数distances被声明为类Poing的友元函数，因为它不是类Point的成员函数，所以没有this指针，在访问类Point的对象的私有数据成员时，必须使用对象名，而不能直接使用成员名。

二、类B的成员函数作为类A的友元函数

一个类的成员函数（包括构造函数和析构函数）可以通过使用friend说明为另一个类的友元函数。例如将类One的成员函数func说明为类Two的友元，可在类Two中用语句：“friend void One::func(Two&);”实现，其中因func属于类One,所以要用限定符说明出处。这样One的对象就可以通过友元函数func（Two&）访问类Two的所有成员，因为是访问类Two，所以应使用类Two对象的引用作为传递参数。

例：类One的对象通过友元函数访问类Two的对象的私有数据

```
#include <iostream>
using namespace std;
class Two;   //先声明类Two，以便类One引用Two&
class One {
    private:
        int x;
    public:
        One(int a){x=a;}   //语句①，定义类One的构造函数为内联公有函数
```

```
        int Getx( ){return x;}  //语句②
        void func(Two&);  //声明本类的成员函数，参数为类 Two 的引用
};          //类 One 声明结束
class Two{
    private:
        int y;
    public:
        Two(int b){y=b;}  //语句③，定义类 Two 的构造函数为内联公有函数
        int Gety( ){return y;}  //语句④
        friend void One::func(Two&);  //声明类 One 的成员函数为本类的友元函数
};          //类 Two 声明结束
void One::func(Two& r) {r.y=x;}  //语句⑤，定义函数 func，以类 Two 对象的引用为
参数。因其是类 One 的成员函数，所以可自由存取所在类 One 的私有数据成员；因其在类 B 中
被声明为友元函数，所以可使用对象名存取类 Two 的私有数据成员（r.y=x 即说明了这些）
void main( ){
    One Obj1(5);    //语句⑥，生成类 One 的对象 Obj1
    Two Obj2(8);    //语句⑦，生成类 Two 的对象 Obj2
    Cout<<Obj1.Getx( )<<" "<<Obj2.Gety( )<<endl;  //语句⑧，输出 5 8
    Obj1.func(Obj2);    //语句⑨
    Cout<<Obj1.Getx( )<<" "<<Obj2.Gety( )<<endl;  //语句⑩，输出 5 5
```

上面程序的主要执行步骤为：

第一步：执行语句⑥，调用语句①，定义类 One 的对象 Obj1 并初始化，将实参的值 5 赋给形参 a，并执行函数体内语句 x=a,将 a 的值赋给 Obj1 的私有数据成员 x，使 x 的值变为 5；

第二步：执行语句⑦，调用语句③，定义类 Two 的对象 Obj2 并初始化，将实参的值 8 赋给形参 b，并执行函数体内语句 y=b,将 b 的值赋给 Obj2 的私有数据成员 y，使 y 的值变为 8；

第三步：执行语句⑧，调用语句②和语句④，输出“5 8”；

第四步：执行语句⑨，调用语句⑤，将实参 Obj2 的名字赋给形参 r（其实是将 Obj2 的地址传给 r），r 为 Ojb2 的引用，因为采用传引用的方式，所以不产生临时对象，执行函数体内语句，将 Ojb1 的私有数据成员 x 的值赋给 Obj2 的私有数据成员 y，y 的值变为 5，x 的值不变仍为 5；

第五步：执行语句⑩，调用语句②和语句④，输出“5 5”；

第六步：程序结束，调用类 Two 的默认析构函数析构对象 Obj2，然后调用类 One 的默认析构函数析构对象 Obj1。

注意：因为类 Two 通过关键字 friend 将类 One 的成员函数 func 声明为友元函数，所以 One 的对象可以通过友元函数 func 访问 Two 对象的私有数据成员。因为 func 是类 One 的成员函数，所以它不用对象名即可自由存取所在类 One 的私有数据成员；同时因为 func 不是类 Two 的成员函数，而只是它的友元函数，所以 func 只能通过对象名存取类 Two 的私有数据成员，func 函数的函数体内语句 r.y=x 清楚的说明了这一点。

三、将一个类说明为另一个类的友元

可以将一个类 One 说明为另一个类 Two 的友元，这时，整个类 One 的成员函数均是类 Two 的友元函数，声明语句简化为：“friend class 类名;”。

例：类 One 作为类 Two 的友元的实例

```
#include <iostream>
using namespace std;
class Two{
        int y;
    public:
        friend class One;  //声明类 One 为类 Two 的友元
};
class One{                 //类 One 的成员函数均是类 Two 的友元函数
        int x;
    public:
        One(int a,Two&r,int b){x=a;r.y=b;}  //语句①，利用类 One 的构造函数给本类及类 Two 的对象赋值
```

```
    Void Display(Two&);  //声明类 Ont 的成员函数，它能访问类 Two 的成员
};
void One::Display(Two&r){cout<<x<<" "<<r.y<<endl;}  //语句②
void main( ){
    Two Obj2;  //语句③
    One Obj1(23,Obj2,55);  //语句④
    Obj1.Display(Obj2);    //语句⑤
}
```

上面程序的主要执行步骤为：

第一步：执行语句③，调用类 Two 的默认构造函数 Two::Two( ){ };生成类 Two 的对象 Obj2，其值未定；

第二步：执行语句④，调用语句①，将实参 23、Obj2、55 赋给形参 a、r、b，其中 r 为类 Two 的对象 Obj2 的引用，执行函数体内语句，将 a、b 的值赋给 x 和 y，x、y 的值变为 23、55。其中因构造函数本身属于类 One 的成员，所以可以直接存取本类对象 Ojb1 的私有数据成员 x，而作为类 Two 的友元，其对类 Two 的对象 Obj2 的私有数据 y 的修改则只能通过对象名实现，相应语句为“r.y=b;”。

第三步：执行语句⑤，调用语句②，将实参 Obj2 的名字赋给形参 r（实际上传递的是地址），r 作为 Obj2 的引用，不产生临时对象，不需调用构造函数，执行函数体内语句，输出 Obj1 的私有数据成员 x 的值和 Obj2 的私有数据成员 y 的值，输出信息“23 55”。

第四步：先后执行类 One 和类 Two 的默认析构函数，析构对象 Obj1 和 Obj2。

§5.4 const 对象

const 对象只能访问 const 成员函数，否则将产生编译错误。

一、常量成员

常量成员包括静态常数据成员、常数据成员和常引用，其中前者仍保留静态成员特征，需要在类外初始化，后两者则只能通过初始化列表来获得初值。

例：常数据成员初始化和常引用作为函数参数

```
#include <iostream>
using namespace std;
class Base{
    private:
        int x;
        const int a;              //常数据成员只能通过初始化列表来获得初值
        static const int b;       //静态常数据成员需在类外初始化
        const int& r;             //常引用只能通过初始化列表来获得初值
    public:
        Base(int,int);  //声明含两个整型参数的Base类的构造函数
        void Show( ){cout<<x<<","<<a<<","<<b<<","<<r<<endl;}  //语句①
        void Display(const double& r){cout<<r<<endl;}    //语句②
};
const int Base::b=125;     //语句③，静态常数据成员在类外初始化
Base::Base(int i,int j):x(i),a(j),r(x){ }  //语句④，初始化列表
void main( ){
    Base a1(104,118),a2(119,140);  //语句⑤
    a1.Show( );     //语句⑥
    a2.Show( );     //语句⑦
    a2.Display(3.14159);     //语句⑧
}
```

上面函数的主要执行步骤为：

第一步：执行语句③，在类外初始化静态常数据成员b，b值变为125

第二步：执行语句⑤，生成Base类对象a1，调用语句④，初始化a1，执行初始化列表，将实参104、118的值赋给形参x、a，将x的值104赋给形参r，a1的私有数据成员x、a、b、r的值分别为104、118、125、104；生成Base类

对象 a2，调用语句④，初始化 a2，执行初始化列表，将实参 119、140 的值赋给形参 x、a，将 x 的值 104 赋给形参 r，a2 的私有数据成员 x、a、b、r 的值分别为 119、140、125、119；

第三步：执行语句⑥，调用语句①，输出“104，118，125，104”；

第四步：执行语句⑦，调用语句①，输出“119，140，125，119”；

第五步：执行语句⑧，调用语句②，将实参 3.14159 的值赋给形参 r，因引用不产生临时对象，所以输出“3.14159”

第六步：执行 Base 类的默认析构函数，先后析构对象 a2、a1 和 b。

说明：使用引用作为函数参数，传送的是地址，所以形参改变，则实参也跟着改变，但如果不希望函数改变对象的值，就要使用常引用作为参数，例如上例中 Display 函数中将一个 double 型对象作为实参，因声明为常引用，故 Display 只能使用而不能改变 r 所引用的对象。

二、常对象

在对象名前使用 const 声明的对象就是常对象，常对象必须在声明的同时进行初始化，而且不能被更新，形式为：const 类名 对象名（参数表）; 或

类名 const 对象名（参数表）; //必须在声明的同时进行初始化

例如上例中定义一个 Base 类的常对象，语句为：Base const a(24,35);

三、常成员函数

为防止覆盖函数改变数据成员的值，可以将一个成员函数声明为 const 函数，const 函数不能更新对象的数据成员，也不能调用该类中没有用 const 修饰的成员函数。

注意用 const 声明 static 成员函数没有什么作用，另外，在 C++中声明构造函数和析构函数时使用 const 关键字是非法的，而 volatile 关键字的使用方法与 const 类似，因使用很少，故不再介绍。

对于一般对象，它可以调用所有成员函数，包括常成员函数，而对于 const 对象，它只能调用 const 函数，不能调用非 const 函数。故可以通过常成员函数参与对重载函数的区分。

1、Const 函数的声明及定义形式为：

（1）在类体内定义 const 函数为内联函数时的形式

类型标识符 函数名（参数列表） const {……//函数体}

（2）在类体内声明，类体外定义时的形式

声明形式：类型标识符 函数名（参数列表） const;

定义形式：类型标识符 类名::函数名（参数列表） const {……//函数体}

2、例：const 对象调用 const 函数的实例

```
#include <iostream>
using namespace std;
class Base{
    private:
        double x,y;
        const double p;
    public:
        Base(double m,double n,double d):p(d) {x=m;y=n;}
                                    //语句 1，常数据成员只能通过初始化列表来获得初值
        void Show( );
        void Show( ) const;     //声明常成员函数
};
void Base::Show( ){cout<<x<<","<<y<<" p="<<p<<endl;}  //语句 2
void Base::Show( ) const {cout<<x<<","<<y<<" const p="<<p<<endl;}  //语句 3
void main( ){
    Base a(8.9,2.5,3.1416);     //语句 4，调用语句 1 进行初始化
    const Base b(2.5,8.9,3.14);       //语句 5，调用语句 1 进行初始化
    b.Show( );        //语句 6，调用语句 3，输出“2.5,8,9 const p=3.14”
    a.Show( );        //语句 7，调用语句 2，输出“2.5,8,9 p=3.1416”
}
```

3、例：const 函数返回的常量对象与其他常量对象一起使用的实例

```
#include <iostream>
using namespace std;
class ConstFun{
    public:
        int f5( ) const{return 5;}     //常成员函数，返回常量对象
        int Obj( ){return 45;}         //一般成员函数
};
void main( ){
    ConstFun s;  //声明 ConstFun 类的一般对象 s
    const int i=s.f5( );  //对象 s 使用常成员函数 f5( )初始化常整数 i
    const int j=s.Obj( ); //对象 s 使用一般成员函数 Obj( )初始化常整数 j
    int x=s.Obj( );  //对象 s 使用一般成员函数 Obj( )初始化整数 x
    int y=s.f5( );  //对象 s 使用常成员函数 f5( )初始化整数 y
    cout<<i<<" "<<j<<" "<<x<<" "<<y<<endl;  //输出“5 45 45 5”
    const ConstFun d;  //声明 ConstFun 类的常对象 d
    int k=d.f5( );     //常对象 d 只能调用常成员函数 f5( )
    cout<<"k="<<k<<endl;     //输出“k=5”
}
```

§5.5 数组和类

类的引入产生了除 C 语言可以声明的数组类型外的一系列新的数组类型，下面简单介绍类对象数组和类对象指针数组。

一、使用类对象数组和指针的实例

```
class Test{
        int num;
        double f1;
    public:
        Test (int n){num=n;}  //语句 1，一个参数的构造函数
        Test(int n,double f){num=n;f1=f;}  //语句 2，两个参数的构造函数
```

```
        int GetNun( ){return num;}
        int GetF( ){return f1;}
};
#include <iostream>
using namespace std;
void main( ){
    Test one[2]={2,4},*p;  //定义含两个元素的Test类对象数组one和一个指向Test类的对象指针p，one的两个元素调用语句1初始化
    Test two[2]={Test(1,3.2),Test(5,9.5);  //定义含两个元素的Test类的对象数组two，two的两个元素调用语句2初始化
    for (int i=0;i<2;i++)
        cout<<"one["<<i<<"]="<<one[i].GetNum( )<<",";
    p=two;  //因数组名即代表数组首地址，所以可以用数组名给指针p赋值
    for (i=0;i<2;i++,p++)
        cout<<"two["<<i<<"]=("<<p->.GetNum( )<<","<<p->GetF( )<<"),";
}
```

程序输出结果：one[0]=2,one[1]=4,two[0]=(1,3.2),two[1]=(5,9.5),

二、仍然使用类Test，但主程序改用对象指针数组演示

……

```
void main( ){
    Test *one[2]={new Test(2),new Test(4)};
    Test *two[2]={new Test(1,3.2),new Test(5,9.5)};
    for (int i=0;i<2;i++)
        cout<<"one["<<i<<"]="<<one[i]->GetNum( )<<",";
    for (i=0;i<2;i++,p++)
       cout<<"two["<<i<<"]=("<<two[i]->GetNum( )<<","<<two[i]->GetF( )<<"),";
}
```

例中 one 和 two 都是直接用动态分配的对象初始化的,编译器自动调用构造函数语句 1 和语句 2 来产生 one 和 two 的每一个分量。

§5.6 指向类的成员函数的指针

C++除普通指针外，还包含了能指向类的成员的指针。普通指针指向对象，而指向类的成员的指针则可指向某个特定类的对象的数据成员或成员函数。

一、指向类的数据成员的指针

该指针要求数据成员必须是公有的，用途不大，故不详细介绍。

二、指向类的成员函数的指针

如类 A 的成员函数的参数类型列表为 list，返回类型为 type，则指向该函数的指针 pointer 的声明形式为“type (A::*pointer) (list);”，此声明可读做：pointer 是一个指针，指向类 A 的成员函数，此成员函数参数类型列表为 list，返回值类型为 type。

如果类 A 的成员函数 fun 的原型和 pointer 指向的函数的原型一样,则语句“pointer＝A::fun;”将函数 fun 的地址（不是真实地址，而是在类 A 中所有对象的偏移）置给指针 pointer，即指针 pointer 指向函数 fun（好比数组的情况，数组名即是数组的首地址)。

在使用指向类成员函数的指针访问类的某个成员函数时，必须指定一个对象，通过对象名、引用或指向对象的指针调用该成员函数，其中前两者使用运算符“.*”，后者使用运算符“->”。

例：使用指向类的成员函数的指针

```
#include <iostream>
using namespace std;
class A{
    private:
        int val;
    public:
        A(int i){val=i;}     //含一个整型参数的类 A 的构造函数
```

```
        int value(int a){return val+a;}
};
void main( ){
    int(A::*pfun)(int);     //声明指向类 A 的成员函数的指针 pfun
    pfun=A::value;      //指针指向 A 的具体的成员函数 value
    A obj(10);      //创建类 A 的对象 obj，并调用构造函数初始化
    cout<<(obj.*pfun)(15)<<endl;  //对象 obj 调用指针指向的成员函数 value，输出 25
    A *pc=&obj;      //创建类型为 A 的指针 pc，该指针指向类 A 的对象 obj
    cout<<(pc->*pfun)(15)<<endl;  //指针 pc 调用成员函数 value，输出 25
}
```

注意：程序中(obj.*pfun)或(pc->*pfun)必须用括号括起来。类似于指向对象的指针，指向类成员函数的指针也可看作存储的是类成员函数地址的指针变量，好比 pfun 为指向类成员函数的指针，指向类成员函数 value，则有：

*pfun=value　　但和对象的 p=&x;不同，这里是 pfun=A::value，这和数组名即为数组首地址相似，value 即是函数名，也看作是函数首地址。

§5.7　求解一元二次方程

一、设计代表方程的类

FindRoot
a:float b:float c:float d:float x1:double x2:double
FindRoot:FindRoot Find:void Display:void

类示意图

Obj:FindRoot
a=1 b=-3 c=2 d=1 x1=2 x2=1
FindRoot Find Display

obj 对象图

①②③④⑤⑥⑦⑧⑨⑩