C++基础摘要

笔记本：	C++		
创建时间：	2015/12/15 11:00	**更新时间：**	2016/1/26 15:31
标签：	C++Base		
URL：	http://blog.csdn.net/u013791974/article/details/23604883		

基于C++primerplus和21天学通C++

1，数据类型

1，基本输出和数据类型

1，climits

Climits头文件中定义里符号常量来表示各种整形类型的最大值和最小值。浮点数类型：float，double，long double。cfloat头文件中定义了三种浮点类型的指数范围。

2，大括号赋值

将大括号初始化器用于单值变量的情形还不多，但 C++11 标准使得这种情形更多了。首先，采用这种方式时，可以使用等号（=），也可以不使用：

```
int emus{7};       // set emus to 5
int rheas = {12};  // set rheas to 12
```

其次，大括号内可以不包含任何东西。在这种情况下，变量将被初始化为零：

3，转义字符

通用字符名的用法类似于转义序列。通用字符名可以以\u 或\U 打头。\u 后面是 8 个十六进制位，\U 后面则是 16 个十六进制位。这些位表示的是字符的 ISO 10646 码点（ISO 10646 是一种正在制定的国际标准，为大量的字符提供了数值编码，请参见本章后面的“Unicode 和 ISO 10646”）。

如果所用的实现支持扩展字符，则可以在标识符（如字符常量）和字符串中使用通用字符名。例如，请看下面的代码：

```
int k\u00F6rper;
cout << "Let them eat g\u00E2teau.\n";
```

ö 的 ISO 10646 码点为 00F6，而 â 的码点为 00E2。因此，上述 C++代码将变量名设置为 körper，并显示下面的输出：

```
Let them eat gâteau.
```

4，wchar_t

wchar_t是一种整数类型，有足够的空间，可以表示系统使用的最大扩展字符集。另外char16_t和char32_t分别使用u和U作为前缀，比wchat_t更为实用

cin 和 cout 将输入和输出看作是 char 流，因此不适于用来处理 wchar_t 类型。iostream 头文件的最新版本提供了作用相似的工具——wcin 和 wcout，可用于处理 wchar_t 流。另外，可以通过加上前缀 L 来指示宽字符常量和宽字符串。下面的代码将字母 P 的 wchar_t 版本存储到变量 bob 中，并显示单词 tall 的 wchar_t 版本：

```
wchar_t bob = L'P';         // a wide-character constant
wcout << L"tall" << endl;   // outputting a wide-character string
```

5，bool

C++中已经有bool类型，可以直接使用true和false关键字

6，浮点数E表示法

记住：d.dddE+n 指的是将小数点向右移 n 位，而 d.dddE-n 指的是将小数点向左移 n 位。之所以称为“浮点”，就是因为小数点可移动。

7，类型转换

- 将一种算术类型的值赋给另一种算术类型的变量时，C++将对值进行转换；
- 表达式中包含不同的类型时，C++将对值进行转换；
- 将参数传递给函数时，C++将对值进行转换。

强制转换的通用格式如下：

```
(typeName) value   // converts value to typeName type
typeName (value)   // converts value to typeName type
```

8，auto

C++提供了一个新的关键字，auto作为类型声明，类似于javascrpt中的var

12，

7．这个问题的答案取决于这两个类型的长度。如果 long 为 4 个字节，则没有损失。因为最大的 long 值将是 20 亿，即有 10 位数。由于 double 提供了至少 13 位有效数字，因而不需要进行任何舍入。long long 类型可提供 19 位有效数字，超过了 double 保证的 13 位有效数字。

9，访问全局变量

如果在代码块内外都有一个名称一样的变量，默认情况下访问的是局部变量，如果要访问代码块外面的全局变量，需要使用：：运算符：

```
cout<<"inner:x="<<x<<endl;
cout<<"::x="<<::x<<endl;     //访问重名的全局变量
```

2，复合数据类型

1，数组

如果只对数组一部分进行赋值，则编译器会将其他元素设置为0.

C++11中创建数组的新方式：

首先，初始化数组时，可省略等号（=）：

```
double earnings[4] {1.2e4, 1.6e4, 1.1e4, 1.7e4};  // okay with C++11
```

其次，可不在大括号内包含任何东西，这将把所有元素都设置为零：

```
unsigned int counts[10] = {};  // all elements set to 0
float balances[100] {};        // all elements set to 0
```

第三，列表初始化禁止缩窄转换，这在第 3 章介绍过：

```
long plifs[] = {25, 92, 3.0};           // not allowed
char slifs[4] {'h', 'i', 1122011, '\0'}; // not allowed
char tlifs[4] {'h', 'i', 112, '\0'};     // allowed
```

C++11新增了array模板类，可以替代传统数组，另外vector模板类也可以替代数组。

```
vector<int> vi;

array<int, 5> ai;
```

首先，注意到无论是数组、vector 对象还是 array 对象，都可使用标准数组表示法来访问各个元素。其次，从地址可知，array 对象和数组存储在相同的内存区域（即栈）中，而 vector 对象存储在另一个区域（自由存储区或堆）中。第三，注意到可以将一个 array 对象赋给另一个 array 对象；而对于数组，必须逐元素复制数据。

2，string

string类——C++字符串风格，在string.h头文件中

3，结构体

在C++中定义了结构体之后声明变量时可以直接使用结构体名，前面不需要加入struct。使用.号来访问结构体中的成员。

```
struct inflatable   // structure declaration
{
    char name[20];
    float volume;
    double price;
};
```

```
union one4all
{
    int int_val;
    long long_val;
    double double_val;
};
```

与数组一样，C++11 也支持将列表初始化用于结构，且等号（=）是可选的：

```
inflatable duck {"Daphne", 0.12, 9.98};  // can omit the = in C++11
```

其次，如果大括号内未包含任何东西，各个成员都将被设置为零。例如，下面的声明导致 mayor.volume 和 mayor.price 被设置为零，且 mayor.name 的每个字节都被设置为零：

```
inflatable mayor {};
```

最后，不允许缩窄转换。

访问结构体内容的方法：

1，第一种方式使用->运算符 pt->name
2，使用.号运算符，更像类 (*pt).name

4，枚举

可以使用赋值运算符来显式地设置枚举量的值：

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
```

指定的值必须是整数。也可以只显式地定义其中一些枚举量的值：

```
enum bigstep{first, second = 100, third};
```

这里，first 在默认情况下为 0。后面没有被初始化的枚举量的值将比其前面的枚举量大 1。因此，third 的值为 101。

最后，可以创建多个值相同的枚举量：

```
enum {zero, null = 0, one, numero_uno = 1};
```

其中，zero 和 null 都为 0，one 和 umero_uno 都为 1。在 C++早期的版本中，只能将 int 值（或提升为 int 的值）赋给枚举量，但这种限制取消了，因此可以使用 long 甚至 long long 类型的值。

枚举类型只能复制为前面的key，不能直接赋值，而且不能赋值为不存在的值。

2，C++内存模型

1，指针基本概念

变量应用地址运算符（&），就可以获得它的位置；例如，如果 home 是一个变量，则&home 是它的地址

*被称为解除引用运算符，用于指针，可以获取到改地址所存储的值。C++11种nullprt表示空指针，类似java中NULL。

```
int * pointer = &donuts;——声明指针
```

指针变量+1，增加的量是它指向的类型的字节数，跳过指定类型的字节数；-1亦然。

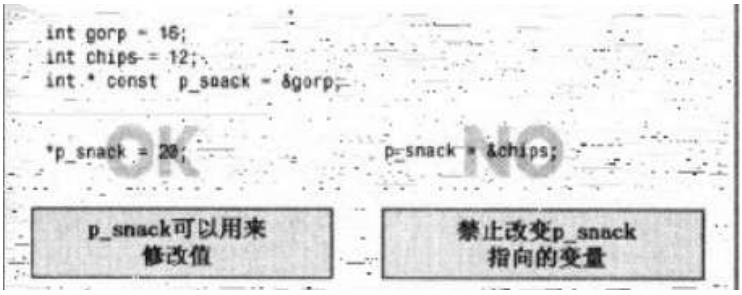
指针与const的关系：

```
int gorp = 16;
int chips = 12;
const int * p_snack = &gorp;

*p_snack = 20;          NO
p_snack = &chips;       OK
```

禁止修改p_snack指向的值

p_snack可以指向另一个变量



2，内存分配

```
int * pn = new int;
```

new int 告诉程序，需要适合存储 int 的内存。new 运算符根据类型来确定需要多少字节的内存。然后，它找到这样的内存，并返回其地址。接下来，将地址赋给 pn，pn 是被声明为指向 int 的指针。现在，pn 是地址，而*pn 是存储在那里的值。将这种方法与将变量的地址赋给指针进行比较：

对于指针，需要指出的另一点是，new 分配的内存块通常与常规变量声明分配的内存块不同。变量 nights 和 pd 的值都存储在被称为栈（stack）的内存区域中，而 new 从被称为堆（heap）或自由存储区（free store）的内存区域分配内存。第 9 章将更详细地讨论这一点。

函数内部局部变量储存在栈中；static静态变量也储存在栈中。

```
delete ps;          //free memory with delete when done
```

这将释放 ps 指向的内存，但不会删除指针 ps 本身。例如，可以将 ps 重新指向另一个新分配的内存块。

表 9.1　5 种变量储存方式

存储描述	持续性	作用域	链接性	如何声明
自动	自动	代码块	无	在代码块中
寄存器	自动	代码块	无	在代码块中，使用关键字 register
静态，无链接性	静态	代码块	无	在代码块中，使用关键字 static
静态，外部链接性	静态	文件	外部	不在任何函数内
静态，内部链接性	静态	文件	内部	不在任何函数内，使用关键字 static

3，new和delete准则

总之，使用 new 和 delete 时，应遵守以下规则。

- 不要使用 delete 来释放不是 new 分配的内存。
- 不要使用 delete 释放同一个内存块两次。
- 如果使用 new [ ]为数组分配内存，则应使用 delete [ ]来释放。
- 如果使用 new [ ]为一个实体分配内存，则应使用 delete（没有方括号）来释放。
- 对空指针应用 delete 是安全的。

除了基本的new函数，new头文件中还有new的其他形式，可以在指定的地址或缓冲区内分配内存：

就像常规 new 调用一个接收一个参数的 new()函数一样，标准定位 new 调用一个接收两个参数的 new()函数：

```
int * pi = new int;             // invokes new(sizeof(int))
int * p2 = new(buffer) int;     // invokes new(sizeof(int), buffer)
int * p3 = new(buffer) int[40]; // invokes new(40*sizeof(int), buffer)
```

4，头文件

头文件中通常包含以下内容，一般不将函数定义和变量声明放在头文件中：

- 函数原型。
- 使用#define 或 const 定义的符号常量。
- 结构声明。
- 类声明。
- 模板声明。
- 内联函数。

```
namespace elements
{
    namespace fire
    {
        int flame;
        ...
    }
    float water;
}
```

在同一个文件中只能将同一个头文件包含一次。记住这个规则很容易，但很可能在不知情的情况下将头文件包含多次。例如，可能使用包含了另外一个头文件的头文件。有一种标准的 C/C++技术可以避免多次包含同一个头文件。它是基于预处理器编译指令#ifndef（即 if not defined）的。下面的代码片段意味着仅当以前没有使用预处理器编译指令#define 定义名称 COORDIN_H_时，才处理#ifndef 和#endif 之间的语句：

```
#pragma once
#ifndef COORDIN_H_
#define COORDIN_H_
    ` ` ` ` `
#endif
```

5，命名空间

命名空间可以是全局的，也可以位于另一个命名空间之内，但不能在代码块中。用namespace关键字声明，通过..运算符来使用某个命名空间中的变量或函数。

using 声明使一个名称可用，而 using 编译指令使所有的名称都可用。using 编译指令由名称空间名和它前面的关键字 using namespace 组成，它使名称空间中的所有名称都可用，而不需要使用作用域解析运算符：

一般说来，使用 using 声明比使用 using 编译指令更安全，这是由于它只导入指定的名称。如果该名称与局部名称发生冲突，编译器将发出指示。using 编译指令导入所有名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

随着程序员逐渐熟悉名称空间，将出现统一的编程理念。下面是当前的一些指导原则。

- 使用在已命名的名称空间中声明的变量，而不是使用外部全局变量。
- 使用在已命名的名称空间中声明的变量，而不是使用静态全局变量。
- 如果开发了一个函数库或类库，将其放在一个名称空间中。事实上，C++当前提倡将标准函数库放在名称空间 std 中。这种做法扩展到了来自 C 语言中的函数。例如，头文件 math.h 是与 C 语言兼容的，没有使用名称空间，但 C++头文件 cmath 应将各种数学库函数放在名称空间 std 中。实际上，并非所有的编译器都完成了这种过渡。
- 仅将编译指令 using 作为一种将旧代码转换为使用名称空间的权宜之计。
- 不要在头文件中使用 using 编译指令。首先，这样做掩盖了要让哪些名称可用；另外，包含头文件的顺序可能影响程序的行为。如果非要使用编译指令 using，应将其放在所有预处理器编译指令#include 之后。
- 导入名称时，首选使用作用域解析运算符或 using 声明的方法。
- 对于 using 声明，首选将其作用域设置为局部而不是全局。

3，C++ I/O

1，文件操作

1，文件结尾检查

检测到 EOF 后，cin 将两位（eofbit 和 failbit）都设置为 1。可以通过成员函数 eof( )来查看 eofbit 是否被设置：如果检测到 EOF，则 cin.eof( )将返回 bool 值 true，否则返回 false。同样，如果 eofbit 或 failbit 被设置为 1，则 fail( )成员函数返回 true，否则返回 false。注意，eof( )和 fail( )方法报告最近读取的结果；也就是说，它们在事后报告，而不是预先报告。因此应将 cin.eof( )或 cin.fail( )测试放在读取后，程序清单 5.18 中的设计体现了这一点。它使用的是 fail( )，而不是 eof( )，因为前者可用于更多的实现中。

```
cin.get(ch);//读取一个字符之后开始判断是否有EOF
     while (!cin.fail())
     {
          cout << ch;
          ++count;
          cin.get(ch);
     }
```

其他变种：

```
while (cin.get(ch))  // while input is successful
{
    ... -            // do stuff
}

while (cin)    // while input is successful
```

这比! cin.fail( )或!cin.eof( )更通用，因为它可以检测到其他失败原因，如磁盘故障。

2，文件写入

- 必须包含头文件 fstream。
- 头文件 fstream 定义了一个用于处理输出的 ofstream 类。
- 需要声明一个或多个 ofstream 变量（对象），并以自己喜欢的方式对其进行命名，条件是遵守常用的命名规则。
- 必须指明名称空间 std，例如，为引用元素 ofstream，必须使用编译指令 using 或前缀 std::。
- 需要将 ofstream 对象与文件关联起来。为此，方法之一是使用 open( )方法。
- 使用完文件后，应使用方法 close( )将其关闭。
- 可结合使用 ofstream 对象和运算符<<来输出各种类型的数据。

```
ofstream outFile;
     outFile.open("info.txt",8);//打开文件,追加模式
     cout << "enter the make and model of automobile:";
     cin >> str;
     cout << "enter the model year:";
     cin >> year;
     cout << "enter the origial asking price:";
     cin >> a_price;
     //使用ofstream输出到文件
     outFile << fixed;//
     outFile.precision(2);//
     outFile.setf(ios_base::showpoint);//
     outFile << "make and model:" << str << endl;
     outFile << "year:" <<year << endl;
     outFile << "was asking $" << a_price << endl;
     outFile.close();
```

3，文件读取

- 必须包含买文件 fstream。
- 头文件 fstream 定义了一个用于处理输入的 ifstream 类。
- 需要声明一个或多个 ifstream 变量（对象），并以自己喜欢的方式对其进行命名，条件是遵守常用的命名规则。
- 必须指明名称空间 std：例如，为引用元素 ifstream，必须使用编译指令 using 或前缀 std::。
- 需要将 ifstream 对象与文件关联起来。为此，方法之一是使用 open( )方法。
- 使用完文件后，应使用 close( )方法将其关闭。
- 可结合使用 ifstream 对象和运算符>>来读取各种类型的数据。
- 可以使用 ifstream 对象和 get( )方法来读取一个字符，使用 ifstream 对象和 getline( )来读取一行字符。
- 可以结合使用 ifstream 和 eof( )、fail( )等方法来判断输入是否成功。
- ifstream 对象本身被用作测试条件时，如果最后一个读取操作成功，它将被转换为布尔值 true，否则被转换为 false。

按字节读取：

```
ifstream inFile;
    inFile.open("info.txt", 1);
    char ch;
    if (inFile.is_open()) {
        while (inFile.get(ch)) {
            cout << ch;//会将换行符打印出来
        }
        inFile.close();
    }
```

按行读取：

```
ifstream inFile;
    inFile.open("info.txt", ios_base::in);
    string ch;
    if (inFile.is_open()) {
        while (inFile) {
            getline(inFile, ch);//一次读取一行
            cout << ch << endl;//手动打印换行符
        }
        inFile.close();
    }
```

防止ifstream读取文件时最后一行读取两次的方法，在while语句最后加上：

```
in.get(); // 读取最后的回车符
if(in.peek() == '\n') break;
```

4，文件打开模式

表 17.8　C++和 C 的文件打开模式

C++模式	C 模式	含　义
ios_base :: in	"r"	打开以读取
ios_base :: out	"w"	等价于 ios_base :: out | ios_base :: trunc
ios_base :: out | ios_base :: trunc	"w"	打开以写入，如果已经存在，则截短文件
ios_base :: out | ios_base :: app	"a"	打开以写入，只追加
ios_base :: out | ios_base :: out	"r+"	打开以读写，在文件允许的位置写入
ios_base :: out | ios_base :: out | ios_base::trunc	"w+"	打开以读写，如果已经存在，则首先截短文件
c++mode | ios_base :: binary	"cmodeb"	以 C++mode（或相应的 cmode）和二进制模式打开；例如，ios_base :: in | ios_base :: binary 成为 "rb"
c++mode | ios_base :: ate	"cmode"	以指定的模式打开，并移到文件尾。C 使用一个独立的函数调用，而不是模式编码。例如，ios_base :: in | ios_base :: ate 被转换为 "r" 模式和 C 函数调用 fseek(file, 0, SEEK_END)

上述代码也使用|运算符来合并模式，因此 ios_base::out | ios_base::app 意味着启用模式 out 和 app

注意，ios_base::ate 和 ios_base::app 都将文件指针指向打开的文件尾。二者的区别在于，ios_base::app 模式只允许将数据添加到文件尾，而 ios_base::ate 模式将指针放到文件尾。

5，二进制文件读写

二进制的存取使用fstream中的write和read方法，这两种方法可以直接将一种数据类型以二进制方式写入文件，读取时再使用相同的类型和相同的长度即可读取到一样的数据：

```
struct planet
{
    char name[20];
```

```
        double population;
        double g;
    };
    //添加新数据到文件
        ofstream fout;
        fout.open(file, ios_base::out | ios_base::app | ios_base::binary);
        if (!fout.is_open()) {
            cout << "该文件无法打开成功" << endl;
            return -1;
        }
        cout << "Enter planet name(Enter a blan line to quit):" << endl;
        cin.get(pl.name, 20);
        while (pl.name[0] != '\0')
        {
            eatline();
            cout << "Enter planet population: ";
            cin >> pl.population;
            cout << "Enter planet acceleration og gravity:";
            cin >> pl.g;
            eatline();
            fout.write((char*)&pl, sizeof(pl));//写入文件
            //开始下一轮输入
            cout << "Enter planet name(Enter a blan line to quit):" << endl;
            cin.get(pl.name, 20);
        }
        fout.close();
    //显示二进制文件中的数据
        ifstream fin;
        fin.open(file, ios_base::in | ios_base::binary);//以二进制方式打开文件
        if (fin.is_open())
        {
            cout << "Here are the current contents of the " << file << " file:" <<endl;
            while (fin.read((char*)&pl,sizeof(pl)))//每次读取一个结构体的数据
            {
                cout << pl.name << ":"<< pl.population << pl.g << endl;
            }
            fin.close();
        }
```

6，随机存取

接下来，需要一种在文件中移动的方式。fstream 类为此继承了两个方法：seekg( )和 seekp( )，前者将输入指针移到指定的文件位置，后者将输出指针移到指定的文件位置（实际上，由于 fstream 类使用缓冲区来存储中间数据，因此指针指向的是缓冲区中的位置，而不是实际的文件）。也可以将 seekg( )用于 ifstream 对象，将 seekp( )用于 oftream 对象。下面是 seekg( )的原型：

```
basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);
basic_istream<charT,traits>& seekg(pos_type);
```

```
fin.seekg(30, ios_base::beg);     // 30 bytes beyond the beginning
fin.seekg(-1, ios_base::cur);     // back up one byte
fin.seekg(0, ios_base::end);      // go to the end of the file
```

如果要检查文件指针的当前位置，则对于输入流，可以使用 tellg( )方法，对于输出流，可以使用 tellp( )方法。它们都返回一个表示当前位置的 streampos 值（以字节为单位，从文件开始处算起）。创建 fstream 对象时，输入指针和输出指针将一前一后地移动，因此 tellg( )和 tellp( )返回的值相同。然而，如果使用 istream 对象来管理输入流，而使用 ostream 对象来管理同一个文件的输出流，则输入指针和输出指针将彼此独立地移动，因此 tellg( )和 tellp( )将返回不同的值。

```
//2,让用户选择要修改的记录
    cout << "输入你要修改的记录索引号：";
    int indexChanged;
    cin >> indexChanged;
    eatline();
    streampos pos = indexChanged*sizeof(pl);//计算要修改记录的位置
    finout.seekg(pos);
    finout.read((char*)&pl, sizeof(pl));
    cout << "你要修改的是这条记录吧？ (Y/N)" << endl;
    cout<< pl.name << "  " <<pl.population << "  " <<pl.g << endl;
    if (cin.get() == 'y') {
        eatline();
        cout << "Enter planet name:" << endl;
        cin.get(pl.name, 20);
        eatline();
        cout << "Enter planet population: ";
```

```
        cin >> pl.population;
        cout << "Enter planet acceleration og gravity:";
        cin >> pl.g;
        //开始修改
        finout.seekp(pos);
        finout.write((char*)&pl, sizeof(pl));
    }
```

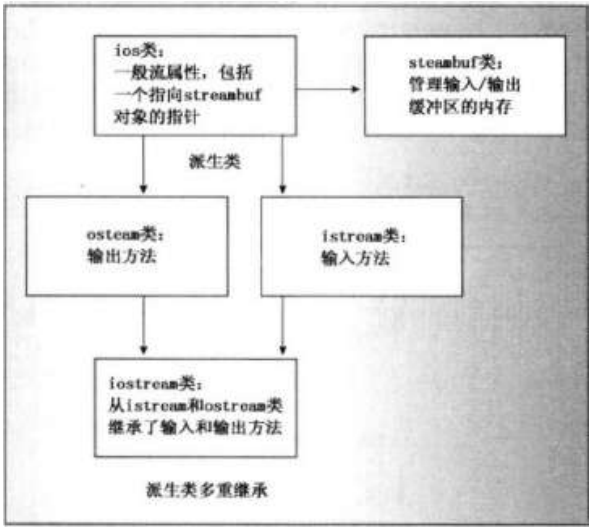
7，临时文件

先，需要为临时文件制定一个命名方案，但如何确保每个文件都被指定了独一无二的文件名呢？cstdio 中声明的 tmpnam( )标准函数可以帮助您。

```
char* tmpnam( char* pszName );
```

tmpnam( )函数创建一个临时文件名，将它放在 pszName 指向的 C-风格字符串中。常量 L_tmpnam 和 TMP_MAX（二者都是在 cstdio 中定义的）限制了文件名包含的字符数以及在确保当前目录中不生成重复文件名的情况下 tmpnam( )可被调用的最多次数。下面是生成 10 个临时文件名的代码。

2，标准输出



- streambuf 类为缓冲区提供了内存，并提供了用于填充缓冲区、访问缓冲区内容、刷新缓冲区和管理缓冲区内存的类方法；
- ios_base 类表示流的一般特征，如是否可读取、是二进制流还是文本流等；
- ios 类基于 ios_base，其中包括了一个指向 streambuf 对象的指针成员；
- ostream 类是从 ios 类派生而来的，提供了输出方法；
- istream 类也是从 ios 类派生而来的，提供了输入方法；
- iostream 类是基于 istream 和 ostream 类的，因此继承了输入方法和输出方法。

1，格式化输出：

1，调整进制：cout提供了控制符dec，hex和oct分别用于指示cout以十进制，十六进制，八进制格式显示整数：dec等控制符也是在std命名空间内

```
cout << "Monsieur cuts a striking figure!"  << endl;
cout << "chest = " << chest << " (decimal for 42)" << endl;
cout << hex;       // manipulator for changing number base
cout << "waist = " << waist << " (hexadecimal for 42)" << endl;
cout << oct;       // manipulator for changing number base
cout << "inseam = " << inseam << " (octal for 42)" << endl;
```

其他适用的控制符，可以起到和setf一样的效果：

控制符	调用
boolalpha	setf(ios_base::boolalpha)
noboolalpha	unset(ios_base::noboolalpha)
showbase	setf(ios_base::showbase)
noshowbase	unsetf(ios_base::showbase)
showpoint	setf(ios_base::showpoint)
noshowpoint	unsetf(ios_base::showpoint)
showpos	setf(ios_base::showpos)
noshowpos	unsetf(ios_base::showpos)
uppercase	setf(ios_base::uppercase)
nouppercase	unsetf(ios_base::uppercase)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::base-field)
hex	setf(ios_base::hex, ios_base::base-field)
oct	setf(ios_base::oct, ios_base::base-field)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)

2，调整字段宽度：width方法可以设置宽度，但是只影响接下来要显示的一个项目，然后会恢复默认值。

```
cout.width(5);
      cout << i << ":";
      cout.width(8);
      cout << i*i << endl;
```

3，填充字符：默认使用空格填充未被使用的字段，fill可以改变以使用其他字符，该方法会一直有效，除非手动变回默认。

```
cout.fill('*');
```

4，浮点数精度：precision方法可以设置显示的总位数（小数点前后），一直有效

5，打印小数点后隐藏的0：

```
cout.setf(ios_base::showpoint);
```

Setf其他可以使用的值和作用：

单一参数：

ios_base ::boolalpha	输入和输出 bool 值，可以为 true 或 false
ios_base ::showbase	对于输出，使用 C++基数前缀（0，0x）
ios_base ::showpoint	显示末尾的小数点
ios_base ::uppercase	对于 16 进制输出，使用大写字母，E 表示法
ios_base ::showpos	在正数前面加上+

两个参数：

第二个参数	第一个参数	含　义
ios_base ::basefield	ios_base ::dec	使用基数 10
	ios_base ::oct	使用基数 8
	ios_base ::hex	使用基数 16

ios_base ::floatfield	ios_base ::fixed	使用定点计数法
	ios_base ::scientific	使用科学计数法
	ios_base ::left	使用左对齐
		符号或基数前缀左对齐，值右对齐

6：iomanip头文件：可以使用里面的方法设置宽度精度等，与cout.width形式不一样

```
cout << setw(6) << "N" << setw(14) << "square root"
     << setw(15) << "fourth root\n";
```

2，其他方法

其他ostream方法：

除了各种 operator<<( )函数外，ostream 类还提供了 put( )方法和 write( )方法，前者用于显示字符，后者用于显示字符串。

Write第一个参数提供要显示的字符串的地址，第二个参数指出要显示多少个字符。

手动刷新缓冲区：

事实上，控制符也是函数。例如，可以直接调用 flush( )来刷新 cout 缓冲区：

```
flush(cout);
```

然而，ostream 类对<<插入运算符进行了重载，使得下述表达式将被替换为函数调用 flush(cout)：

```
cout << flush
```

2，标准输入

1，cin输入方法

1，getline()方法：读取一行，并丢弃换行符

cin.getline(name, size);——读取一行字符串放入name中

2，get()方法：读取下一个输入字符，即使是空格、制表符和换行符

下面方式与getline效果一样，不同的是会保存换行符，因此需要在中间加一个get()，作用是读取换行符并丢弃

```
cin.get(name, ArSize);      // read first line
cin.get();                  // read newline
cin.get(dessert, Arsize);   // read second line
```

3，ignore()方法：

程序清单 17.13 演示了 getline( )和 get( )是如何工作的，它还介绍了 ignore( )成员函数。该函数接受两个参数：一个是数字，指定要读取的最大字符数；另一个是字符，用作输入分界符。例如，下面的函数调用读取并丢弃接下来的 255 个字符或直到到达第一个换行符：

```
cin.ignore(255, '\n');
```

Cin默认的识别模式：

```
char philosophy[20];
int distance;
char initial;

cin >> philosophy >> distance >> initial;
```

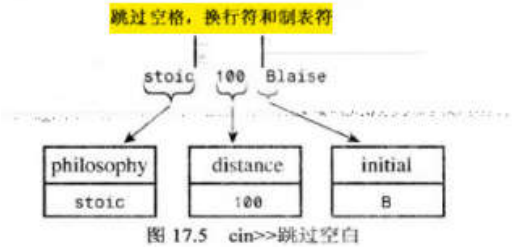


图 17.5 cin>>跳过空白

2，输入流状态

只有在流状态良好（所有的位都被清除）情况下，cin输入才正常，并且返回true。

表 17.4　　流状态

成　员	描　述
eofbit	如果到达文件尾，则设置为 1
badbit	如果流被破坏，则设置为 1；例如，文件读取错误
failbit	如果输入操作未能读取预期的字符或输出操作没有写入预期的字符，则设置为 1
goodbit	另一种表示 0 的方法
good( )	如果流可以使用（所有的位都被清除），则返回 true
eof( )	如果 eofbit 被设置，则返回 true
bad( )	如果 badbit 被设置，则返回 true
fail( )	如果 badbit 或 failbit 被设置，则返回 true
rdstate( )	返回流状态
exceptions( )	返回一个位掩码，指出哪些标记导致异常被引发
exceptions(iostate ex)	设置哪些状态将导致 clear( )引发异常；例如，如果 ex 是 eofbit，则如果 eofbit 被设置，clear( )将引发异常
clear(iostate s)	将流状态设置为 s；s 的默认值为 0（goodbit）；如果(rdstate( )& exceptions( ))! = 0，则引发异常 basic_ios::failure
setstate(iostate s)	调用 clear（rdstate( )| s），这将设置与 s 中设置的位对应的流状态位，其他流状态位保持不变

4，函数

1，函数参数

1，数组作为函数参数

```
int sum_arr(int * arr, int n) // arr = array name, n = size
```

其中用 int * arr 替换了 int arr [ ]。这证明这两个函数头都是正确的，因为在 C++中，当（且仅当）用于函数头或函数原型中，int *arr 和 int arr [ ]的含义才是相同的。它们都意味着 arr 是一个 int 指针。然而，数组表示法（int arr[ ]）提醒用户，arr 不仅指向 int，还指向 int 数组的第一个 int。当指针指向数组的第一个元素时，本书使用数组表示法；而当指针指向一个独立的值时，使用指针表示法。别忘了，在其他的上下文中，int * arr 和 int arr [ ]的含义并不相同。例如，不能在函数体中使用 int tip[ ]来声明指针。

注意，上面两种方式相当于都是创建了一个指向数组的指针作为参数，因此使用sizef并不能得到数组的长度，得到的是指针的长度（一般为4）。

2，二维数组作为函数参数

```
int sum(int ar2[][4],int size)
{
    int total = 0;
    for (int r = 0; r < size; r++)
        for (int c = 0; c < 4; c++)
```

3，结构体传址调用

假设要传递结构的地址而不是整个结构以节省时间和空间，则需要重新编写前面的函数，使用指向结构的指针。首先来看一看如何重新编写 show_polar( )函数。需要修改三个地方：

- 调用函数时，将结构的地址（&pplace）而不是结构本身（pplace）传递给它；
- 将形参声明为指向 polar 的指针，即 polar *类型。由于函数不应该修改结构，因此使用了 const 修饰符；
- 由于形参是指针而不是结构，因此应间接成员运算符（->），而不是成员运算符（句点）。

完成上述修改后，该函数如下所示：

```
// show polar coordinates, converting angle to degrees
void show_polar (const polar * pda)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << pda->distance;
    cout << ", angle = " << pda->angle * Rad_to_deg;
    cout << " degrees\n";
}
```

4，默认参数

对于带参数列表的函数，必须从右向左添加默认值。也就是说，要为某个参数设置默认值，则必须为它右边的所有参数提供默认值：

```
int harpo(int n, int m = 4, int j = 5);          // VALID
int chico(int n, int m = 6, int j);              // INVALID
int groucho(int k = 1, int m = 2, int n = 3);    // VALID
```

注意默认参数在函数原型中指定，在函数定义中不需要再次指定。

5，引用变量（和指针很像）

将引用变量组我欸参数，函数将使用原始数据，而不是副本，和指针传址调用一致。

前面讲过，C 和 C++使用&符号来指示变量的地址。C++给&符号赋予了另一个含义，将其用来声明引用。例如，要将 rodents 作为 rats 变量的别名，可以这样做：

```
int rats;
int & rodents = rats;    // makes rodents an alias for rats
```

其中，&不是地址运算符，而是类型标识符的一部分。就像声明中的 char*指的是指向 char 的指针一样，int &指的是指向 int 的引用。上述引用声明允许将 rats 和 rodents 互换——它们指向相同的值和内存单元。

传引用变量作为函数参数

```
void showTime1(Time & t) {
     cout << t.hours << " Hours" << t.mins << " minutes" <<
     endl;
}
showTime1(t1);
```

对比：传址调用

```
void showTime(Time * t) {
     cout << t->hours << " Hours"<< t->mins << " minutes" <<
     endl;
}
showTime(&t1);
```

引用变量在声明的时候必须初始化，并且初始化之后不能再改变，不能通过赋值的方式再指定引用对象。

下面是将引用用于类对象的示例，是最简单有效的方式：

```
//最简单的设计
string version1(const string & s1, const string & s2) {
     string tmp;
     tmp = s2 + s1 + s2;
     return tmp;
}
```

对于使用传递的值而不作修改的函数。

- 如果数据对象很小，如内置数据类型或小型结构，则按值传递。
- 如果数据对象是数组，则使用指针，因为这是唯一的选择，并将指针声明为指向 const 的指针。
- 如果数据对象是较大的结构，则使用 const 指针或 const 引用，以提高程序的效率。这样可以节省复制结构所需的时间和空间。
- 如果数据对象是类对象，则使用 const 引用。类设计的语义常常要求使用引用，这是 C++新增这项特性的主要原因。因此，传递类对象参数的标准方式是按引用传递。

对于修改调用函数中数据的函数：

- 如果数据对象是内置数据类型，则使用指针。如果看到诸如 fixit(&x)这样的代码（其中 x 是 int），则很明显，该函数将修改 x。
- 如果数据对象是数组，则只能使用指针。
- 如果数据对象是结构，则使用引用或指针。
- 如果数据对象是类对象，则使用引用。

2，函数指针

函数名就代表函数的开始位置，是函数指针。

```
double pam(int);  // prototype
```

则正确的指针类型声明如下：

```
double (*pf)(int);   // pf points to a function that takes
                     // one int argument and that
                     // returns type double
```

这与 pam( )声明类似，这是将 pam 替换为了（*pf）。由于 pam 是函数，因此（*pf）也是函数。而如果*pf）是函数，则 pf 就是函数指针。

使用函数指针调用函数：

```
double pam(int);
double (*pf)(int);
pf = pam;          // pf now points to the pam() function
double x = pam(4); // call pam() using the function name
double y = (*pf)(5); // call pam() using the pointer pf
```

实际上，C++也允许像使用函数名那样使用 pf：

```
double y = pf(5);  // also call pam() using the pointer pf
```

3，内联函数

C++内联函数提供了另一种选择。内联函数的编译代码与其他程序代码“内联”起来了。也就是说，编译器将使用相应的函数代码替换函数调用。对于内联代码，程序无需跳到另一个位置处执行代码，再跳回来。因此，内联函数的运行速度比常规函数稍快，但代价是需要占用更多内存。如果程序在 10 个不同的地方调用同一个内联函数，则该程序将包含该函数代码的 10 个副本（参见图 8.1）。

要使用这项特性，必须采取下述措施之一：

- 在函数声明前加上关键字 inline；
- 在函数定义前加上关键字 inline。

通常的做法是省略原型，将整个定义（即函数头和所有函数代码）放在本应提供原型的地方。

4，函数重载

函数重载的关键是函数的参数列表——也称为函数特征标（function signature）。如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而变量名是无关紧要的。C++允许定义名称相同的函数，条件是它们的特征标不同。如果参数数目和/或参数类型不同，则特征标也不同。例

注意几点：

1. 类型引用和类型本身视为同一个特征标
2. const指针和常规指针是不同的特征标
3. 返回值类型不同并不能作为重载的依据
4.

虽然函数重载很吸引人，但也不要滥用。仅当函数基本上执行相同的任务，但使用不同形式的数据时，才应采用函数重载。另外，您可能还想知道，是否可以通过使用默认参数来实现同样的目的。例如，可以用两个重载函数来代替面向字符串的 left( )函数：

```
char * left(const char * str, unsigned n);   // two arguments
char * left(const char * str);               // one argument
```

5，函数模板

1，函数模板

函数模板是通用的函数描述，使用泛型来定义函数，其中的泛型可以用具体的类型来替换。通过将类型作为参数传递给模板，可以使编译器生成该类型的函数，也称为通用编程。

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

第一行指出，要建立一个模板，并将类型命名为 AnyType。关键字 template 和 typename 是必需的，除非可以使用关键字 class 代替 typename。另外，必须使用尖括号。类型名可以任意选择（这里为 AnyType）。

```
template <typename T>
void Swap(T &a, T &b) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

注意：并不是所有的参数都必须是泛型类型，可以有基本类型参数

2，具体化模板函数

对于复杂的数据类型，函数实现逻辑和基本的类型可能会有所不同，因此需要针对复杂类型特别定义函数逻辑，成为具体化模板函数。函数的三种形式和优先级如下：

- 对于给定的函数名，可以有非模板函数、模板函数和显式具体化模板函数以及它们的重载版本
- 显式具体化的原型和定义应以 template<>打头，并通过名称来指出类型。
- 具体化优先于常规模板，而非模板函数优先于具体化和常规模板。

非模板函数：
```
void Swap(Job &j1, Job &j2)
```
模板函数：
```
template <typename T>
void Swap(T &a, T &b)
```
具体化模板函数：
```
template<>
void Swap(Job &j1, Job &j2)
```
示例：
```
template<>
void Swap(Job &j1, Job &j2) {
      double t1;
      int t2;
      t1 = j1.salary;
      j1.salary = j2.salary;
      j2.salary = t1;

      t2 = j1.floor;
      j1.floor = j2.floor;
      j2.floor = t2;
}
```
C++11 新增的关键字 decltype 提供了解决方案。可这样使用该关键字：
```
int x;
decltype(x) y;   // make y the same type as x
```
给 decltype 提供的参数可以是表达式，因此在前面的模板函数 ft()中，可使用下面的代码：
```
decltype(x + y) xpy;  // make xpy the same type as x + y
xpy = x + y;
```

5，面向对象

1，Class基础

1，class创建

```
class className
{
private:
    data member declarations
public:
    member function prototypes
};
```

- 定义成员函数时，使用作用域解析运算符（::）来标识函数所属的类；
- 类方法可以访问类的 private 组件。

在类声明中实现的函数会自动成为内联函数，需要注意这一点。

在C++中使用类时，类名+对象名声明之后就可以直接使用，不需要和java一样使用new。如果使用new则赋值的就不是对象名了，而是指针，如果使用了new就必须提供析构函数用于释放对象时的操作。

C++提供了两种使用构造函数来初始化对象的方式。第一种方式是显式地调用构造函数：
```
Stock food = Stock("World Cabbage", 250, 1.25);
```
这将 food 对象的 company 成员设置为字符串“World Cabbage”，将 shares 成员设置为 250，依此类推。

另一种方式是隐式地调用构造函数：
```
Stock garment("Furry Mason", 50, 2.5);
```

在 C++ 中，如果一个类有只有一个参数的构造函数，C++ 允许一种特殊的声明类变量的方式。在这种情况下，可以直接将一个对应于构造函数参数类型的数据直接赋值给类变量，编译器在编译时会自动进行类型转换，将对应于构造函数参数类型的数据转换为类的对象。如果在构造函数前加上explicit 修饰词，则会禁止这种自动

转换

在函数前使用explicit关键字（一般用于构造函数）可以关闭C++自动的隐式类型转换，但是允许显示类型转换。需要注意：类中this是指针，因此访问类成员时需要使用->运算符，*this则使用.运算符。

```
void show() const;          // promises not to change invoking object
```

同样，函数定义的开头应像这样：

```
void stock::show() const    // promises not to change invoking object
```

以这种方式声明和定义的类函数被称为const 成员函数。就像应尽可能将 const 引用和指针用作函数形参一样，只要类方法不修改调用对象，就应将其声明为 const。从现在开始，我们将遵守这一规则。

类中的static静态属性的赋值也要放在源文件中，头文件中仅仅定义变量名和类型即可。

2，特殊成员函数

C++类提供了下面的特殊成员函数：

- 默认构造函数，如果没有定义构造函数；
- 默认析构函数，如果没有定义；
- 复制构造函数，如果没有定义；
- 赋值运算符，如果没有定义；
- 地址运算符，如果没有定义。

复制构造函数用于将一个对象复制到新创建的对象中。也就是说，它用于初始化过程中（包括按值传递参数），而不是常规的赋值过程中。类的复制构造函数原型通常如下：

```
Class_name(const Class_name &);
```

对于复制构造函数，按值传递参数和函数返回值时都会使用复制构造函数。默认提供的复制构造函数的操作是逐个复制类非静态成员属性。

与复制构造函数相似，赋值运算符的隐式实现也对成员进行逐个复制。如果成员本身就是类对象，则程序将使用为这个类定义的赋值运算符来复制该成员，但静态数据成员不受影响。

```
Class_name & Class_name::operator=(const Class_name &);
```

3，指针和对象

2，运算符重载

运算符重载需要使用成为运算符函数的特殊函数形式，格式如下：

返回类型 Operator 运算符（参数列表）

示例：

头文件： `Time operator +(const Time & t) const;`
源文件： `Time Time::operator+(const Time & t) const {`

```
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours +
    t.hours+sum.minutes/60;
    sum.minutes %= 60;
    return sum;
}
```

运算符重载的限制：

1. 重载后的运算符必须至少有一个操作数是用户定义的类型
2. 使用运算符时不能违反运算符原来的句法规则，和优先级
3. 不能使用C++不存在的运算符，不能创建新运算符
4. 仅可以重载下面的运算符

表 11.1 可重载的运算符

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

5，下面的运算符重载时只能通过成员函数进行重载，不能使用友元函数：

- =：赋值运算符。
- ()：函数调用运算符。
- []：下标运算符。
- ->：通过指针访问类成员的运算符。

3，友元

仅通过public方法访问类的私有部分，在某些情况下不适合特定的编程问题，所以还有另一种形式的访问权限：友元。**友元分为：友元函数，友元类，友元成员函数**。

1，友元函数

让函数成为友元，可以使该函数与类的成员函数具有相同的访问权限。

创建友元函数的第一步是将其原型放在类声明中，并在原型声明前加上关键字 friend：

```
friend Time operator*(double m, const Time & t);  // goes in class declaration
```

该原型意味着下面两点：

- 虽然 operator *( )函数是在类声明中声明的，但它不是成员函数，因此不能使用成员运算符来调用；
- 虽然 operator *( )函数不是成员函数，但它与成员函数的访问权限相同。

```
Time operator*(double m, const Time & t)
{
    return t * m;      // use t.operator*(m)
}
```

提示：一般来说，要重载<<运算符来显示 c_name 的对象，可使用一个友元函数，其定义如下：

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ;  // display object contents
    return os;
}
```

2，友元类

友元类的所有方法都可以访问原始类的私有成员和保护乘员，提高了公有接口的灵活性。友元类声明的位置无关紧要，在什么位置都可以，但要注意两个类的顺序：

```
class Tv
{
public:
    friend class Remote;   // Remote can access Tv private parts
    enum {Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};

class Remote
{
private:
    int mode;
```

3，友元成员函数

一个类中也许并不是所有的方法都需要访问另一个类的成员，可能仅仅是其中几个方法，这时候将整个类声明为友元就有些多余，可以只将需要的方法声明为友元，同时这几个方法又是类的成员函数，所以称为友元成员函数。
需要注意的是头文件声明时对两个类的顺序由要求（向前声明），如下：

```
class Tv;                // forward declaration
class Remote { ... };     // Tv-using methods as prototypes only
class Tv { ... };

friend void Remote::set_chan(Tv & t, int c);
```

4，转换函数

要将其他类型转化为类类型可以使用构造函数完成，如果要将类类型转换为其他类型就需要实现转换函数自定义转换的规则。转换函数的格式如下：

```
operator typeName();
```

请注意以下几点：

- 转换函数必须是类方法；
- 转换函数不能指定返回类型；
- 转换函数不能有参数。

例如，转换为 double 类型的函数的原型如下：

```
operator double();
```

示例：

头文件：

```
/转换函数
    operator int()const;
```

源文件：

```
Time::operator int()const {
    return hours * 60 + minutes;
}
```

调用：

```
int result = int(total);
```

5，继承

1，公有继承

C++中继承分为三种：公有继承，私有继承，保护继承；公有继承的格式如下：

```
// RatedPlayer derives from the TableTennisPlayer base class
class RatedPlayer : public TableTennisPlayer
{
...
};
```

子类不能直接访问父类的私有成员，仅可以通过public方法来操作私有成员，protected成员在子类中可以访问。

有关派生类构造函数的要点如下：

- 首先创建基类对象；
- 派生类构造函数应通过成员初始化列表将基类信息传递给基类构造函数；
- 派生类构造函数应初始化派生类新增的数据成员。

```
derived::derived(type1 x, type2 y) : base(x,y) // initializer list
{
    ...
}
```

另外，抽象基类类似java中的接口Interface，子类必须实现父类里的虚函数，而且抽象基类不能创建对象，只能使用其子类创建对象。

友元函数不是类的成员，因此不能给继承。

2，虚函数

正如前面介绍的，如果在基类中没有将 ViewAcct( )声明为虚的，则 bp->ViewAcct( )将根据指针类型（Brass *）调用 Brass::ViewAcct( )。指针类型在编译时已知，因此编译器在编译时，可以将 ViewAcct( )关联到 Brass::ViewAcct( )。总之，编译器对非虚方法使用静态联编。

然而，如果在基类中将 ViewAcct( )声明为虚的，则 bp->ViewAcct( )根据对象类型（BrassPlus）调用 BrassPlus::ViewAcct( )。在这个例子中，对象类型为 BrassPlus，但通常（如程序清单 13.10 所示）只有在运行程序时才能确定对象的类型。所以编译器生成的代码将在程序执行时，根据对象类型将 ViewAcct( )关联到 Brass::ViewAcct( )或 BrassPlus::ViewAcct( )。总之，编译器对虚方法使用动态联编。

简而言之，如果要在派生类中重新定义基类的方法，就将它设置为虚方法，这样在多态调用的时候，编译器就知道要去调用子类的方法实现，而不会去调用父类原本的实现。

注意事项：

- 构造函数不能是虚函数
- 西沟函数应当是虚函数，除非基类不被使用
- 友元不能是虚函数，因为友元不是类的成员
- 如果在子类中重新定义了虚函数，则父类的虚函数会被覆盖

表 13.1 成员函数属性

函数	能否继承	成员还是友元	默认能否生成	能否为虚函数	是否可以有返回类型
构造函数	否	成员	能	否	否
析构函数	否	成员	能	能	否
=	否	成员	能	能	能
&	能	任意	能	能	能
转换函数	能	成员	否	能	否
()	能	成员	否	能	能
[]	能	成员	否	能	能
->	能	成员	否	能	能
op=	能	任意	否	能	能
new	能	静态成员	否	否	void*
delete	能	静态成员	否	否	void
其他运算符	能	任意	否	能	能
其他成员	能	成员	否	能	能
友元	否	友元	否	否	能

3，私有继承/保护继承

C++还有另一种实现 has-a 关系的途径——私有继承。使用私有继承，基类的公有成员和保护成员都将成为派生类的私有成员。这意味着基类方法将不会成为派生对象公有接口的一部分，但可以在派生类的成员函数中使用它们。

私有继承和在一个类中包含另一个类的效果一样：获得了实现，无法获得接口。一般建议使用包含，而不是私有继承。

使用保护继承时，基类的公有成员和保护成员都将成为派生类的保护成员。和私有私有继承一样，基类的接口在派生类中也是可用的，但在继承层次结构之外是不可用的。当从派生类派生出另一个类时，私有继承和保护继承之间的主要区别便呈现出来了。使用私有继承时，第三代类将不能使用基类的接口，这

表 14.1 各种继承方式

特征	公有继承	保护继承	私有继承
公有成员变成	派生类的公有成员	派生类的保护成员	派生类的私有成员
保护成员变成	派生类的保护成员	派生类的保护成员	派生类的私有成员
私有成员变成	只能通过基类接口访问	只能通过基类接口访问	只能通过基类接口访问
能否隐式向上转换	是	是（但只能在派生类中）	否

4，多重继承

MI 描述的是有多个直接基类的类。与单继承一样，公有MI 表示的也是 is-a 关系。例如，可以从 Waiter 类和 Singer 类派生出 SingingWaiter 类：

```
class SingingWaiter : public Waiter, public Singer {...};
```

请注意，必须使用关键字 public 来限定每一个基类。这是因为，除非特别指出，否则编译器将认为是私有派生：

6，模板类

1，模板类

模板类就是类似vector和array，本身是一个类，同时可以接收类型作为参数，生成特定类型的容器。

采用模板时，将使用模板定义替换 Stack 声明，使用模板成员函数替换 Stack 的成员函数。和模板函数一样，模板类以下面这样的代码开头：

```
template <class Type>
```

关键字 template 告诉编译器，将要定义一个模板。尖括号中的内容相当于函数的参数列表。可以把关键字 class 看作是变量的类型名，该变量接受类型作为其值，把 Type 看作是该变量的名称。

这里使用 class 并不意味着 Type 必须是一个类；而只是表明 Type 是一个通用的类型说明符，在使用模板时，将使用实际的类型替换它。较新的 C++实现允许在这种情况下使用不太容易混淆的关键字 typename 代替 class：

```
template <typename Type>  // newer choice
```

类模板的另一项新特性是，可以为类型参数提供默认值：

```
template <class T1, class T2 = int> class Topo {...};
```

这样，如果省略 T2 的值，编译器将使用 int：

需要注意：模板成员函数要和类定义放在一个文件里，不能再放在单独的文件里。另外，template这一行指定的参数除了泛型参数，还可以有其他类型的参数。

```
#pragma once
//定义模板类stack
template <typename Type>
class StackType {
private:
	enum {MAX=10};
	Type items[MAX];
	int top;
public:
	StackType();
	bool isEmpty() const;
	Type pop();
};
//模板成员函数要在同一个文件里
template <typename Type>
StackType<Type>::StackType() {
	top = 0;
}
template <typename Type>
bool StackType<Type>::isEmpty() const {
	return top == 0;
}
template <typename Type>
Type StackType<Type>::pop() {
	if (top>0)
	{
		return items[--top];
	}
	else
	{
		return nullptr;
	}
}
```

2，具体化模板类

具体化类模板定义的格式如下：

```
template <> class Classname<specialized-type-name> { ... };
```

早期的编译器可能只能识别早期的格式，这种格式不包括前缀 template<>：

```
class Classname<specialized-type-name> { ... };
```

要使用新的表示法提供一个专供 const char *类型使用的 SortedArray 模板，可以使用类似于下面的代码：

```
template <> class SortedArray<const char char *>
{
    ...// details omitted
};
```

C++还允许部分具体化（partial specialization），即部分限制模板的通用性。例如，部分具体化可以给类型参数之一指定具体的类型：

```
// general template
    template <class T1, class T2> class Pair {...};
// specialization with T2 set to int
    template <class T1> class Pair<T1, int> {...};
```

关键字 template 后面的<>声明的是没有被具体化的类型参数。因此，上述第二个声明将 T2 具体化为 nt，但 T1 保持不变。注意，如果指定所有的类型，则<>内将为空，这将导致显式具体化：

6，异常处理

1，异常处理和exception

下面介绍如何使用异常机制来处理错误。C++异常是对程序运行过程中发生的异常情况（例如被 0 除）的一种响应。异常提供了将控制权从程序的一个部分传递到另一部分的途径。对异常的处理有 3 个组成部分：

- 引发异常；
- 使用处理程序捕获异常；
- 使用 try 块。

引发异常并捕获

```
double hmean2(double a, double b) {
    if (a == -b)
    {
        throw "had hmean() arguments:a=-b not allowed";//使用简单文本，没有用exception类，这里也可以使用其他类型或自定义的类，只要和catch可以对应即可
    }
    return 2.0*a*b / (a + b);
}

    try
        {
            z = hmean2(x, y);
        }
        catch (const char * s)
        {
            cout << s << endl;
        }
```

Exception类位于exception头文件中，作为异常类的基类，有一个名为what()的虚成员函数，返回一个字符串。库实现的异常类和用户自定义的异常类都要实现what方法。

```
#include <exception>
class bad_hmean : public std::exception
{
public:
    const char * what() { return "bad arguments to hmean()"; }
...
};
class bad_gmean : public std::exception
{
public:
    const char * what() { return "bad arguments to gmean()"; }
...
};
```

如果不想以不同的方式处理这些派生而来的异常，可以在同一个基类处理程序中捕获它们

```
try {
...
}
catch(std::exception & e)
{
    cout << e.what() << endl;
...
}
```

否则，可以分别捕获它们。

2，库常见异常类

Stdexcept头文件：

- Domain_error：参数取值或函数返回值不在可能的范围内
- Invalid_argument：函数参数值为不合法
- Length_error：没有足够的空间执行操作，一般是长度过长
- Out_of_bounds：index索引越界
- Range_error：计算结果不在允许范围内，但没有溢出
- Overflow_error：数值上溢
- Underflow_error：数值下溢

New头文件：

- Bad_alloc：无法分配请求的内存量，请求失败

7，STL标准模板库

1，string

1，创建string

表 16.1　　string 类的构造函数

构 造 函 数	描　　述
string(const char * s)	将 string 对象初始化为 s 指向的 NBTS
string(size_type n, char c)	创建一个包含 n 个元素的 string 对象，其中每个元素都被初始化为字符 c
string(const string & str)	将一个 string 对象初始化为 string 对象 str（复制构造函数）
string( )	创建一个默认的 sting 对象，长度为 0（默认构造函数）
string(const char * s, size_type n)	将 string 对象初始化为 s 指向的 NBTS 的前 n 个字符，即使超过了 NBTS 结尾
template<class Iter> string(Iter begin, Iter end)	将 string 对象初始化为区间[begin, end)内的字符，其中 begin 和 end 的行为就像指针，用于指定位置，范围包括 begin 在内，但不包括 end
string(const string & str, string size_type pos = 0, size_type n = npos)	将一个 string 对象初始化为对象 str 中从位置 pos 开始到结尾的字符，或从位置 pos 开始的 n 个字符
string(string && str) noexcept	这是 C++11 新增的，它将一个 string 对象初始化为 string 对象 str，并可能修改 str（移动构造函数）
string(initializer_list<char> il)	这是 C++11 新增的，它将一个 string 对象初始化为初始化列表 il 中的字符

对于类，很有帮助的另一点是，知道有哪些输入方式可用。对于 C-风格字符串，有 3 种方式：

```
char info[100];
cin >> info;              // read a word
cin.getline(info, 100);   // read a line, discard \n
cin.get(info, 100);       // read a line, leave \n in queue
```

对于 string 对象，有两种方式：

```
string stuff;
cin >> stuff;             // read a word
getline(cin, stuff);      // read a line, discard \n
```

两个版本的 getline( )都有一个可选参数，用于指定使用哪个字符来确定输入的边界：

```
cin.getline(info,100,':');     // read up to :, discard :
getline(stuff, ':');           // read up to :, discard :
```

2，常用函数

Size()和length()都返回字符串的字符数，功能完全一样，length是较老的方法。

表 16.2　　重载的 find( )方法

方 法 原 型	描　述
size_type find(const string & str, size_type pos = 0)const	从字符串的 pos 位置开始，查找子字符串 str。如果找到，则返回该子字符串首次出现时其首字符的索引；否则，返回 string :: npos
size_type find(const char * s, size_type pos = 0)const	从字符串的 pos 位置开始，查找子字符串 s。如果找到，则返回该子字符串首次出现时其首字符的索引；否则，返回 string :: npos
size_type find(const char * s, size_type pos= 0, size_type n)	从字符串的 pos 位置开始，查找 s 的前 n 个字符组成的子字符串。如果找到，则返回该子字符串首次出现时其首字符的索引；否则，返回 string :: npos
size_type find(char ch, size_type pos = 0)const	从字符串的 pos 位置开始，查找字符 ch。如果找到，则返回该字符首次出现的位置；否则，返回 string :: npos

string 库还提供了相关的方法：rfind( )、find_first_of( )、find_last_of( )、find_first_not_of( )和 find_last_not_of( )，它们的重载函数特征标都与 find( )方法相同。rfind( )方法查找子字符串或字符最后一次出现的位置；find_first_of( )方法在字符串中查找参数中任何一个字符首次出现的位置。例如，下面的语句返

2，智能指针

智能指针在memory头文件中，目前有3种指针模板，其中auto_ptr比较老。

这三个智能指针模板（auto_ptr、unique_ptr 和 shared_ptr）都定义了类似指针的对象，可以将 new 获得（直接或间接）的地址赋给这种对象。当智能指针过期时，其析构函数将使用 delete 来释放内存。因此，如果将 new 返回的地址赋给这些对象，将无需记住稍后释放这些内存；在智能指针过期时，这些内存将自动被释放。

```
auto_ptr<double> pd(new double);  // pd an auto_ptr to double
                                  // (use in place of double * pd)
auto_ptr<string> ps(new string);  // ps an auto_ptr to string
                                  // (use in place of string * ps)
```

new double 是 new 返回的指针，指向新分配的内存块。它是构造函数 auto_ptr<double>的参数，即对应于原型中形参 p 的实参。同样，new string 也是构造函数的实参。其他两种智能指针使用同样的语法：

```
unique_ptr<double> pdu(new double);  // pdu an unique_ptr to double
shared_ptr<string> pss(new string);  // pss a shared_ptr to string
```

所有智能指针类都一个 explicit 构造函数，该构造函数将指针作为参数。因此不需要自动将指针转换为智能指针对象：

```
shared_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                          // not allowed (implicit conversion)
pd = shared_ptr<double>(p_reg);      // allowed (explicit conversion
shared_ptr<double> pshared = p_reg; // not allowed (implicit conversion)
shared_ptr<double> pshared(p_reg);  // allowed (explicit conversion)
```

三种智能指针的区别：

- 建立所有权（ownership）概念，对于特定的对象，只能有一个智能指针可拥有它，这样只有拥有对象的智能指针的构造函数会删除该对象。然后，让赋值操作转让所有权。这就是用于 auto_ptr 和 unique_ptr 的策略，但 unique_ptr 的策略更严格。
- 创建智能更高的指针，跟踪引用特定对象的智能指针数。这称为引用计数（reference counting）。例如，赋值时，计数将加 1，而指针过期时，计数将减 1。仅当最后一个指针过期时，才调用 delete。这是 shared_ptr 采用的策略。

警告：使用 new 分配内存时，才能使用 auto_ptr 和 shared_ptr，使用 new [ ]分配内存时，不能使用它们。不使用 new 分配内存时，不能使用 auto_ptr 或 shared_ptr；不使用 new 或 new []分配内存时，不能使用 unique_ptr。

3，标准容器模板

1，vector

以vector为例，介绍容器模板，下面方法大都适用于所有的容器模板：

要创建 vector 模板对象，可使用通常的<type>表示法来指出要使用的类型。另外，vector 模板使用动态内存分配，因此可以用初始化参数来指出需要多少矢量：

```
#include vector
using namespace std;
vector<int> ratings(5);       // a vector of 5 ints
int n;
cin >> n;
vector<double> scores(n);     // a vector of n doubles
```

由于运算符[ ]被重载，因此创建 vector 对象后，可以使用通常的数组表示法来访问各个元素：

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
    cout << scores[i] << endl;
```

除分配存储空间外，vector 模板还可以完成哪些任务呢？所有的 STL 容器都提供了一些基本方法，其中包括 size( )——返回容器中元素数目、swap( )——交换两个容器的内容、begin( )——返回一个指向容器中第一个元素的迭代器、end( )——返回一个表示超过容器尾的迭代器。

erase( )方法删除矢量中给定区间的元素。它接受两个迭代器参数，这些参数定义了要删除的区间。了解 STL 如何使用两个迭代器来定义区间至关重要。第一个迭代器指向区间的起始处，第二个迭代器位于区间终止处的后一个位置。例如，下述代码删除第一个和第二个元素，即删除 begin( )和 begin( )+1 指向的元素（由于 vector 提供了随机访问功能，因此 vector 类迭代器定义了诸如 begin( )+2 等操作）：

```
scores.erase(scores.begin(), scores.begin() + 2);
```

insert( )方法的功能与 erase( )相反。它接受 3 个迭代器参数，第一个参数指定了新元素的插入位置，第二个和第三个迭代器参数定义了被插入区间，该区间通常是另一个容器对象的一部分。例如，下面的代码

Random_shuffle( )函数接受两个指定区间的迭代器参数，并随机排列该区间中的元素。例如，下面的语句随机排列 books 矢量中所有元素：

```
random_shuffle(books.begin(), books.end());
```

与可用于任何容器类的 for_each 不同，该函数要求容器类允许随机访问，vector 类可以做到这一点。

sort( )函数也要求容器支持随机访问。该函数有两个版本，第一个版本接受两个定义区间的迭代器参数，并使用为存储在容器中的类型元素定义的<运算符，对区间中的元素进行操作。例如，下面的语句按升序

而，如果想按降序或是按 rating（而不是 title）排序，该如何办呢？可以使用另一种格式的 sort( )。它接受 3 个参数，前两个参数也是指定区间的迭代器，最后一个参数是指向要使用的函数的指针（函数对象），而不是用于比较的 operator<( )。返回值可转换为 bool，false 表示两个参数的顺序不正确。下面是一个例子：

```
bool WorseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

有了这个函数后，就可以使用下面的语句将包含 Review 对象的 books 矢量按 rating 升序排列：

```
sort(books.begin(), books.end(), WorseThan);
```

2，其他容器种类

STL 具有容器概念和容器类型。概念是具有名称（如容器、序列容器、关联容器等）的通用类别；容器类型是可用于创建具体容器对象的模板。以前的 11 个容器类型分别是 deque、list、queue、priority_queue、stack、vector、map、multimap、set、multiset 和 bitset（本章不讨论 bitset，它是在比特级处理数据的容器）；C++11 新增了 forward_list、unordered_map、unordered_multimap、unordered_set 和 unordered_multiset，且

表 16.5　　一些基本的容器特征

表 达 式	返 回 类 型	说　明	复 杂 度
X :: iterator	指向 T 的迭代器类型	满足正向迭代器要求的任何迭代器	编译时间
X :: value_type	T	T 的类型	编译时间
X u;		创建一个名为 u 的空容器	固定
X( );		创建一个匿名的空容器	固定
X u(a);		调用复制构造函数后 u == a	线性
X u = a;		作用同 X u(a);	线性
r = a;	X&	调用赋值运算符后 r == a	线性
(&a)->~X( )	void	对容器中每个元素应用析构函数	线性
a.begin( )	迭代器	返回指向容器第一个元素的迭代器	固定
a.end( )	迭代器	返回超尾值迭代器	固定
a.size( )	无符号整型	返回元素个数，等价于 a.end( )- a.begin( )	固定
a.swap(b)	void	交换 a 和 b 的内容	固定
a == b	可转换为 bool	如果 a 和 b 的长度相同，且 a 中每个元素都等于（==为真）b 中相应的元素，则为真	线性
a != b	可转换为 bool	返回!(a== b)	线性

表 16.6　　C++11 新增的基本容器要求

表 达 式	返 回 类 型	说　明	复 杂 度
X u(rv);		调用移动构造函数后，u 的值与 rv 的原始值相同	线性
X u = rv;		作用同 X u(rv);	
a = rv;	X&	调用移动赋值运算符后，u 的值与 rv 的原始值相同	线性
a.cbegin( )	const_iterator	返回指向容器第一个元素的 const 迭代器	固定
a.cend( )	const_iterator	返回超尾值 const 迭代器	固定

3，键值关联容器

STL 提供了 4 种关联容器：set、multiset、map 和 multimap。前两种是在头文件 set（以前分别为 set.h 和 multiset.h）中定义的，而后两种是在头文件 map（以前分别为 map.h 和 multimap.h）中定义的。

最简单的关联容器是 set，其值类型与键相同，键是唯一的，这意味着集合中不会有多个相同的键。确实，对于 set 来说，值就是键。multiset 类似于 set，只是可能有多个值的键相同。例如，如果键和值的类型为 int，则 multiset 对象包含的内容可以是 1、2、2、2、3、5、7、7。

在 map 中，值与键的类型不同，键是唯一的，每个键只对应一个值。multimap 与 map 相似，只是一个键可以与多个值相关联。

无序关联容器是对容器概念的另一种改进。与关联容器一样，无序关联容器也将值与键关联起来，并使用键来查找值。但底层的差别在于，关联容器是基于树结构的，而无序关联容器是基于数据结构哈希表的，这旨在提高添加和删除元素的速度以及提高查找算法的效率。有 4 种无序关联容器，它们是 unordered_set、unordered_multiset、unordered_map 和 unordered_multimap，将在附录 G 更详细地介绍。

4，函数符

概念不多解释，类似android中事件监听器，使用一个函数符作为参数，指定要进行的操作，具体的操作在一个函数中。

对于所有内置的算术运算符、关系运算符和逻辑运算符，STL 都提供了等价的函数符。表 16.12 列出了这些函数符的名称。它们可以用于处理 C++内置类型或任何用户定义类型（如果重载了相应的运算符）。

表 16.12　　运算符和相应的函数符

运　算　符	相应的函数符
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

4，算法库

STL 将算法库分成 4 组：

- 非修改式序列操作；
- 修改式序列操作；
- 排序和相关操作；
- 通用数字运算。

前 3 组在头文件 algorithm（以前为 algo.h）中描述，第 4 组是专用于数值数据的，有自己的头文件，称为 numeric（以前它们也位于 algol.h 中）。

非修改式序列操作对区间中的每个元素进行操作。这些操作不修改容器的内容。例如，find( )和 for_each( )就属于这一类。

修改式序列操作也对区间中的每个元素进行操作。然而，顾名思义，它们可以修改容器的内容。可以修改值，也可以修改值的排列顺序。transform( )、random_shuffle( )和 copy( )属于这一类。

排序和相关操作包括多个排序函数（包括 sort( )）和其他各种函数，包括集合操作。

数字操作包括将区间的内容累积、计算两个容器的内部乘积、计算小计、计算相邻对象差的函数。通常，这些都是数组的操作特性，因此 vector 是最有可能使用这些操作的容器。