数据的输入和输出

2016年9月23日　　10:15

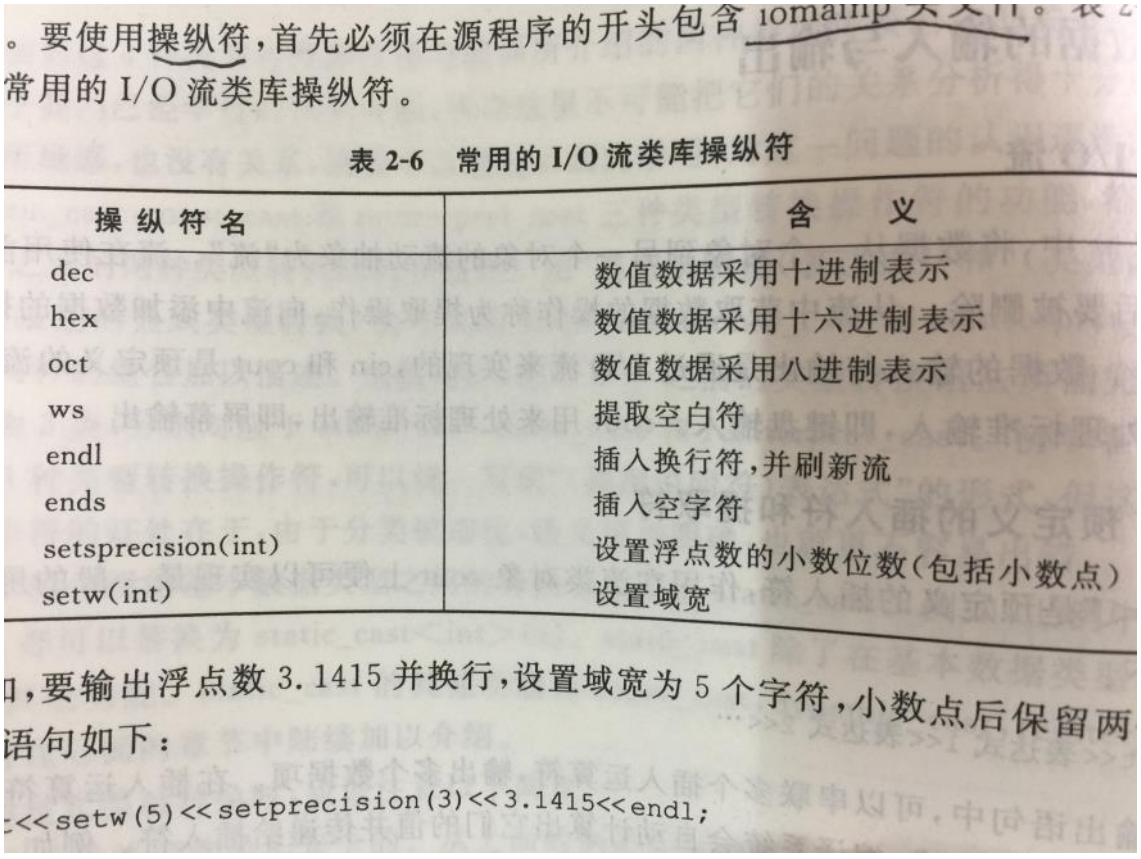
I/O流：

- 预定义的流类对象cin、cout
- 预定义的插入符<<和提取符>>

cout<<表达式1<<表达式2<<…;
cin>>表达式1>>表达式2>>…;　//以空格为分隔符

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

- 操纵符（包含在iomainp头文件中）



。要使用操纵符，首先必须在源程序的开头包含 iomainp 头文件。表 2
常用的 I/O 流类库操纵符。

表 2-6　常用的 I/O 流类库操纵符

操纵符名	含　义
dec	数值数据采用十进制表示
hex	数值数据采用十六进制表示
oct	数值数据采用八进制表示
ws	提取空白符
endl	插入换行符，并刷新流
ends	插入空字符
setsprecision(int)	设置浮点数的小数位数（包括小数点）
setw(int)	设置域宽

，要输出浮点数 3.1415 并换行，设置域宽为 5 个字符，小数点后保留两
语句如下：

```
<<setw(5)<<setprecision(3)<<3.1415<<endl;
```

C++程序的一般组织结构：
类定义文件（*.h）
类实现文件（*.cpp）
类的使用文件（*.cpp，主函数文件）

决定一个声明的位置的一般原则是：将需要分配空间的定义放在.cpp文件

中

- 内联函数应该写在.h文件中
- 外部函数和外部变量的定义应该写在.h文件中

位运算符

2016年9月23日　　10:09

按位与**&**
按位或|
按位异或**^**
按位取反**~**
移位**<<** 、 **>>**

参数传递

2016年9月23日　　11:05

- 值传递：直接将实参的值传递给形参，一旦形参获得了值便于形参脱离关系
- 引用传递：
  int I,j;
  int &ri=I;//ri是i的一个别名，改变ri的同时也改变了i
  j=10;
  ri=j;//相当于i=j
  声明一个引用必须同时对它进行初始化，使它指向一个已存在的对象，且无法更改。

- 带默认形参值的函数
  int add(int a, int x=5, int y=6) /*有默认值的形参必须在形参列表的最后*/
  {…}

函数重载

2016年9月24日　　11:29

两个以上的函数具有相同的函数名，但形参个数或类型不同，编译器根据实参和形参的类型、个数的最佳匹配，自动确定调用哪个函数。

- 重载函数的形参必须不同，且注意默认形参的二义性
- 重载函数是服务于在相同作用域内功能相近的函数，方便使用和记忆

C++系统函数

2016年9月24日　　15:16

http://www.cppreference.com C++函数的原型、头文件及用法

基本概念

2016年9月26日　　9:02

- 抽象：对具体问题（对象）进行概括，抽出一类对象的公共性质并加以描述的过程。
  数据抽象（变量）、功能抽象（函数）

- 封装：将抽象得到的数据和行为相结合，形成一个有机整体“类”，气冲的数据和函数就是这个“类”的成员。

- 继承：允许在保持原有类特性的基础上进行更具体、详细的说明。

- 多态：一段程序能够处理多种类型对象的能力。
  强制多态、重载多态、类型参数化多态、包含多态

类&对象

2016年9月26日 9:35

类的定义

```
class 类名称
{
public:
    外部接口
protected:
    保护型成员
private:
    私有型成员
};
```

类成员的访问控制

对类成员访问权限的控制是通过设置成员的访问控制属性而实现的

- 公有类型：在类外只能访问类的公有成员
- 私有类型：只能被本类的成员函数访问，来自类外部的任何访问都是非法的。（一个类的数据成员都应该声明为私有成员）
- 保护类型：与私有类型成员的性质相似，差别在与继承过程中对产生的新类影响不同

对象

1. 声明：类名　对象名;
2. 访问：　对象名 . 成员名
   　　对象指针 -> 成员名

类和对象的关系就相当于基本数据类型与它的变量的关系

类的成员函数

1. 成员函数的实现：函数原型声明写在类体中，说明函数的参数表和返回值类型，函数的具体实现写在类定义之外。

   返回值类型　类名::函数成员名(参数表)

{
    函数体
}

2. 内联成员函数的声明

（1）隐式声明：函数体直接放在类体内

```
class clock
{
public:
    void showTime()
    {
        cout << hour << endl;
    }
};
```

（2）显示声明：用关键字inline

inline 返回值类型 类名::函数成员名(参数表)

{

    函数体

}

构造函数

2016年9月26日　　10:46

构造函数的作用：在对象被创建时利用特定的值构造对象，将对象初始化为一个特定的状态。

- 构造函数是类的成员函数
- 构造函数的函数名与类名相同
- 没有返回值
- 被声明为公有函数
- 在对象被创建时，构造函数被自动调用

复制构造函数：使用一个已经存在的对象去初始化同类的一个新对象

- 形参是本类的对象的引用

声明和定义：

```
class 类名
{
public:
    类名(形参表);
    类名(类名  &对象名);
    …
};

类名::类名(类名  &对象名)
{
    函数体
}
```

调用：

用类的一个对象初始化该类的另一个对象时，调用复制构造函数

```
int main()
{
    Point a(1,2);
    Point b(a);
    Point c=a;
    …
}
```

如果函数的形参是类的对象，调用函数时进行形参和实参结合时，调用复制构造函数

```
void f(Point p)
{
    cout<<p.getX()<<end;
}
int main()
{
    Point a(1,2);
    f(a);
    return 0;
}
```

如果函数的返回值是类的对象，函数执行完成返回调用者时，调用复制构造函数

```
Point g()
{
    Point a(1,2);
    return a;
}
int main()
{
    Point b;
    b=g();
    return 0;
}
```

析构函数

2016年9月26日　　14:41

作用：在对象的生存期即将结束的时候被自动调用，相应的内存空间被释放。

- 析构函数是类的成员函数
- 析构函数的函数名是类名前加上“~”
- 没有返回值
- 被声明为公有函数
- 在对象被删除时，析构函数被自动调用
- 不接受任何参数
- 可以是虚函数

类的组合

2016年9月26日 15:08

类的组合描述：类内嵌其他类的对象作为成员

当创建类的对象时，如果这个类具有内嵌对象成员，那么各个内嵌对象将首先被自动创建

组合类的构造函数定义形式：
类名::类名(形参表):内嵌对象1(形参表),内嵌对象2(形参表),...
{类的初始化}

组合类的构造函数的调用顺序：

1. 调用内嵌对象的构造函数，顺序按照内嵌对象在组合类的定义中出现的次序
2. 执行本类构造函数的函数体

组合类的析构函数的调用顺序和构造函数的相反

前向引用声明：适用在两个类相互引用的情况，在使用前向引用声明时，只能使用被声明的符号，而不能涉及类的任何细节

联合体&结构体与类的区别

2016年9月26日　　19:22

联合体

- 默认访问控制属性是公共类型的
- 联合体的各个对象成员不能由自定义的构造函数、析构函数、重载赋值运算符
- 不能继承，不支持多态

结构体

- 对于未指定访问控制属性的成员，在类中其访问控制属性为private，在结构体中为public

结构体适用场合：在程序中定义一些数据，目的是将一些不同类型的数据组合成一个整体，从而方便地保持数据。

类型转换操作符

2016年9月30日　　15:46

- **static_cast<新的数据类型>(变量名);** //将一种类型的变量转换成另一种数据类型的变量
- **reinterpret_cast<新的数据类型 *>(&变量名);** //将一种类型的指针转换为另一种类型的指针
- **const_cast<数据类型>(变量名);** //将数据类型中的const属性去除

UML简介

2016年9月26日　　17:00

UML是一种面向对象建模语言，用符号描述概念，概念间的关系描述为连接符号的线。

类图：由类和与之相关的各种静态关系共同组成的图形，展示软件建模的静态结构、类的内部结构以及其他类的关系。

UML类图

2016年9月26日　　16:30

示例

对象名:类名
数据成员 **[访问控制属性] 名称 [重数] [:类型] [=默认值] [{约束特征}]** 中括号的内容时可选的
函数成员 **[访问控制属性] 名称 [(参数表)] [:返回值类型] [{约束特征}]**

- 对象名要加下划线
- 访问控制属性：
  public　　+
  private　　-
  protected　#
- 参数表格式：**[方向] 名称: 类型=默认值**　　，方向指明参数时用于表示 in、out、inout
- 约束特征：用户对该数据成员性质约束的说明，例如“{只读}”

UML图的几种关系图：

1. **泛化（Generalization）**

【泛化关系】：是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。例如：老虎是动物的一种，即有老虎的特性也有动物的共性。

【箭头指向】：带三角箭头的实线，箭头指向父类

2. **实现（Realization）**

【实现关系】：是一种类与接口的关系，表示类是接口所有特征和行为的实现.

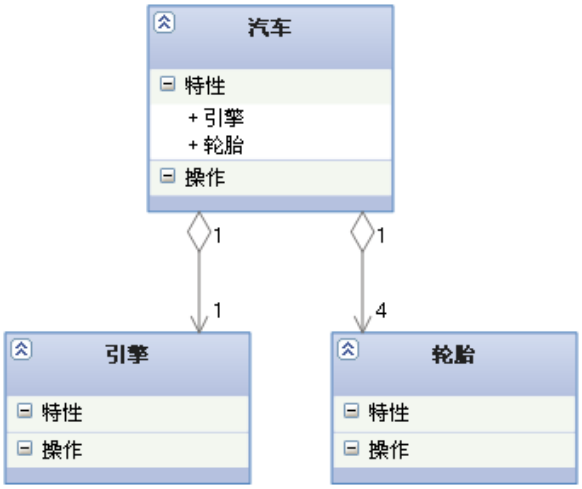
【箭头指向】：带三角箭头的虚线，箭头指向接口

3. **关联（Association）**

【关联关系】：是一种拥有的关系，它使一个类知道另一个类的属性和

方法；如：老师与学生，丈夫与妻子关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。
【代码体现】：成员变量
【箭头及指向】：带普通箭头的实心线，指向被拥有者

4. 聚合（Aggregation）
【聚合关系】：是整体与部分的关系，且部分可以离开整体而单独存在。如车和轮胎是整体和部分的关系，轮胎离开车仍然可以存在。聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。
【代码体现】：成员变量
【箭头及指向】：带空心菱形的实心线，菱形指向整体



5. 组合(Composition)
【组合关系】：是整体与部分的关系，但部分不能离开整体而单独存在。如公司和部门是整体和部分的关系，没有公司就不存在部门。
组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。
【代码体现】：成员变量
【箭头及指向】：带实心菱形的实线，菱形指向整体

6. 依赖(Dependency)
【依赖关系】：是一种使用的关系，即一个类的实现需要另一个类的协助，所以要尽量不使用双向的互相依赖.
【代码表现】：局部变量、方法的参数或者对静态方法的调用
【箭头及指向】：带箭头的虚线，指向被使用者

各种关系的强弱顺序：泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖

标识符的作用域和可见性

2016年9月26日　　20:52

局部作用域-类作用域-命名空间作用域

（内）————————————————（外）

- 如果在两个或多个具有包含关系的作用域中声明了同名标识符，则外层标识符在内层不可见
- 具有命名空间作用域的变量称为全局变量
- 作用域和可见性原则适合与变量名、常量名、自定义类型名、函数名、枚举类型的取值

对象的生存期

2016年9月27日　　8:47

- **static**静态生存期：对象的生存期与程序的运行期相同。当一个函数返回后，下次再调用时，该变量还会保持上一回的值，即使发生了递归调用，也不会为该变量建立新的副本。

- **动态生存期**

类的静态成员

2016年9月27日 9:14

- 静态成员：解决同一个类的不同对象直接数据和函数共享问题的。具有静态生存期。
- 类属性：描述类的所有对象共同特征的一个数据项，对于任何对象实例，它的属性值是相同的。

静态数据成员

声明：再类体内 **static 数据类型 变量名;**

```
class point
{
public:
    point(int x = 0, int y = 0) :x(x), y(y)
    {
        count++;
    }
    ~point(){ count--; }
    static int count;
};
```

定义和初始化：再类体外部 **数据类型 类名::变量名=初值;**

某个属性是整个类所共有的，不属于任何一个具体对象

```
int point::count = 0;
```

调用：

```
int main(void)
{
    int *p = &point::count;
    point a(2, 4);
}
```

静态函数成员

声明：**static 返回值类型 函数名(参数表);**

```
static void showCount();
```

定义：**返回值类型 函数名(参数表){...}**

```
void point::showCount()
{
    cout << "Object count=" << count << endl;
}
```

调用：对静态成员函数的调用是没有目的对象的，因此不能像非静态成员函数那样，隐含地通过目的对象访问类的非静态成员。

类名::函数名(参数表);

```
point::showCount();
```

访问：可以直接访问该类的静态数据和函数成员，而访问非静态成员必须通过对象名。

static的功能

- 隐藏
  Static可以用作函数和变量的前缀。可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲突。对于函数来讲，static的作用仅限于隐藏。

- 保持变量内容的持久
  存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量。

- 默认初始化为0

类的友元

2016年9月27日　　9:49

友元提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。

- ◇ 友元关系是不能传递的
- ◇ 友元关系是单向的：如果B类是A类的友元，只能是B类的成员函数访问A类的私有和保护数据。
- ◇ 友元关系是不被继承的：如果B类是A类的友元，B类的派生类并不会自动成为A类的友元

- 友元函数friend
  在函数体中可以通过对象名访问类的私有和保护成员
  声明：**friend 返回值类型 函数名(参数表);**
  定义：**返回值类型 函数名(参数表)**
  {
  函数体;
  }

- 友元类

```
class A
{
public:
    …
    friend class B;
    …
private:
    int x;
};

class B
{
public:
    void set(int i);
private:
    A a;
};
void B::set(int i)
{
    a.x=i;
}
```

B是A的友元类，所以可以在B的成员函数中可以访问A类对象的私有成员

共享数据的保护

2016年9月27日　　11:02

对于既需要共享又需要防止改变的数据应该声明为常量，常量在程序运行期间是不能被改变的。

- 常对象：**const 类型说明符 对象名;**
  - 它的数据成员值在对象的整个生存期间内不能被改变。
  - 常对象必须进行初始化，而且不能被更新(赋值)
  - 常对象只能调用它的常成员函数

- 常成员函数：**类型说明符 函数名(参数表) const;**
  - 常成员函数不能更新目的对象的数据成员

- 常数据成员：**const 变量类型 变量名;**
  - 在任何函数中都不能对常数据成员赋值
  - 构造函数对常数据成员进行初始化只能通过初始化列表

- 常引用：**const 类型说明符 &引用名;**
  - 通过常引用访问的对象只能被当作常对象
  - 常引用作为形参，则不会改变实参
  - 只有常引用可以传递常对象给函数

外部变量和函数

2016年9月27日　　16:38

外部变量是可以多个源文件共享的全局变量。使用时需要加上关键字“extern”
外部变量可以有多处声明，但只能定义一次。

外部函数

string

2017年4月25日 星期二 20:09

string类的构造函数：

```
string(const char *s);     //用c字符串s初始化
string(int n,char c);      //用n个字符c初始化
```

此外，string类还支持默认构造函数和复制构造函数，如string s1；string s2="hello"；都是正确的写法。当构造的string太长而无法表达时会抛出length_error异常

string类的字符操作：

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

operator[]和at()均返回当前字符串中第n个字符的位置，但at函数提供范围检查，当越界时会抛出out_of_range异常，下标运算符[]不提供检查访问。

```
const char *data()const;//返回一个非null终止的c字符数组
const char *c_str()const;//返回一个以null终止的c字符串
int copy(char *s, int n, int pos = 0) const;//把当前串中以pos开始的n个字符拷贝到以s为起始位置的字符数组中，返回实际拷贝的数目
```

string的特性描述:

```
int capacity()const;     //返回当前容量（即string中不必增加内存即可存放的元素个数）
int max_size()const;     //返回string对象中可存放的最大字符串的长度
int size()const;         //返回当前字符串的大小
int length()const;        //返回当前字符串的长度
bool empty()const;         //当前字符串是否为空
void resize(int len,char c);//把字符串当前大小置为len，并用字符c填充不足的部分
```

string类的输入输出操作:

string类重载运算符operator>>用于输入，同样重载运算符operator<<用于输出操作。

函数getline(istream &in,string &s);用于从输入流in中读取字符串到s中，以换行符'\n'分开。

string的赋值：

```
string &operator=(const string &s);//把字符串s赋给当前字符串
```

string &assign(const char *s);//用c类型字符串s赋值

string &assign(const char *s,int n);//用c字符串s开始的n个字符赋值

string &assign(const string &s);//把字符串s赋给当前字符串

string &assign(int n,char c);//用n个字符c赋值给当前字符串

string &assign(const string &s,int start,int n);//把字符串s中从start开始的n个字符赋给当前字符串

string &assign(const_iterator first,const_itertor last);//把first和last迭代器之间的部分赋给字符串

string的连接：

string &operator+=(const string &s);//把字符串s连接到当前字符串的结尾

string &append(const char *s);　　//把c类型字符串s连接到当前字符串结尾

string &append(const char *s,int n);//把c类型字符串s的前n个字符连接到当前字符串结尾

string &append(const string &s);　　//同operator+=()

string &append(const string &s,int pos,int n);//把字符串s中从pos开始的n个字符连接到当前字符串的结尾

string &append(int n,char c);　　//在当前字符串结尾添加n个字符c

string &append(const_iterator first,const_iterator last);//把迭代器first和last之间的部分连接到当前字符串的结尾

string的比较：

bool operator==(const string &s1,const string &s2)const;//比较两个字符串是否相等

运算符">","<",">=","<=","!="均被重载用于字符串的比较；

int compare(const string &s) const;//比较当前字符串和s的大小

int compare(int pos, int n,const string &s)const;//比较当前字符串从pos开始的n个字符组成的字符串与s的大小

int compare(int pos, int n,const string &s,int pos2,int n2)const;//比较当前字符串从pos开始的n个字符组成的字符串与s中pos2开始的n2个字符组成的字符串的大小

int compare(const char *s) const;

int compare(int pos, int n,const char *s) const;

int compare(int pos, int n,const char *s, int pos2) const;

compare函数在>时返回1，<时返回-1，==时返回0

string的子串：

string substr(int pos = 0,int n = npos) const;//返回pos开始的n个字符

组成的字符串

string的交换：

```
void swap(string &s2);    //交换当前字符串与s2的值
```

string类的查找函数：

```
int find(char c, int pos = 0) const;//从pos开始查找字符c在当前字符串的位置
int find(const char *s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置
int find(const char *s, int pos, int n) const;//从pos开始查找字符串s中前n个字符在当前串中的位置
int find(const string &s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置
//查找成功时返回所在位置，失败返回string::npos的值

int rfind(char c, int pos = npos) const;//从pos开始从后向前查找字符c在当前串中的位置
int rfind(const char *s, int pos = npos) const;
int rfind(const char *s, int pos, int n = npos) const;
int rfind(const string &s,int pos = npos) const;
//从pos开始从后向前查找字符串s中前n个字符组成的字符串在当前串中的位置，成功返回所在位置，失败时返回string::npos的值

int find_first_of(char c, int pos = 0) const;//从pos开始查找字符c第一次出现的位置
int find_first_of(const char *s, int pos = 0) const;
int find_first_of(const char *s, int pos, int n) const;
int find_first_of(const string &s,int pos = 0) const;
//从pos开始查找当前串中第一个在s的前n个字符组成的数组里的字符的位置。查找失败返回string::npos

int find_first_not_of(char c, int pos = 0) const;
int find_first_not_of(const char *s, int pos = 0) const;
int find_first_not_of(const char *s, int pos,int n) const;
int find_first_not_of(const string &s,int pos = 0) const;
//从当前串中查找第一个不在串s中的字符出现的位置，失败返回string::npos

int find_last_of(char c, int pos = npos) const;
int find_last_of(const char *s, int pos = npos) const;
int find_last_of(const char *s, int pos, int n = npos) const;
int find_last_of(const string &s,int pos = npos) const;
```

```
int find_last_not_of(char c, int pos = npos) const;
int find_last_not_of(const char *s, int pos = npos) const;
int find_last_not_of(const char *s, int pos, int n) const;
int find_last_not_of(const string &s,int pos = npos) const;
//find_last_of和find_last_not_of与find_first_of和find_first_not_of相似，只不过是从后向前查找
```

string类的替换函数：

```
string &replace(int p0, int n0,const char *s);//删除从p0开始的n0个字符，然后在p0处插入串s
string &replace(int p0, int n0,const char *s, int n);//删除p0开始的n0个字符，然后在p0处插入字符串s的前n个字符
string &replace(int p0, int n0,const string &s);//删除从p0开始的n0个字符，然后在p0处插入串s
string &replace(int p0, int n0,const string &s, int pos, int n);//删除p0开始的n0个字符，然后在p0处插入串s中从pos开始的n个字符
string &replace(int p0, int n0,int n, char c);//删除p0开始的n0个字符，然后在p0处插入n个字符c
string &replace(iterator first0, iterator last0,const char *s);//把[first0，last0）之间的部分替换为字符串s
string &replace(iterator first0, iterator last0,const char *s, int n);//把[first0，last0）之间的部分替换为s的前n个字符
string &replace(iterator first0, iterator last0,const string &s);//把[first0，last0）之间的部分替换为串s
string &replace(iterator first0, iterator last0,int n, char c);//把[first0，last0）之间的部分替换为n个字符c
string &replace(iterator first0, iterator last0,const_iterator first, const_iterator last);//把[first0，last0）之间的部分替换成[first，last）之间的字符串
```

string类的插入函数：

```
string &insert(int p0, const char *s);
string &insert(int p0, const char *s, int n);
string &insert(int p0,const string &s);
string &insert(int p0,const string &s, int pos, int n);
//前4个函数在p0位置插入字符串s中pos开始的前n个字符
string &insert(int p0, int n, char c);//此函数在p0处插入n个字符c
iterator insert(iterator it, char c);//在it处插入字符c，返回插入后迭代器的位置
```

```
void insert(iterator it, const_iterator first, const_iterator last);//在it处插入[first，last）之间的字符
void insert(iterator it, int n, char c);//在it处插入n个字符c
```

string类的删除函数

```
iterator erase(iterator first, iterator last);//删除[first，last）之间的所有字符，返回删除后迭代器的位置
iterator erase(iterator it);//删除it指向的字符，返回删除后迭代器的位置
string &erase(int pos = 0, int n = npos);//删除pos开始的n个字符，返回修改后的字符串
```

string类的迭代器处理：

string类提供了向前和向后遍历的迭代器iterator，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。
用string::iterator或string::const_iterator声明迭代器变量，const_iterator不允许改变迭代的内容。常用迭代器函数有：

```
const_iterator begin()const;
iterator begin();                  //返回string的起始位置
const_iterator end()const;
iterator end();                     //返回string的最后一个字符后面的位置
const_iterator rbegin()const;
iterator rbegin();                  //返回string的最后一个字符的位置
const_iterator rend()const;
iterator rend();                    //返回string第一个字符位置的前面
```

rbegin和rend用于从后向前的迭代访问，通过设置迭代器
string::reverse_iterator,string::const_reverse_iterator实现

字符串流处理：

通过定义ostringstream和istringstream变量实现，<sstream>头文件中
例如：

```
    string input("hello,this is a test");
    istringstream is(input);
    string s1,s2,s3,s4;
    is>>s1>>s2>>s3>>s4;//s1="hello,this",s2="is",s3="a",s4="test"
    ostringstream os;
    os<<s1<<s2<<s3<<s4;
    cout<<os.str();
```

指向类的对象的指针

2016年9月29日　　10:30

声明：**类名 *对象指针名;**
初始化：**对象指针名=&对象名;**
访问对象成员：**对象指针名->成员名**

指向类的非静态成员的指针

2016年9月29日　　11:03

指向数据成员的指针：
定义：类型说明符 类名::*指针名;
初始化：指针名=& 类名::数据成员名;
调用：对象名.*类成员指针名;

指向函数成员的指针：
定义：类型说明符 (类名::*指针名)(参数表);
初始化：指针名=& 类名::函数成员名;
调用：(对象名.*类成员指针名)(参数表)

```
int main()
{
    Point a(4, 5);
    Point *p1 = &a;
    int(Point::*funcPtr)()const = &Point::getX;

    cout << (a.*funcPtr)() << endl;     //使用成员函数指针和对象名访问成员函数getX
    cout << a.getX() << endl;           //使用对象名访问成员函数getX
    cout << (p1->funcPtr)() << endl;    //使用成员函数指针和对象指针访问成员函数getX
    cout << p1->getX() << endl;         //使用对象指针访问成员函数getX

}
```

指针和引用

2016年9月30日　　15:28

普通指针可以被多次赋值，多次更改它所指向的对象；
引用只能在初始化时指定被引用的对象，之后无法更改。

操作	T类型的指针常量	对T类型的引用
定义并用v初始化	T *const p=&v	T &r=v
取v的值	*p	r
访问成员m	p->m	r.m
取v的地址	p	&r
取指针常量的地址	&p	can't

动态内存分配

2016年9月30日　　9:26

new 数据类型(初始化参数列表);
new 类型名[数组长度] ();
紫色部分可省略，表示不初始化；若括号内容缺省表示初始化0值。
如果内存申请成功，new运算返回一个指向新分配内存首地址的类型的指针，可以通过这个指针对堆对象进行访问；如果失败，则抛出异常

delete 指针名;
delete[] 数组名;
删除由new建立的对象，释放指针所指向的内存空间或数组所占用的空间

封装的动态数组

2016年9月30日　　9:56

vector<元素类型>数组对象名(数组长度,元素初值);
vector数组对象的名字表示的就是一个数组对象，而非数组的首地址

vector定义的数组对象有一个成员函数**size()**，返回数组的大小

加上头文件
#include <vector>

string类

2016年9月30日 10:31

string类型：
在<string.h>头中

构造函数原型：
string();
string(const string &rhs); //复制构造函数
string(const char *s); //用指针s所指向的字符串常量初始化string类的对象
string(const string &rhs, unsigned int pos, unsigned int n);
//将对象rhs中的串从位置pos开始取n个字符，用来初始化string类的对象
string(const char *s, unsigned int n);
//用指针s所指向的字符串中前n个字符初始化string类的对象
string(unsigned int n, char c);
//将参数c中的字符重复n次，用来初始化string类的对象

string类的常用成员函数介绍
string append(const char *s); //将字符串s添加在本串末尾
string assign(const char *s); //将s所指向的字符串赋值给本对象
unsigned int length() const; //返回串的长度（字符个数）
void swap(string &str); //将本串与str中的字符串进行交换
getline(cin, str, ','); //以逗号为分隔符将输入的字符存储在str字符串中

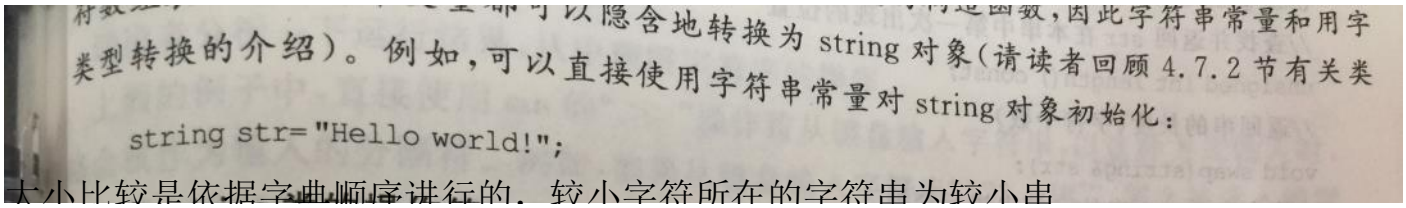
int compare(const string &str) const;
//比较本串与str串的大小，若本串<str串，返回负数

string & insert(unsigned int pos, const char *s);
//将s所指向的字符串插入在本串中位置pos之前

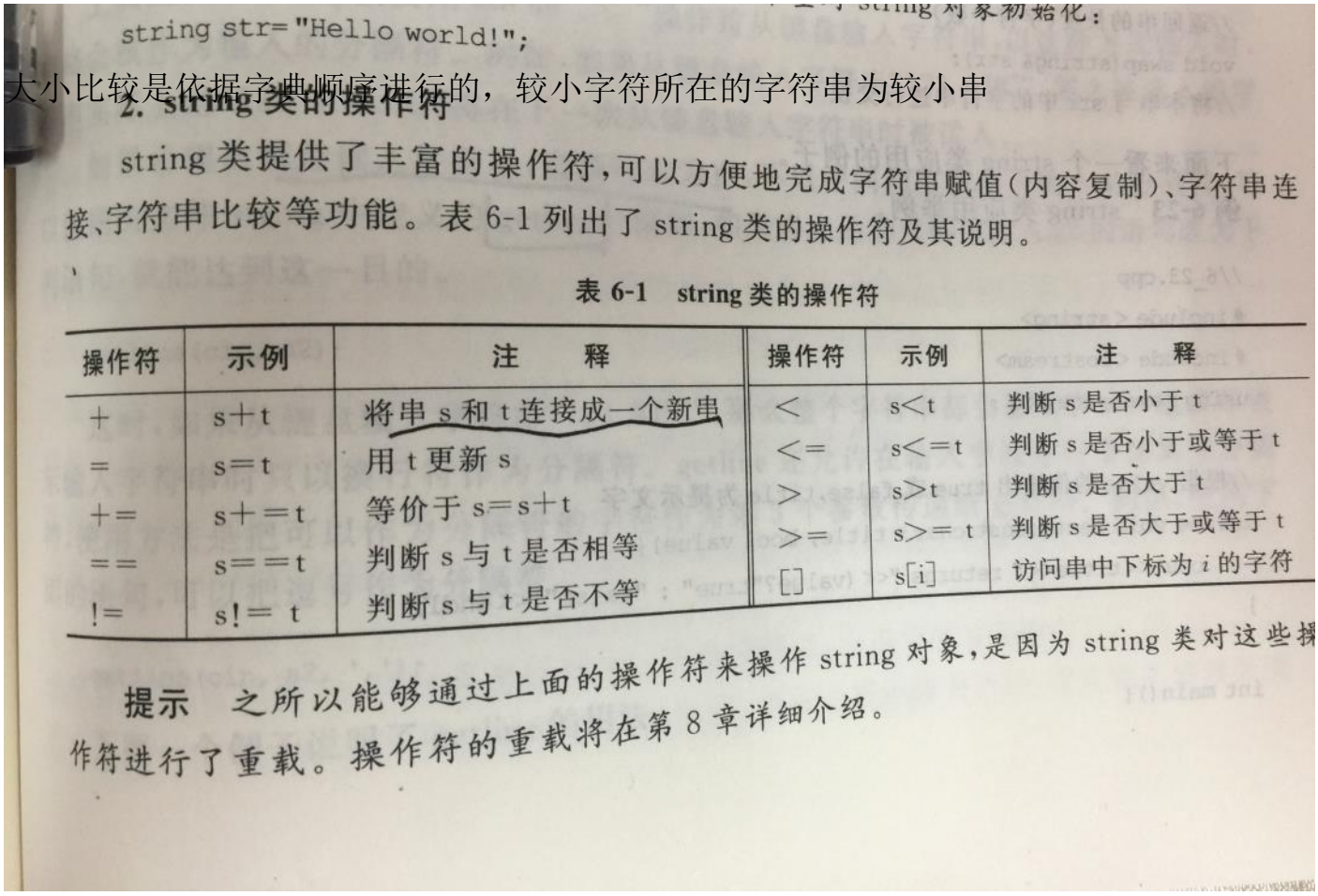
string substr(unsigened int pos, unsigned int n) const;
//取本串中位置pos开始的n个字符，构成新的string类对象作为返回值

unsigned int find(const basic_string &str) const;
//查找并返回str在本串中第一次出现的位置

string类的操作符



大小比较是依据字典顺序进行的，较小字符所在的字符串为较小串



```
string str="Hello world!";
```

大小比较是依据字典顺序进行的，较小字符所在的字符串为较小串

2. string类的操作符

string类提供了丰富的操作符，可以方便地完成字符串赋值(内容复制)、字符串连接、字符串比较等功能。表6-1列出了string类的操作符及其说明。

表6-1 string类的操作符

操作符	示例	注　释	操作符	示例	注　释
+	s+t	将串s和t连接成一个新串	<	s<t	判断s是否小于t
=	s=t	用t更新s	<=	s<=t	判断s是否小于或等于t
+=	s+=t	等价于s=s+t	>	s>t	判断s是否大于t
==	s==t	判断s与t是否相等	>=	s>=t	判断s是否大于或等于t
!=	s!=t	判断s与t是否不等	[]	s[i]	访问串中下标为i的字符

提示 之所以能够通过上面的操作符来操作string对象，是因为string类对这些操作符进行了重载。操作符的重载将在第8章详细介绍。

类的继承与派生

2016年9月30日　　16:04

类的继承：新的类从已有类那里得到已有的特性

定义：

```
class 派生类名:继承方式 基类名1, 继承方式 基类名2...
{
    派生类成员声明
}
```

- 继承方式规定了如何访问从基类继承的成员，默认的是private继承方式。
- 在派生的过程中，基类的构造函数和析构函数是不能被继承的。

公有继承

基类的公有成员和保护成员的访问属性在派生类中不变，基类的私有成员不能直接访问。类族外部只能通过派生类的对象访问从基类继承的公有成员

类型兼容规则：

公有继承使得在需要基类对象的地方，都可以用公有派生类的对象来替代。

- 派生类的对象可以隐含转换为基类对象
- 派生类的对象可以初始化基类的引用
- 派生类对象的地址可以隐含转换为基类的指针

私有继承

基类的公有成员和保护成员都以私有成员的身份出现在派生类中，基类的私有成员不能直接访问。类族外部不能访问从基类继承的任何成员。基类原有的所有外部接口都被派生类封装和隐藏起来了。

保护继承

基类的公有成员和保护成员都以保护成员的身份出现在派生类中，基类的私有成员不能直接访问。类族外部不能访问从基类继承的任何成员。

私有继承和保护继承的区别

答：区别在于当派生类1作为新的基类继续派生的时候。私有继承方式下，再次派生出的类2的成员和对象都无法访问派生类1从基类继承的成员。在保护继承方式下，再次派生出的类2有机会访问基类中的公有和保护成员。

继承与组合

2016年10月7日　　17:58

组合：如果类B中含有一个类A的内嵌对象，表示的是每一个B类型的对象都“有一个”A类的对象。B类对象虽然包括A类对象的全部内容，但一般A类对象是作为B类对象的私有成员存在的，隐藏了A类对象的对外接口，这些接口只能被B类所用，而不能直接作为B类的对外接口。**“整体-部分”**

公有继承：如果类A是类B的公有基类，表示的是每一个B类型的对象都“是一个”A类型的对象。B类对象包括A类对象的全部内容以及全部接口。**“一般-特殊”**

派生类的构造和析构函数

2016年10月6日　　9:46

派生类的构造函数

派生类的构造函数只负责对派生类新增的成员进行初始化。

构造派生类对象的流程：

1. 调用基类的构造函数来初始化它们的数据成员，调用顺序是按照派生类继承基类时定义的顺序。
2. 调用内嵌对象的构造函数，调用顺序按照成员在类中声明的顺序。
3. 执行派生类构造函数的函数体。

派生类名::派生类名(参数表): 基类名1(基类1初始化参数表),...,基类名n(基类n初始化参数表, 成员对象名1(成员对象1初始化参数表),...,成员对象名m(成员对象m初始化参数表)

{

派生类构造函数的其他初始化操作;

}

```
class derived: public base2, public base3, public base1
{
public:
    derived(int x, int y);

derived::derived(int x, int y) :base1(x), base2(y), member2(x - y), member1(x + y)
{
    cout << "constructing derived" << endl;
}
```

何时需要声明派生类的构造函数

在对基类初始化时，需要调用基类的带有形参表的构造函数时，派生类就必须声明构造函数，提供一个将参数传递给基类构造函数的途径，保证在基类进行初始化时能够获得必要的数据。

复制构造函数

派生类名::派生类名(派生类名 &对象名): 基类名1(派生类对象名),...

{

...

}

- 派生类的析构函数

  派生类析构函数的执行顺序：

  1. 执行派生类析构函数的函数体
  2. 调用派生类新增的类类型的成员对象所在类的析构函数
  3. 调用基类析构函数

派生类成员的标识与访问

2016年10月6日　　10:54

隐藏：派生类声明一个和基类成员同名的新成员（即使参数表不同），从基类继承的同名函数的所有重载形式也会被隐藏。如果要访问被隐藏的成员，就需要使用作用域分辨符和基类名来限定。

作用域分辨符

类名::成员名

类名::成员名(参数表)

- using

㈠. 解决派生类没有声明与基类同名的成员对象时，使用“对象名.成员名”，“对象指针->成员名”的方式访问。

```
class derived: public base1
{
public:
    using base1::var;
};
```

㈡. 将一个作用域中的名字引入到另一个作用域中

```
class derived: public base1
{
public:
    using base1::fun;
    void fun(int x) {…}
};
/*这样使用derived对象可以访问无参数的fun函数，也可以访问带参数的fun函数*/
```

- 多层继承

如果派生类的部分或全部直接基类是从另一个共同的基类派生而来的，在这些直接基类中，从上一级基类继承来的成员就拥有相同的名称，因此派生类中也会产生同名现象（派生类拥有多个副本），对这种类型的同名成员也要使用作用域分辨符来唯一标识，且必须用直接基类来限定。

```
class base0
{
```

```
public:
        int var0;
        …
};
class base1: public base0
{
public:
        int var1;
        …
};
class base2: public base0
{
public:
        int var2;
        …
};

int main()
{
        derived d;
        d.base1::var0=2;
        d.base2::var0=3;
        …
}
```

虚基类

2016年10月6日　　20:09

将多继承情况中的共同基类设置为虚基类，这时从不同路径继承过来的同名数据成员在内存中只有一个副本，同一个函数名也只有一个映射。

```
class 派生类名: virtual  继承方式  基类名
{
    …
};
```

虚基类的初始化：如果虚基类声明有带形参的构造函数，那么在整个继承关系中直接或间接继承虚基类的所有派生类都必须在构造函数的成员初始化列表中对虚基类进行初始化

虚基类构造函数的调用：在建立一个对象时，如果这个对象中含有从虚基类继承来的成员，则虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。

构造一个类对象的一般顺序：

1. 若该类有直接或间接的虚基类，则先执行虚基类的构造函数。
2. 若该类还有其他基类，则按照它们在继承声明列表中出现的顺序，分别执行它们的构造函数，在构造的过程中不再执行它们的虚基类的构造函数。
3. 按照在类定义中出现的顺序，对派生类中新增的成员对象进行初始化。
4. 执行构造函数的函数体。

多态的概述

2016年10月7日　　17:58

多态：同样的消息被不同类型的对象接收时导致不同的行为。

多态的类型：重载多态（运算符重载）、强制多态（类型转换）、包含多态（虚函数）、参数多态（类模板）。

运算符重载

2016年10月7日　　18:33

运算符重载：对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时产生不同的行为。

```
返回类型 operator 运算符 (返回类型 &对象1)
{
    函数体
}
```

重载规则：

(1) 只能重载c++中已经有的运算符。

(2) 重载之后运算符的优先级和结合性都不会改变。

(3) 运算符重载不能改变原运算符的操作对象个数，同时至少要有一个自定义操作对象。

(4) 以下集中运算符不能重载：

类属关系运算符“.”

成员指针运算符“.*”

作用域分辨符“::”

三目运算符“?:”

重载为成员函数

当重载为成员函数时，函数的参数个数比原来的操作数个数少1，左操作数是对象本身的数据，右操作数是通过形参表传递的。

```
complex operator+ (const complex &c2)const;
```

```
clock clock::operator++ (int)//后置单目运算符(B++)重载
{
    clock old = *this;
    ++(*this);
    return old;
}

clock& clock::operator++ ()//前置单目运算符(++B)重载
{
    second++;
    if (second >= 60)
    {
        second -= 60;
        minute++;
        if (minute >= 60)
        {
            minute -= 60;
            hour = (hour + 1) % 24;
        }
    }
    return *this;

}
```

- 重载为非成员函数

运算需要的操作数都需要通过函数的形参表来传递，在形参表中形参从左到右的顺序就是运算符操作数的顺序。

```
complex operator+ (const complex &c1, const complex &c2)
{
    return complex(c1.real + c2.real, c1.imag + c2.imag);
}

ostream & operator<< (ostream &out, const complex &c)
{
    out << "(" << c.real << "," << c.imag << ")";
    return out;
}
```

```
int main()
{
    complex c1(5, 4), c2(1, 9),c3;
    cout << "c1=" << c1 << endl;
    cout << "c2="<<c2 << endl;//执行“cout<<c2”时调用了“operator<<(cout,c1)”函数
    c3 = c1 - c2;
    cout << "c3=c1-c2="<<c3<<endl;
    c3 = c1 + c2;
    cout << "c3=c1+c2="<<c3 <<endl;
```

虚函数

2016年10月7日　　20:28

虚函数的作用：如果需要通过基类的指针指向派生类的对象，并访问某个与基类同名的成员，那么首先在基类中将这个同名函数说明为虚函数，这样通过基类类型的指针，可以使属于不同派生类的不同对象产生不同的行为，从而实现运行过程的多态。

注意：

1. 虚函数必须使非静态的成员函数
2. 虚函数一般不声明为内联函数
3. 派生类的虚函数会覆盖基类的虚函数，如果要调用基类的虚函数可以使用“基类名::函数名(...)”

声明：

- **virtual 函数类型 函数名(形参表);**
- **void fun(base *ptr)**
  {
  ptr->虚函数名();
  }

虚函数的声明只能出现在类定义的函数原型声明中，不能在成员函数实现的时候声明。

```
class base1
{
public:
    virtual void display() const { cout << "base1::display()" << endl; }

};

class base2:public base1
{
public:
    virtual void display() const { cout << "base2::display()" << endl; }
};

class derived :public base2
{
public:
    virtual void display() const { cout << "derived::display()" << endl; }
};

void fun(base1 *ptr)
{
    ptr->display();
}
```

```
int main()
{
    base1 b1;
    base2 b2;
    derived d;
    fun(&b1);
    fun(&b2);
    fun(&d);
    return 0;
}
```

? 判断是否为虚函数

1. 该函数是否与基类的虚函数有相同的名字
2. 该函数是否与基类的虚函数有相同的参数个数、类型
3. 该函数是否与基类的虚函数有相同的返回值，或者是满足赋值兼容规则的指针、引用类型的返回值

虚析构函数

- virtual ~类名() {...}

```
class base1
{
public:
    virtual ~base1() { cout << "base1 destructor" << endl; }
    virtual void display() const { cout << "base1::display()" << endl; }
};
class base2:public base1
{
public:
    virtual ~base2() { cout << "base2 destructor" << endl; }
    virtual void display() const { cout << "base2::display()" << endl; }
};
class derived :public base2
{
public:
    virtual ~derived() { cout << "derived destructor" << endl; }
    virtual void display() const { cout << "derived::display()" << endl; }
};

void fun(base1 *ptr) { delete ptr;}
int main()
{
    base1 *b1=new derived();
    fun(b1);
    return 0;
}
```

如果基类不具有虚析构函数，删除派生类的对象只能通过派生类的指针，而通过基类指针删除时会出现不确定性问题

纯虚函数与抽象类

2016年10月8日　　8:37

抽象类：抽象类是一种带有纯虚函数的类，处于类层次的上层，无法实例化。作用主要是为一个类族建立一个公共的接口，而接口的完整实现（纯虚函数的函数体）要由派生类自己定义。

纯虚函数：在基类中声明的虚函数，在基类中没有定义具体的操作内容，要求各派生类根据实际需要给出各自的定义。
除了析构函数，如果将析构函数声明为纯虚函数，则必须给出它的实现，因为派生类的析构函数体执行完之后需要调用基类的析构函数。

- 声明：**virtual 函数类型 函数名(参数表)=0;**

运行时类型识别

2016年10月8日　　17:59

- **dynamic_cast**
  将基类的指针显示转换为派生类的指针，基类的引用显示转换为派生类的引用。在转换前会检查：
  a. 指针或引用的原始类型为多态类型的指针。
  b. 目的类型为派生类指针、void指针。
  c. 指针或引用所指向对象的实际类型是否与转换的目的类型兼容。
  派生类名 *指针名 = dynamic_cast<派生类名 *> (基类指针名);

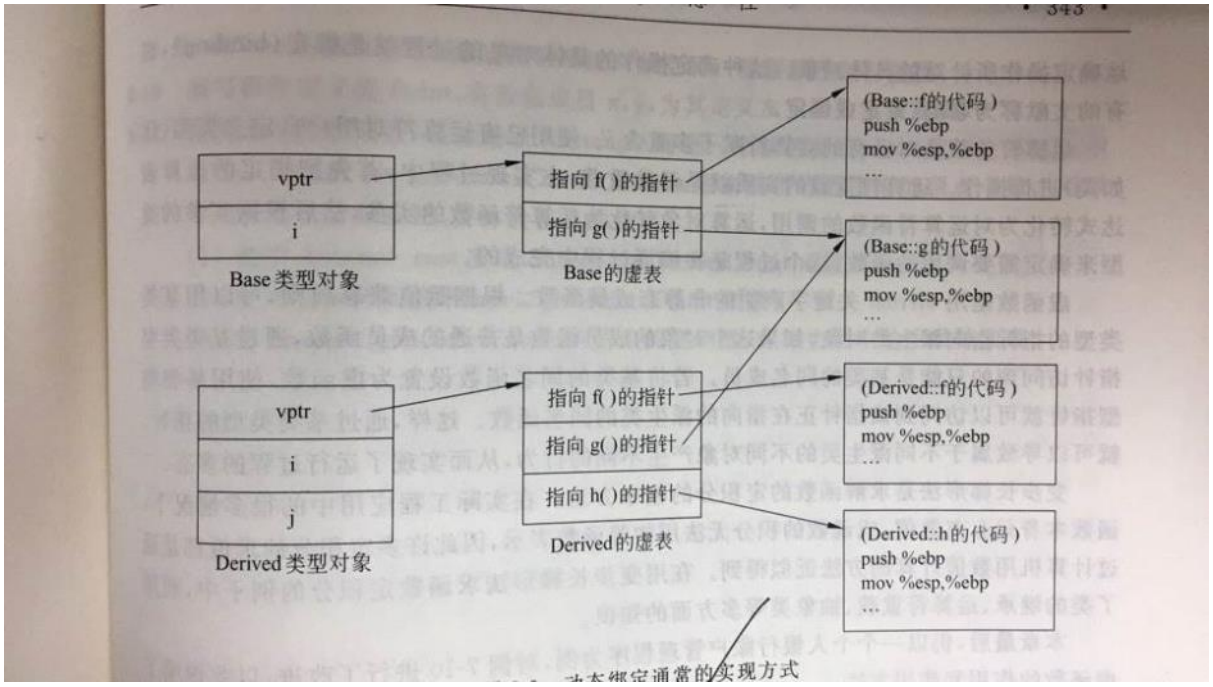
- **typeid**
  获得一个类型的相关信息。通过typeid得到的时一个type_info类型的常引用。type_info是C++标准库中的一个类，专用与在运行时表示类型信息，定义在<typeinfo>头文件中。
  const type_info &info = typeid (表达式);
  const type_info &info = typeid (类型说明符);

```
const type_info &info1 = typeid(b);
const type_info &info2 = typeid(*b);
cout << "typeid(b):" << info1.name() << endl;
cout << "typeid(*b):" << info2.name() << endl;
```

虚函数动态绑定的实现原理

2016年10月8日　　20:56

为每个类建立虚表：虚表的内容由编译器安排，存放着指向各个成员函数的指针。每个对象中保存一个指向这个虚表首地址的指针——虚表指针。只有多态类型有虚表，因此只有多态类型支持运行时类型识别。



函数模板

2016年10月9日　　9:01

参数化多态性：将程序所处理的对象的类型参数化，使一段程序可以用于处理多种不同类型的对象。

函数模板：一种算法用于处理多种数据类型。

```
template<typename 类型名>
类型名  函数名(参数表)
{
    函数体的定义
}
```

注意：

- 被多个源文件引用的函数模板，应当连同函数体一同放在头文件中，而不能像普通函数那样只将声明放在头文件中

```
#include <iostream>
using namespace std;
template<typename T>
T func(T x)
{
    return x < 0 ? -x : x;
}

int main()
{
    int x = 8;
    double y = 9;
    cout << func(x) << endl;
    cout << func(y) << endl;
    return 0;
}
```

类模板

2016年10月9日　　9:53

用户可以为类自定义一种模式，使得类中的某些数据成员、函数成员的参数、返回值或局部变量能取任意类型。

类：对一组对象的公共性质的抽象
类模板（参数化类）：对不同类的公共性质的抽象

声明

```
template<typename 参数名1, typename 参数名2,...>
class 类模板名
{
    类成员声明;
};
```

模板参数的默认实参：

```
template<typename T=int, int size=100>
class 类模板名
{
    类成员声明;
};
```

注意：函数模板、类模板的成员函数、类模板的静态数据成员的定义都必须放在头文件中

定义

在类模板外定义成员函数的形式：

```
template<模板参数列表>
返回值类型 类模板名<模板参数列表>::函数名(参数表)
{
    函数体
}
```

用模板类来建立对象：

```
模板名<模板实参> 对象1,...,对象n;
```

类模板的静态成员

2016年10月9日　　11:00

类模板的静态成员需要类内声明，类外定义。

template<模板参数列表>
数据类型 类模板名<模板参数列表>::静态数据成员=初始化值;

模板的实例化机制

2016年10月9日　　15:57

显式实例化：

template class Store<double>; //实例化Store类

template void outputArray<int> (const int *array, int count); //实例化函数

模板的全特化：为一个函数模板或类模板在某些特定参数下提供特殊的定义。

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

template<typename T1,typename T2>
class Test
{
public:
    Test(T1 s, T2 t) :a(s), b(t) { cout << "模板类Test" << endl; }
private:
    T1 a;
    T2 b;
};

template<>
class Test < int, char >
{
private:
    int a;
    char b;
public:
    Test(int s, char t) :a(s), b(t) { cout << "全特化" << endl; }
};
```

模板的偏特化：在一部分模板参数固定，另一部分模板参数可变的情况下规定类模板的特殊实现。

```
template<typename T2>
class Test < char, T2 >
{
private:
    char a;
    T2 b;
public:
    Test(char s, T2 t) :a(s), b(t) { cout << "偏特化" << endl; }
};
```

```
int main(void)
{
    Test<double, double> t1(0.1, 0.2);
    Test<int, char> t2(2,'a');
    Test<char, bool> t3('a',true);
    return 0;
}
```

函数模板的全特化

```
//模板函数
template<typename T1, typename T2>
void fun(T1 a , T2 b)
{
    cout<<"模板函数"<<endl;
}

//全特化
template<>
void fun<int ,char >(int a, char b)
{
    cout<<"全特化"<<endl;
}
```

函数模板的重载

```
template<typename T>
T myMax(T a, T b)
{
    return (a > b) ? a : b;
}

template<typename T>
T *myMax(T *a, T *b)
{
    return (*a>*b) ? a:b ;
}
```

在用指针调用myMax函数时，虽然两个模板都有可以与之匹配，但由于后一个更特殊，因此实际被调用的是后一个模板的实例。

模板元编程

2016年10月10日　　14:40

主要思想：

利用模板特化机制实现编译期条件选择结构，利用递归模板实现编译期循环结构，模板元程序则由编译器在编译期解释执行。

模板元编程的优势：

1. 以编译耗时为代价换来卓越的运行期性能（一般用于为性能要求严格的数值计算换取更高的性能）。通常来说，一个有意义的程序的运行次数（或服役时间）总是远远超过编译次数（或编译时间）。

2. 提供编译期类型计算

```
// 主模板
template<int N>
struct Fib
{
    enum { Result = Fib<N-1>::Result + Fib<N-2>::Result };
};
// 完全特化版
template <>
struct Fib<1>
{
    enum { Result = 1 };
};
// 完全特化版
template <>
struct Fib<0>
{
    enum { Result = 0 };
};
int main()
{
    int i = Fib<10>::Result;
    // std::cout << i << std::endl;
}
```

```
// 主模板
template<unsigned N>
inline double power(double v)
{
    return v*power<N-1>(v);
}

// 完全特化版
template<>
inline double power<1>(double v)
{
    return v;
}
```

输出流

2016年10月10日　　16:26

ostream标准设备的输出

- cout：标准输出流
- cerr：标准错误输出流，没有缓冲，发送给它的内容被立即读出
- clog：错误输出流，有缓冲

ofstream磁盘文件输出

若仅使用ostream类，则不需要构造输出流对象。若要使用文件流将信息输出到文件，则需要构造输出流对象。

- **输出流open函数**

```
ofstream myFile;
myFile.open("filename", ios_base::out|ios_base::binary);
/*以二进制模式输出数值到文件*/
...
myFile.close();
```

或

```
ofstream myFile("filename");
```

表 11-2　文件输出流文件打开模式

标　志	功　　能
ios_base::app	打开一个输出文件用于在文件尾添加数据
ios_base::ate	打开一个现存文件(用于输入或输出)并查找到结尾
ios_base::in	打开一个输入文件,对于一个 ofstream 文件,使用 ios_base::in 作为一个 open-mode 可避免删除一个现存文件中现有的内容
ios_base::out	打开一个文件,用于输出。对于所有 ofstream 对象,此模式是隐含指定的
ios_base::trunc	打开一个文件,如果它已经存在则删除其中原有的内容。如果指定了 ios_base::out,但没有指定 ios_base::ate,ios_base::app 和 ios_base::in,则隐含为此模式
ios_base::binary	以二进制模式打开一个文件(默认是文本模式)

- **输出流close函数：关闭与一个文件输出流关联的磁盘文件**
- **put函数：把一个字符写到输出流中**

```
cout.put('S');
```

- **write函数：把一个内存中的一块内容写入到一个文件输出流中**

```
ofstream myFile;
myFile.write ( reinterpret_cast<char *>(& 对象名), sizeof(对象名) );
```

- **skeep函数：**设置一个内部指针指出下一次写数据的位置
  tellp函数：返回该文件位置指针值

错误处理函数

cout.bad();

表 11-3　错误处理成员函数及其功能

函数	功能及返回值
bad	如果出现一个不可恢复的错误，则返回一个非 0 值
fail	如果出现一个不可恢复的错误或一个预期的条件，例如一个转换错误或文件未找到，则返回一个非 0 值。在用零参量调用 clear 之后，错误标记被清除
good	如果所有错误标记和文件结束标记都是清除的，则返回一个非 0 值
eof	遇到文件结尾条件，则返回一个非 0 值
clear	设置内部错误状态，如果用默认参量调用，则清除所有错误位
rdstate	返回当前错误状态

ostringstream字符串输出流

典型用法：将数值转换为字符串

创建字符串输出流：ostringstream os;

调用特有的函数str：os.str();

```
template<typename T>
inline string toString(const T &v)
{
    ostringstream os;
    os << v;                    //将变量v的值输出到字符串流
    return os.str();
}
```

```
int main()
{
    /*int v1 = fromString<int>("5");
    cout << v1 << endl;
    double v2 = fromString<double>("5.5");
    cout << v2 << endl;*/

    string str1=toString(5);
    cout << str1 << endl;
    return 0;
}
```

插入运算符&输出流操纵符

2016年10月10日　　17:09

插入运算符“<<”，用于传送字节到一个输出流对象

输出宽度

cout<<width(n);　　//定义在头文件“iostream”中

cout<<setw(n)<<...　　//定义在头文件“iomanip”中

width，setw函数都不截断数值，仅影响紧随其后的域

对齐方式

cout<<setiosflags(ios_base:: *)<<...**　　//定义在头文件“iomanip”中

影响是持久的，直到用resetiosflags重新恢复默认值为止

ios_base::skipws	在输入中跳过空白
ios_base::left	左对齐值
ios_base::right	右对齐值
ios_base::internal	在规定的宽度内，指定前缀符号之后，数值之前插入指定的填充字符
ios_base::dec	
ios_base::oct	
ios_base::hex	以16进制形式格式化数值
ios_base::showbase	插入前缀符号以表明整数的数制
ios_base::showpoint	对浮点数制显示小数点和尾部的0
ios_base::uppercase	对16进制数值显示大写字母A到F
ios_base::showpos	对非负数显示正号
ios_base::scientific	以科学格式显示浮点数值
ios_base::fixed	以定点格式显示浮点数值（没有指数部分）
ios_base::unitbuf	在每次插入之后转储并清除缓冲区内容

精度

cout<<setprecision(n)<<...　　//定义在头文件“iomanip”中

进制

填充符

```
cout.fill('*');
```
//用星号填充，影响是持久的

输入流

2016年10月10日　　20:36

- istream

  cin>>...

- ifstream

  - 输入流的open函数

ios_base::in	以文本形式打开文件（默认）
ios_base::binary	以二进制模式打开文件

```
ifstream myFile;
myFile.open("filename");
```

或

```
ifstream myFile("filename");
```

  - 输入流的close函数

  - get函数：在读入数据时包括空白符

```
char ch;
while((ch=cin.get())!=EOF)  //表示文件结尾，EOF还可以表示标准输入的结尾
    cout.put(ch);
```

  - getline函数：从输入流中读取多个字符，并允许指定输入终止字符（默认是换行符），读取完成后从读取的内容中删除终止字符。结果存储在字符数组中，数组的大小不能自动扩展。声明在“<string>”头文件中。

```
string 字符数组名;
getlin(cin, 字符数组名, ' 终止符' );
```

  - read函数：从一个文件读字节到一个指定的存储器区域，由长度参数确定要读取的字节数。如果给出长度参数，当遇到文件结束或在文本模式文件中遇到文件结束标记字符时读取操作终止。

```
ifstream myFile;
myFile.read ( reinterpret_cast<char *>(& 对象名), sizeof(对象名) );
```

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

struct SalaryInfo
{
    unsigned id;
    double salary;
};

void main()
{
    SalaryInfo employee1 = { 282120128, 8000 };
    ofstream ostream("payroll", ios_base::out|ios_base::binary);
    /*将内容以二进制格式输出到“payroll”文件中*/
    ostream.write(reinterpret_cast<char *>(&employee1),sizeof(employee1));
    ostream.close();

    ifstream istream("payroll", ios_base::in | ios_base::binary);
    /*从文件“payroll”中读取二进制数据*/
    if (istream)
    {
        SalaryInfo employee2;
        istream.read(reinterpret_cast<char *>(&employee2), sizeof(employee2));
        cout << "#" << employee2.id << " " << employee2.salary << endl;
    }
    else
        cout << "ERROR: Cannot open file 'payroll'" << endl;
    istream.close();
}
```

- seekg函数：在文件输入流中保留一个指向文件中下一个将读取数据的位置的内部指针。
- tellg函数：返回当前文件读指针的位置

- istringstream
  典型用法：将一个字符串转换为数值
  string str=…;
  istringstream is(str);

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

template <typename T>
inline T fromString(const string &str)
{
    istringstream is(str);
    T v;
    is >> v;
    return v;
}

int main()
{
    int v1 = fromString<int>("5");
    cout << v1 << endl;
    double v2 = fromString<double>("5.5");
    cout << v2 << endl;
    return 0;
}
```

提取运算符&输入流操纵符

2016年10月10日　　20:48

提取运算符“>>”以空白符为分割，格式化文本输入。

几个头文件

2016年10月10日　　21:15

<iostream>
<sstream>
<fstream>
<string>

答疑解惑

2016年10月12日　　9:27

1. **c语言中，小写bool与大写BOOL有区别吗？我替换了下似乎对程序不影响呀**

   最佳答案 ：
   当然有，bool是C++标准规定的布尔类型，取值为true或false。而BOOL是Microsoft定义的一种类型，语法为：typedef int BOOL。也就是说BOOL类型实际上是int类型不是bool类型，取值为：TRUE或FALSE、定义语法为：
   #define TRUE 1
   #define FALSE 0
   但是如果没有严格要求的话，布尔类型和Int类型是通用的。非0值的int类型可以转为布尔类型的true，称为隐式转换。0的Int值则可以转为布尔类型的false。

2. **VOID 和void区别**

   最佳答案：VIOD 是 windows 的头文件里定义的（ windows.h ）。标准的就是 void，这个是通用的

3. **FALSE/TRUE与false/true的区别**

   false/true是标准C++语言里新增的关键字，而FALSE/TRUE是通过#define，这要用途是解决程序在C与C++中环境的差异,以下是FALSE/TRUE在windef.h的定义：
   #ifndef FALSE
   #define FALSE 0
   #endif
   #ifndef TRUE
   #define TRUE 1
   #endif

QT

2016年10月15日　　11:47

QT中的头文件

```
#include <QMainWindow>
#include <QTextEdit>    //QT库中的文本编辑控件
#include <QMenu>        //QT库中的菜单控件
#include <QMenuBar>     //QT库中的菜单栏控件
```

QT中的常用函数

- 将文本编辑窗口放置在对话框中间

```
text1=new QTextEdit;//为指针text1动态分配空间
this->setCentralWidget(text1);//将这个textedit放到对话框中间
```

- 设置对话框的大小

```
MainWindow w;
w.resize(500,300);//将对话框长设置为500，高设置为300
```

- 设置文本编辑窗口内字体的大小

```
QFont f;
f.setPixelSize(30);
text1->setFont(f);//设置对话框中字体大小为30
```

- 设置文本编辑窗口内字体颜色

```
QColor color;
color.setRgb(255,0,0);//设置红、绿、蓝数值
text1->setTextColor(color);//设置字体颜色
```

- 弹出信息窗口

```
QMessageBox::information(this,"提示",command);
//第二个参数：窗口标题
//第三个参数：窗口内容
```

添加菜单项

- 添加菜单项

```
QMenu *file;
QMenu *edit;
QMenu *help;

//在菜单栏加入菜单项
file=this->menuBar()->addMenu("文件");
edit=this->menuBar()->addMenu("编辑");
help=this->menuBar()->addMenu("帮助");
```

- 添加下拉菜单

```
QAction *file_open;
QAction *close;
QAction *save;
//在菜单项中创建下拉菜单
file_open = new QAction("open",this);    //建立open Action
file_open->setShortcut(tr("Ctrl+o"));    //设置open的快捷方式
file->addAction(file_open);              //将file_open这个Action加入到file菜单

file->addSeparator();                    //加入一个分隔符
file_exit = new QAction("exit",this);
file_exit->setShortcut(tr("Ctrl+e"));
file->addAction(file_exit);
```

action链接的函数

- open函数

```
private slots://QT中函数的固定写法
    void on_open();
```

```
connect(file_open,SIGNAL(triggered()),this,SLOT(on_open()));
//参数1：触发这个事件的控件
//参数2：对于Action来讲是固定写法
//参数3：固定写法
//参数4：指定了点击这个Action之后会调用的函数
```

```
void MainWindow::on_open()
{
    QString content;//QT定义的一个字符串

    //函数返回用户选择的路径+文件名
    //QMessageBox::information(this,"提示",filename);
    QString filename = QFileDialog::getOpenFileName();
    //如果用户没有选择任何文件
    if(filename.isEmpty())
        return;

    //filename.toStdString().data()将QString转换为const char* 类型
    FILE *p=fopen(filename.toStdString().data(),"r");
    //如果路径为空
    if(p==NULL)
    {
        QMessageBox::information(this,"error","open failed!");
    }
    else
    {
        while(!feof(p))//feof()函数功能：C语言中检测流上的文件结束符
        {
            char buf[1024]={0};
            fgets(buf,sizeof(buf),p);//读取指针p指向的文件内容存储到buf
            content +=buf;//将buf的内容追加到content的后面
```

```
        }
        fclose(p);
        text1->setText(content);//将字符串content放入text1里面
    }
}
```

- copy函数

```
void MainWindow::on_copy()
{
    text1->copy();
}
```

- paste函数

```
void MainWindow::on_paste()
{
    text1->paste();
}
```

- save函数

```
void MainWindow::on_save()
{

    filename = QFileDialog::getSaveFileName();//打开保存对话框
    if(filename.isEmpty())
        return;
    FILE *p=fopen(filename.toStdString().data(),"w");
    if(p==NULL)
        QMessageBox::information(this,"error","open failed!");
    else {...}
}
```

- compile函数


ide
file edit help build
#include <stdio.h>
int main()
{
printf("hello");
return 0;
}
提示
gcc -o a D:/doc/a.exe.txt D:/doc/a.exe.txt
OK

Tetris

2016年10月15日　　14:29

设置主窗口

1. 功能说明：最小化按钮、关闭按钮

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // 将实例句柄存储在全局变量中

   hWnd = CreateWindow(szWindowClass, szTitle, WS_MINIMIZEBOX | WS_SYSMENU,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
```

2. 定义宏：边框大小、每个小格大小、游戏区x和y轴方块个数、信息区x和y轴方块个数

```
#define BOUND_SIZE 10          //边框大小
#define TETRIS_SIZE 30         //小方块大小
#define GAME_X 10              //游戏区x轴小方块个数
#define GAME_Y 20              //游戏区y轴小方块个数
#define INFO_X 6               //信息区x轴方向的宽度
#define INFO_Y GAME_Y          //信息区y轴方向的宽度
```

3. 调整主窗口位置、大小

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    int nWin_X, nWin_Y, nClient_X, nClient_Y;
    RECT rect;
    BOOL bTmpTetris[4][4];

    switch (message)
    {
    case WM_CREATE:
        //获取窗口大小
        GetWindowRect(hWnd, &rect);
        nWin_X = rect.right - rect.left;
        nWin_Y = rect.bottom - rect.top;
        //获取客户区窗口大小
        GetClientRect(hWnd, &rect);
        nClient_X = rect.right - rect.left;
        nClient_Y = rect.bottom - rect.top;

        MoveWindow(hWnd, 400,50,
            3 * BOUND_SIZE + TETRIS_SIZE*(GAME_X + INFO_X) + (nWin_X - nClient_X),
            2 * BOUND_SIZE + TETRIS_SIZE*GAME_Y + (nWin_Y - nClient_Y),TRUE);
```

4. 编写绘制游戏区、信息区背景的函数

```
//
//函数：DrawGameBackground(HDC)
//目的：绘制游戏区的背景图案
//
void DrawGameBackground(HDC hdc) { ... }

//
//函数：DrawInfoBackground(HDC)
//目的：绘制信息区的背景图案
//
void DrawInfoBackground(HDC hdc) { ... }
```

5. 定义全局变量：模板方块

```
BOOL modelTetris[][4][4] =                          //定义俄罗斯基本的几种方块，1表示选中方格
{
    { { 1, 1, 1, 1 },{ 0, 0, 0, 0 },{ 0, 0, 0, 0 },{ 0, 0, 0, 0 } },//“一”字形
    { { 0, 0, 0, 0 },{ 0, 1, 1, 0 },{ 0, 1, 1, 0 },{ 0, 0, 0, 0 } },//“田”字形
    { { 1, 0, 0, 0 },{ 1, 1, 0, 0 },{ 1, 0, 0, 0 },{ 0, 0, 0, 0 } },//“土”字形
    { { 1, 1, 1, 0 },{ 1, 0, 0, 0 },{ 0, 0, 0, 0 },{ 0, 0, 0, 0 } },//“L”字形
    { { 1, 1, 0, 0 },{ 0, 1, 1, 0 },{ 0, 0, 0, 0 },{ 0, 0, 0, 0 } },//“Z”字形
    { { 0, 1, 1, 0 },{ 1, 1, 0, 0 },{ 0, 0, 0, 0 },{ 0, 0, 0, 0 } } //反“Z”字形
};
```