C++面向对象程序设计考试范围

整理by 2015级计算机科学与技术 冉诗菡

- 程序结构

1.C++内置类型(bool, char, int, float, double, 指针)和引用类型（&）

- bool:

```
bool operator != (const BigInt& temp1,const BigInt& temp2)
{
   if(temp1 == temp2) return 0;
   else return 1;
}
```

- 指针

eg1:已知p是指向一个类A数据成员m的指针，A1是类A的一个对象。
如果要给m赋值为5，可以写为：A1.*p=5
注意“*”比“.”的级别要高的 所以*A1.p=5 相当于 (*A1).p=5 是不对的
eg2:

```
struct student  //声明结构体student
{
    float score;
}stu={22};  //初始化score
int main()
{
   struct student *p=&stu;  //声明指向结构体student的指针
   cout<<p->score;   //通过指针访问结构体中的score成员
   return 0;
 }
```

- 引用:

（1）&在此不是求地址运算，而是起标识作用。
（2）引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，且不能再把该引用名作为其他变量名的别名。

int a=2,int &ra=a;
a为目标原名称，ra为目标引用名。给ra赋值：ra=1; 等价于 a=1;（3）对引用求地址，就是对目标变量求地址。&ra与&a相等。即我们常说引用名是目标变量名的一个别名。别名一词好像是说引用不占据任何内存空间。但是编译器在一般将其实现为const指针，即指向位置不可变的指针。即引用实际上与一般指针同样占用内存。
（4）引用必须在定义时马上被初始化，因为它必须是某个东西的同义词。你不能先定义一个引用后才初始化它。
（5）当大型对象被传递给函数时，使用引用参数可使参数传递效率得到提高，因为引用并不产生对象的副本，也就是参数传递时，对象无须复制。
（6）常引用声明方式：const 类型标识符&引用名=目标变量名；用这种方式声明的引用，不能通过引用对目标变量的值进行修改,从而使引用的目标成为const，达到了引用的安全性。
eg1:
int a ;
const int &ra=a;
ra=1; //错误
a=1; //正确
eg2:
string foo( );
void bar(string & s);
那么下面的表达式将是非法的：
bar(foo( ));
bar("hello world");
原因在于foo( )和"hello world"串都会产生一个临时对象，而在C++中，这些临时对象都是const类型的。因此上面的表达式就是试图将一个const类型的对象转换为非const类型，这是非法的。
引用型参数应该在能被定义为const的情况下，尽量定义为const。

2.声明：常量、变量、函数、结构、类

eg：

```
class BigInt{

        friend std::istream& operator>>(std::istream&, BigInt&);
        friend bool operator==(const BigInt&, const BigInt&);
        friend bool operator>(const BigInt&, const BigInt&);
        friend BigInt operator +(BigInt, BigInt);
        friend BigInt operator -(BigInt, BigInt);
        friend BigInt operator *(const BigInt&, const BigInt&);
        friend BigInt operator /(const BigInt&, const BigInt&);
        friend BigInt operator %(const BigInt&, const BigInt&);
        friend BigInt absv(const BigInt&);


public:
        typedef std::vector<int>::size_type vec_sz;

        BigInt();
        BigInt(int);
        BigInt(double);
        BigInt(vec_sz, unsigned int);
        BigInt(std::istream& in) { read(in); }

        vec_sz size() const { return data.size() - 1; }

        BigInt& operator=(int);
        BigInt& operator=(double);
        BigInt& operator=(const BigInt&);
        BigInt& operator-();

        std::istream& read(std::istream&);
        std::ostream& output(std::ostream&);

private:
        std::vector<int> data;

        int& operator[](vec_sz i) { return data[i]; }
        const int& operator [](vec_sz i) const { return data[i]; }

};
```

3.输入输出：std::cin, std::cout, 文本文件的输入/输出<fstream>

eg1：

语句序列ifstream infile;

infile.open(“data.txt”);

的功能可以用一个语句实现，这个语句是：

ifstream infile(“data.txt”);

其实infile你可以理解成一个对象

就是一个变量

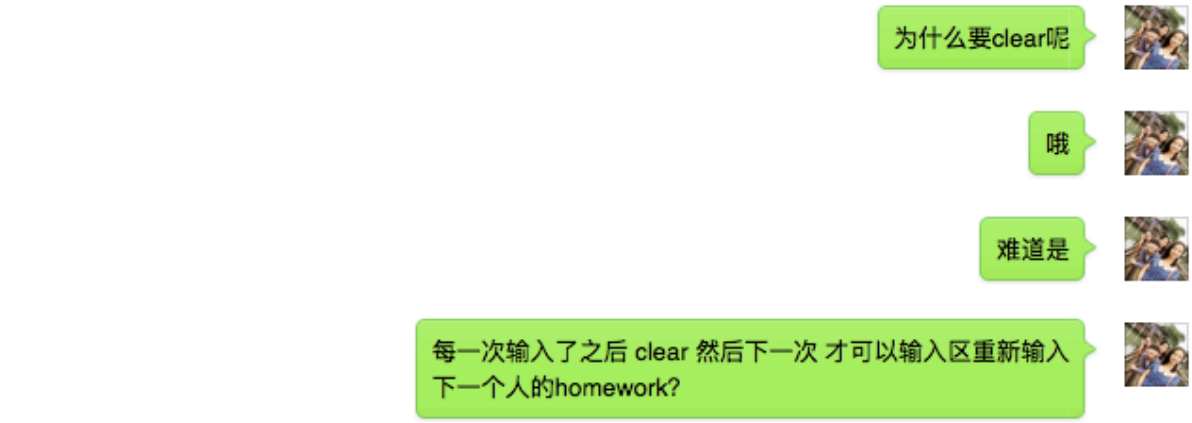
它有一个成员函数叫open(文件名)

如果直接定义infile后面加括号就相当于是构造函数的使用方法

eg2:

```
int main()
{
        //如果想以输入方式打开，就用ifstream来定义;
        //如果想以输出方式打开，就用ofstream来定义
        ifstream infile("grades.txt");
        ofstream outfile("final.txt");

        vector<Student_info> students;
        Student_info record;
        string::size_type maxlen = 0;

        while (record.read(infile)) //读入grades文件
        {
                maxlen = max(maxlen, record.name().size());
                students.push_back(record);
        }

istream& Student_info::read(istream& in)
{
        in >> n >> midterm >> final ;
        read_hw(in,homework);
        score = grade();
        return in;
}

istream& Student_info::read_hw(istream& in, vector<double>& hw)
{
        if (in)
        {
                hw.clear();

                double x;
                while (in >> x)
                        hw.push_back(x);

                in.clear();
        }
        return in;
}
```

至于那个 in.clear(); return in; 是因为



4.重载：函数和操作符重载（包括<<和>>操作符）

eg1：

```
BigInt& BigInt::operator=(const BigInt& temp)
{
    if (&temp != this) {    //先加一步判断是否指向自己本身 再清空
        data.clear();
        data = temp.data;
    }

    return *this;
}
```

eg2:

```
BigInt& BigInt::operator-()     //取负操作符"-".
{
    if (data[0] == 0) data[0] = 1;
    else data[0] = 0;

    return *this;
}
```

eg3:

```
std::istream& BigInt::read(std::istream& is)
{
        char record;

        while (is.get(record) && isspace(record))  //如果读入的是空格的话 就什么也不做 跳过它
                ;

        if (record == '-'){    //如果读入的是负数
                data.push_back(1);
                is.get(record);     //继续读入
        }
        else data.push_back(0);

        if (is){
                do data.push_back(record - '0');
                while (is.get(record) && !isspace(record));    //当能成功读入并且读入的不是空格符的时候就把它加到data里面

                if (is)   //假如读入成功，必然是读入了空格符，因此将空格符返回，下一次读入的时候，这个空格还是会被读入的
                        is.unget();   //将读入的东西返回到输入流里面
        }

        reverse(data.begin() + 1, data.end());  //因为要倒序储存

        return is;
}

std::ostream& BigInt::output(std::ostream& os)
{
        if (os){
                if (data.size() == 0) return os;

                if (absv(*this) == 0){
                        os << '0';
                }
                else {
                        if (data[0] == 1) os << '-';

                        for (vec_sz i = data.size() - 1; i != 0; i--)
                                std::cout << data[i];
                }
                os.clear();
        }

        return os;
}

std::istream& operator>>(std::istream& is, BigInt& temp)  //重载 输入操作符 其实是用read函数来实现
{
        if (is){
                temp.data.clear();
                temp.read(is);
                is.clear();
        }

        return is;
}

std::ostream& operator<<(std::ostream& os, BigInt& temp) //重载 输出操作符 其实使用output函数来实现
{
        if (os){
                temp.output(os);
                os.clear();
        }

        return os;
}
```

5.模板：函数模板，类模板。

eg：自定义模版类vec （详情看之前写的程序吧“模版类vec完成成绩等级函数”像没写过一样的一点都没有印象

```
template <class T> class Vec {
public:
        typedef T* iterator;
        typedef const T* const_iterator;
        typedef size_t size_type;
        typedef T value_type;

        Vec() { create();}
        explicit Vec(size_type n, const T& t=T()) { create(n,t);}  //why t=T()

        Vec(const Vec& v) { create(v.begin() ,v.end()) ;}
        Vec& operator=(const Vec&);
        ~Vec() { uncreate(); }
        T& operator[] ( size_type i) { return data[i] ; }
        const T& operator[] ( size_type i) const { return data[i] ; }

        void push_back(const T& t){
                if(avail == limit)
                        grow();
                unchecked_append(t);
        }
        size_type size() const { return avail-data ; }
        bool empty() const { return size() == 0 ; }

        iterator begin() { return data ; }
        const_iterator begin() const { return data ; }

        iterator end() { return avail ; }
        const_iterator end() const { return avail ; }

private:
        iterator data;
        iterator avail;
        iterator limit;

        //内存分配工具
        allocator<T> alloc;  //控制内存分配的对象

        //为底层的数组分配空间并对它进行初始化
        void create();
        void create(size_type,const T&);
        void create(const_iterator,const_iterator);

        //删除数组中的元素并释放其占用的内存
        void uncreate();

        //支持push_back的函数
        void grow();
        void unchecked_append(const T&);
};
```

6.动态内存管理：new, delete

new和delete运算符是用于动态分配和撤销内存的运算符。

一、new用法

1.开辟单变量地址空间

使用new运算符时必须已知数据类型，new运算符会向系统堆区申请足够的存储空间，如果申请成功，就返回该内存块的首地址，如果申请不成功，则返回零值。

new运算符返回的是一个指向所分配类型变量（对象）的指针。对所创建的变量或对象，都是通过该指针来间接操作的，而动态创建的对象本身没有标识符名。

一般使用格式：

格式1：指针变量名=new 类型标识符；

格式2：指针变量名=new 类型标识符（初始值）；

格式3：指针变量名=new 类型标识符 [内存单元个数]；

说明：格式1和格式2都是申请分配某一数据类型所占字节数的内存空间；但是格式2在内存分配成功后，同时将一初值存放到该内存单元中；而格式3可同时分配若干个内存单元，相当于形成一个动态数组。例如：

1)new int; //开辟一个存放整数的存储空间,返回一个指向该存储空间的地址。int *a = new int 即为将一个int类型的地址赋值给整型指针a

2)int *a = new int(5) 作用同上,但是同时将整数空间赋值为5

2.开辟数组空间

对于数组进行动态分配的格式为：

指针变量名=new 类型名[下标表达式];

delete [ ] 指向该数组的指针变量名;

两式中的方括号是非常重要的，两者必须配对使用，如果delete语句中少了方括号，因编译器认为该指针是指向数组第一个元素的指针，会产生回收不彻底的问题（只回收了第一个元素所占空间），加了方括号后就转化为指向数组的指针，回收整个数组。

delete []的方括号中不需要填数组元素数，系统自知。即使写了，编译器也忽略。

请注意“下标表达式”不必是常量表达式，即它的值不必在编译时确定，可以在运行时确定。（就是在运行到这里的时候 这个表达式的值已知就可以了

一维: int *a = new int[100]; //开辟一个大小为100的整型数组空间

二维: int **a = new int[5][6]

三维及其以上:依此类推.

一般用法: new 类型 （初值）

二、delete用法

1. 删除单变量地址空间

int *a = new int;

delete a; //释放单个int的空间

2. 删除数组空间

int *a = new int[5];

delete []a; //释放int数组空间

三、使用注意事项

1. new 和delete都是内建的操作符，语言本身所固定了，无法重新定制，想要定制new和delete的行为，徒劳无功的行为。

2. 动态分配失败，则返回一个空指针（NULL），表示发生了异常，堆资源不足，分配

失败。

3. 指针删除与堆空间释放。删除一个指针p（delete p;）实际意思是删除了p所指的目标（变量或对象等），释放了它所占的堆空间，而不是删除p本身（指针p本身并没有撤销，它自己仍然存在，该指针所占内存空间并未释放），释放堆空间后，p成了空指针。

4. 内存泄漏（memory leak）和重复释放。new与delete 是配对使用的， delete只能释放堆空间。如果new返回的指针值丢失，则所分配的堆空间无法回收，称内存泄漏，同一空间重复释放也是危险的，因为该空间可能已另分配，所以必须妥善保存new返回的指针，以保证不发生内存泄漏，也必须保证不会重复释放堆内存空间。

5. 动态分配的变量或对象的生命期。我们也称堆空间为自由空间（free store），但必须记住释放该对象所占堆空间，并只能释放一次，在函数内建立，而在函数外释放，往往会出错。

6. 要访问new所开辟的结构体空间,无法直接通过变量名进行,只能通过赋值的指针进行访问。

用new和delete可以动态开辟和撤销地址空间。在编程序时,若用完一个变量(一般是暂时存储的数据),下次需要再用,但却又想省去重新初始化的功夫，可以在每次开始使用时开辟一个空间，在用完后撤销它。

- # 面向对象

面向对象的三大特点：1）封装；2）继承；3）多态。

- 类：

1）构造函数、复制构造函数、带参数构造函数、默认构造函数；
2）构造函数中的初始化列表；

一.默认构造函数.
与用户自定义构造函数相比,默认构造函数有功能缺陷:
只能分配空间,完成不了初始化数据的任务(类的数据成员都是基本类型数据类型时的默认拷贝构造函数是个例外).

1.一般的默认构造函数

若用户没有定义任何构造函数(包括拷贝构造函数),则编译器自动添加默认构造函数(不带任何参数),但该函数不作任何实际工作.即它不能完成对类的数据成员的初始化工作,若用户在用其它方式初始化这些数据成员之前对它们进行访问,就会出错.因此,在任何情况下,用户都应该自定义构造函数,而不使用编译器提供的默认构造函数.

2.默认拷贝构造函数

若用户没有定义拷贝构造函数,则编译器自动添加默认拷贝构造函数(带一个该类类型的

参数),称为浅拷贝.它只能完成基本类型数据类型(如int型变量)的拷贝,若类中有动态数组等数据类型,浅拷贝就会出问题.即是说,浅拷贝有潜在危险(当类的数据成员都是基本类型数据类型时,它是安全的).因此,在任何情况下,用户自定义拷贝构造函数是可取的.

二.用户自定义的构造函数(可以重载).
用户自定义的构造函数需要完成两个功能:分配空间,初始化数据.

eg1:

```
Complex() : a(0),b(0) {}
Complex(double x,double y = 0) :a(x),b(y) {}  //一定要写 double y = 0 这样就把实数包含在了复数里面
Complex(std::istream& in) { in >> a >> b ;}  //输入构造函数
```

eg2:

```
BigInt();
BigInt(int);
BigInt(double);
BigInt(vec_sz, unsigned int);
BigInt(std::istream& in) { read(in); }
```

eg3:

```
Vec() { create(); }  //默认构造函数
explicit Vec(size_type n, const T& t = T()) { create(n, t); }
Vec(const_iterator b, const_iterator e) {create(b, e);}
Vec(const Vec& v) { create(v.begin(), v.end());} //拷贝构造函数
```

//1.构造函数不能有返回值
//2.缺省构造函数时，系统将自动调用该缺省构造函数初始化对象，缺省构造函数会将所有数据成员都 初始化为零或空
// 3.创建一个对象时，系统自动调用构造函数

//拷贝构造函数实际上也是构造函数，具有一般构造函数的所有特性，其名字也与所属类名相同。拷贝构造函数中只有一个参数，这个参数是对某个**同类**对象的引用。
//在三种情况下被调用：1.用类的一个对象去初始化该类的另一个对象时； 2.函数的形参是类的对象，调用函数进行形参和实参的结合时；这也是为什么拷贝构造函数的参数需要加引用，避免反复循环调用构造函数。

//拷贝构造函数使用对象的引用作为形参，防止临时产生一个对象

3）析构函数；

下面通过一个例子来说明一下析构函数的作用：

```
#include<iostream>
using namespace std;
class T
{
    public:
     ~T(){cout<<"析构函数被调用。";} //为了简洁，函数体可以直接写在定义的后面，此时函数为内联函数
};
int main()
{
     T *t=new T();//建立一个T类的指针对象t
     delete t;
     cin.get();
};
```

最后输出：

析构函数被调用。

cin.get() 表示从键盘读入一个字符，为了让我们能够看得清楚结果。***当然，析构函数也可以显示的调用，如 (*t).~T(); 也是合法的。***

//析构函数没有参数，也没有返回值。不能重载，也就是说，一个类中只可能定义一个析构函数
//如果一个类中没有定义析构函数，系统也会自动生成一个默认的析构函数，为空函数，什么都不做
//调用条件：1.在函数体内定义的对象，当函数执行结束时，该对象所在类的析构函数会被自动调用；2.用new运算符动态构建的对象，在使用delete运算符释放它时。3.函数的返回值是类的对象，函数执行完返回调用者。

4）const成员函数；

通过const成员函数只能读取同一类中的数据成员的值，而不能修改它。即常成员函数不能更新对象的数据成员。
当一个对象被声明为const对象，则不能通过该对象调用该类中的非const成员函数，因为它可能企图修改常量的数据成员。
但构造函数和析构函数对这个规则例外，它们从不定义为常量成员，但可被常量对象调用（被自动调用）。它们也能给常量的数据成员赋值，除非数据成员本身是常量。

const成员函数应该在函数原型说明和函数定义中都增加const限定。
eg：
int GetY() const; //声明常成员函数
int Point::GetY() const //定义常成员函数
{ return yVal; }

对于内置的数据类型，我们可以定义它们的常量，用户自定义的类也一样，可以定义它们的常量对象。例如，定义一个整型常量的方法为：
const int i=1 ；
同样，也可以定义常量对象，假定有一个类classA，定义该类的常量对象的方法为：

const classA a(2);
这里，a是类classA的一个const对象，"2"传给它的构造函数参数。const对象的数据成员在对象生存期内不能改变。
为了确保const对象的数据成员不会被改变，在C++中，**const对象只能调用const成员函数**。
如果一个成员函数实际上没有对数据成员作任何形式的修改，但是它没有被const关键字限定的，也不能被常量对象调用。

5）访问器函数；
解决用户无法访问类的private成员的问题

eg：
```
class Point
{
public:
int GetX() const; //声明访问器函数
int GetY() const;
private:
int xVal, yVal;
};

int Point::GetY() const //定义访问器函数
{
return yVal;
}
```

6）赋值操作符；

- 拷贝构造函数是用一个已存在的对象去构造一个不存在的对象（拷贝构造函数毕竟还是构造函数嘛），也就是初始化一个对象。
- 而赋值运算符重载函数是用一个存在的对象去给另一个已存在并初始化过（即已经过构造函数的初始化了）的对象进行赋值。
- 拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用。
- 比如：`String s1("hello"),s2=s1; //拷贝构造函数`
- `Sring s1 ("hello"),s2;  s1=s2; //赋值运算符重载 即赋值函数`

- **以下情况都会调用拷贝构造函数：**

1、一个对象以值传递的方式传入函数体 （形参和实参结合）
2、一个对象以值传递的方式从函数返回 （函数返回
3、一个对象需要通过另外一个对象进行初始化。

- 类String的拷贝构造函数与赋值函数
- // 拷贝构造函数
  String::String(const String &other)
  {
  // 允许操作other的私有成员m_data
  int length = strlen(other.m_data);
  m_data = new char[length+1];
  strcpy(m_data, other.m_data);
  }
- // 赋值函数
  String & String::operator =(const String &other)
  {
  // (1) 检查自赋值
  if(this == &other)
  return *this;
  // (2) 释放原有的内存资源
  delete [] m_data;
  // (3)分配新的内存资源，并复制内容
  int length = strlen(other.m_data);
  m_data = new char[length+1];
  strcpy(m_data, other.m_data);
  // (4)返回本对象的引用
  return *this;
  }

7）操作符重载；
见前面：）

8）属性和成员函数（方法），内联函数(inline)；

1．要求通过函数来实现一种不太复杂的功能，并且要求加快执行速度，选用 A
A．内联函数 B．重载函数 C．内部函数 D．函数模板

- 内联函数：从源代码层看，有函数的结构，而在编译后，却不具备函数的性质。内联

函数不是在调用时发生控制转移，而是在编译时将函数体嵌入在每一个调用处。编译时，类似宏替换，使用函数体替换调用处的函数名。一般在代码中用inline修饰，但是能否形成内联函数，需要看编译器对该函数定义的具体处理。内联扩展是用来消除函数调用时的时间开销。它通常用于频繁执行的函数。 一个小内存空间的函数非常受益。

- 为了减少时间开销，如果在类体中定义的成员函数中不包括循环等控制结构，可以省略inline，C++系统会自动将它们作为内置(inline)函数来处理。
- 如果成员函数不在类体内定义，而在类体外定义，系统并不把它默认为内置(inline)函数，调用这些成员函数的过程和调用一般函数的过程是相同的。如果想将这些成员函数指定为内置函数，应当用inline作显式声明。如果在类体外定义inline函数，则必须将类定义和成员函数的定义都放在同一个头文件中(或者写在同一个源文件中)，否则编译时无法进行置换(将函数代码的拷贝嵌入到函数调用点)。

9）类的保护标识符(public, protected, private)；

protected：

protected专门就是为继承（子类）设计的 用public继承 那么基类所有的访问标识在子类不变 而protected内的内容 只有类本身 和类的子类可以访问，对象是无法访问的！

除了在继承上 他跟private没有任何区别！

private无法继承 也就是说子类也不能用基类的 private...

private只对本类可见 protected 对本类和继承类可见

eg：

```
class shape_base
{
        friend class Shape;
protected:
        virtual shape_base* clone() const { return new shape_base(*this);}
public:
        virtual double area() const { return 0; }
        virtual double circumference() const { return 0; }
        virtual bool check(){ return true; }

};

//定义Circle类
class Circle:public shape_base
{
protected:
        Circle* clone() const { return new Circle(*this);}
        double radius;
public:
        Circle() : radius(0) {}                         //如果写了带参数的构造函数就要显示地写默认构造函数！！
        Circle(double r):radius(r){}                            //带参数的构造函数
        Circle(std::istream& is){ is >> radius; }                //从输入流读入数据的构造函数
        double area() const { return PI*radius*radius;}           //定义虚函数
        double circumference() const { return 2*PI*radius;}
};
```

10）this指针；

- 非静态成员函数有this指针，而友元函数没有this指针。
- 友元函数是不能被继承的，就像父亲的朋友未必是儿子的朋友。

- this指针是类的一个自动生成、自动隐藏的私有成员，它存在于类的非静态成员函数中（即静态成员函数没有this指针），指向被调用函数所在的对象。全局仅有一个this指针，当一个对象被创建时，this指针就存放指向对象数据的首地址。使用时：一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return *this；另外一种情况是当参数与成员变量名相同时使用this指针，如this->n = n （不能写成n = n）。

11）静态属性（类属性，静态成员变量），静态属性的初始化方法、访问方法；
12）静态成员函数；

静态成员函数的声明除了在类体的函数声明前加上关键字static，以及不能声明为const或者volatile之外，与非静态成员函数相同。出现在类体之外的函数定义不能制定关键字static。类的静态成员函数无法直接访问普通数据成员（可以通过对象名间接的访问），而类的任何成员函数都可以访问类的静态数据成员。
静态成员函数没有this指针。

13）类模板；
见前面：）

14）友元

8．关于友元函数的描述中，错误的是 B
A．友元函数不是成员函数
B．友元函数只能访问类中私有成员
C．友元函数破坏隐藏性，尽量少用
D．友元函数说明在类体内，使用关键字friend

- 友元函数是指某些虽然不是类成员却能够访问类的所有成员的函数（可以访问对象的私有成员，但普通函数不行），但它不能直接访问类的成员，只能访问对象的所有成员，调用友元函数时，在实际参数中需要指出要访问的对象。
- 必须在类的说明中说明友元函数，说明时以关键字friend开头，后跟友元函数的函数原型，友元函数的说明可以出现在类的任何地方，包括在private和public部分；
- 注意友元函数不是类的成员函数，所以友元函数的实现和普通函数一样，在实现时不用"::"指示属于哪个类，只有成员函数才使用"::"作用域符号；
- 友元可以是函数与类的关系，即友元函数，也可以是类与类的关系，即友元类。

- 友元关系不能继承，是单向性且不具有传递性。
- 一个类的成员函数也可以作为另一个类的友元，但必须先定义这个类。

- 继承：

1）静态绑定和动态绑定；
2）继承与多态；
3）运行时多态（通过虚函数，基类指针或基类引用实现）
和编译时多态（通过函数模板、类模板、函数重载实现）；

运行时多态是动态多态，其具体引用的对象在运行时才能确定。
编译时多态是静态多态，在编译时就可以确定对象使用的形式。

4）向上类型转换、基类引用；
5）虚函数（纯虚函数）；

14．为了实现运行时的多态性，派生类需重新定义基类中的虚函数

- 虚函数与纯虚函数 在他们的子类中都可以被重写。它们的区别是：
  （1）纯虚函数只有定义，没有实现；而虚函数既有定义，也有实现的代码。
  纯虚函数一般没有代码实现部分，如
  virtual void print() = 0;
  而一般虚函数必须要有代码的实现部分，否则会出现函数未定义的错误。
  virtual void print()
  { printf("This is virtual function\n"); }
  （2）包含纯虚函数的类不能定义其对象，而包含虚函数的则可以。
- 虚函数：

6）继承类对象的构建过程；
7）继承类对象的析构过程；

8）虚析构函数；
虚析构函数是为了解决这样的一个问题：基类的指针指向派生类对象，并用基类的指针删除派生类对象。
当且仅当类里包含至少一个虚函数的时候才去声明虚析构函数。

- 数据结构上的操作

遍历、插入、删除；
搜索：二分法，顺序法；
排序：冒泡、选择排序。

- 库函数（STL）

容器类：string, vector, list
迭代器（5类）：理解掌握；
标准算法类：能够编写算法（函数模板）。

S标准库【这部分来自书后的目录】

标准库是C++标准的一个主要部分。下面每一节都讲了一个或者一组标准库中相关的类，对类提供的相关功能进行了详细的讲解。

一般而言，标准库在名字空间std里定义了所有的名字。所以在用到标准库的函数时，必须在函数名前加上std::前缀或者用using申明该函数可以被使用。不过在本节中，我们忽略了该前缀。例如，输出函数会写成cout的形式，而非std::cout的形式。

S.1 输入输出

1. 基础

container<T>::iterator	申明输入-输出类和相关的操作。
cout	与标准输出流相关联的ostream类型对象。
cin	与标准输出流相关联的istream类型对象。
is>>t	忽略空白字符，从is中读取一个值到t。
os<<t	以一种与t变量的类型匹配的格式，把t的值送到流os中。
is.get(c)	从is中读取下一个字符（包括空白字符）放在变量c中。
is.unget(c)	取消最近一次从流is中读出的一个字符的操作，和is.get的动作相反。

2. 迭代器

istream_iterator<T> in(is);	把in定义为一个输入迭代器，可以使用它来从输入流is中读取T类型的值。
ostream_iterator<T> out(os,const char* sep=” “);	把out定义为一个输出迭代器，可以用它来从输出流os中输出T类型的值。在输出每个元素之后输出一个sep分隔符。默认分割符是空字符。

3. 文件流

#include	为一个与文件关联的流定义输入-输出函数。

<fstream>	
ifstream is(cp)	定义了is，并把它连接到cp参数指定的文件上。ifstream派生自istream。
ofstream os(cp)	定义了os，并将其连接到cp参数指定的文件上。ofstream派生自ostream。

4．流控制输出格式

#include <ios>	定义了streamsize类型，这个一个带符号的整数类型，适用于来表示一个输入-输出缓冲区的大小。
os.width() os.width(n)	返回一个streamsize类型的值，表示与os相关联的输出流的宽度。
os.precision() os.precision(n)	返回一个streamsize类型的值，表示与os相关联的输出流的精度。

5．控制符

#include <iomanip>	申明了除endl（在<iostream>中定义）意外的控制符。
os<<endl;	结束当前行输出并刷新与os关联的流的缓冲区。
os<<flush;	刷新与os关联的流的缓冲区。
os<<setprecision(n) os<<setw(n)	分别等价与os.precision(n)和os.width(n)。

6．错误与文件结尾

strm.bad()	返回一个布尔类型，用来表示对strm的最近一次操作是否因为无效数据而失败。
strm.clear()	在一个无效的操作之后重置strm，使之可以在此使用。如果重置strm失败，则抛出ios::failure异常。
strm.eo	

f()	返回一个布尔类型，表示strm是否已经到了文件结尾处。
strm.fail()	返回一个布尔类型，用来表示对strm的最近一次操作是否因为硬件问题或者系统底层问题而失败。
strm.good()	返回一个布尔类型，用来表示对strm的最近一次操作是否成功。

S.2 容器和迭代器

本节中讨论的容器包括：顺序容器vector和list，关联容器map，还有string类。如果没有特别的理由，通常使用vector容器；如果需要在容器的任意地方插入或者删除几个元素，而不仅仅在尾部操作，此时应当使用list容器。

1. 共有的容器操作

所有容器与string类都提供下面的操作。

container<T>::iterator	一种与container<T>关联的迭代器类型，该类型的对象可以用来改变存储在容器中的值。	
container<T>::const_iterator	一种与container<T>关联的迭代器类型，该类型的对象只能可以用来读取在容器中的值。	
container<T>::reverse_iterator container<T>::const_reverse_iterator	一种用来沿反向顺序访问容器中的元素的迭代器类型，与前面的两种迭代器类型相对应。	
container<T>::size_type	无符号整数类型，用来描述容器的容量。	
container<T>::value_type	容器所存储的元素类型。	
c.begin() c.end()	如果容器中元素的话，分别表示指向容器的首元素的迭代器和指向末元素后面一个元素的迭代器。返回的迭代器的类型取决于容器c的类型。	
c.rbegin() c.rend()	用来沿相反的顺序访问c容器中元素的迭代器。	

container<T> c;	定义c为一个空的容器，此时c.size()的值为0。	
container<T> c2(c);	定义c2为一个容器，而且令c2容器的大小等于c容器的大小。C2中的每一个元素都是c中相应元素的一个复制。	
c=c2 ;	复制c2容器中的元素到c容器中（覆盖c中容器中相应的元素），并返回c作为左值。	
c.size()	c容器中元素的个数。	
c.empty()	如果c为空，则返回真，否则返回假。	
c.clear()	清空c容器。与c.erase(c.begin(),c.end())具有相同的作用。函数执行完毕后，c容器的大小变为0。	

2．顺序容器的操作

顺序容器包括vector和list，同时string类也具有相同的性质。

container<T> c(n,t);	定义容器c并对c进行初始化：c包含n个元素，每个元素都是t的一个复制。
container<T> c(b,e);	定义容器c并对c进行初始化：c容器包含的值为b、e两个迭代器所指序列内容的复制。
c.insert(it,t) c.insert(it,n,t) c.insert(it,b,e)	在容器c中it所指的位置前，插入一个元素。
c.erase(it)	

c.erase(b,e)	删除容器c中it所指向的元素，或者删除[b,e]范围内的元素。
c.assign(b,e)	
c.front()	返回指向容器c首元素的一个引用。如果容器c为空，则无定义。
c.back()	返回指向容器c末元素的一个引用。如果容器c为空，则无定义。
c.push_back(t)	把t的一个复制加入容器c的末尾：该操作使得容器的大小增加1.返回void。
c.pop_back()	从容器c中删除末元素。返回void。如果容器为空，则该操作没有定义。
inserter(c,it)	返回一个输出迭代器，在it迭代器指向的元素前面向容器c插入新的元素。该函数在<iterator>中定义。
back_inserter(c)	返回一个输出迭代器，可以通过调用c.push_back函数向容器c的末尾加入一个新的元素。该函数在<iterator>中定义。

3．其他顺序容器的操作

c[n]	一个指向容器c中第n个对象的引用：该容器的头元素是第0个元素。【只对vector和string类对象有效】
c.push_front(t)	在容器c的起始位置插入一个t的复制。返回void。【只对list类型对象有效】
c.pop_front()	删除容器c的首元素。返回void。如果容器c为空，则该函数的行为没有定义。【只对list类型对象有效】
front_inser	返回一个输出迭代器，可以通过调

ter(c)	用c.push_back函数向容器c的起始位置加入一个新的元素。该函数在<iterator>中定义。

4. 关联容器的操作

关联容器被优化来快速访问容器中的元素，它是基于键来访问容器中的元素。除了1中列出的操作外，关联容器还提供如下的操作：

container<T>::key_type	容器键的类型。如果一个关联容器具有K类型键值和V类型键值，那么该关联容器的value_type类型不是V，而是pair<const K,V>。
container<T> c(cmp)	定义c为空的关联容器，使用判断表达式cmp来为容器中的元素排序。
container<T> c(b,e,cmp)	定义c为空的关联容器，用迭代器b和e界定的元素序列对容器c进行初始化，并使用判断表达式cmp来为容器中的元素排序。
c.erase(it)	从容器c中删除迭代器it所执行的元素。返回值为void。
c.erase(b,e)	从容器c中删除[b,e）范围内的元素。返回值为void。
c.erase(k)	从容器c中删除所有键值为k的元素。返回删除的元素个数。
c.find(k)	返回键值为k的指向元素的迭代器。如果不存在这样的元素，该函数返回c.end()。

5. 迭代器

标准库在很大程度上依赖于迭代器实现其算法数据结构的独立性。迭代器是指针概念的抽象，类似于使用指针来访问数组中的元素，迭代器被用来访问容器中的元素。

在制定标准算法时，迭代器分为不同的类型。

输出迭代器	可以通过该迭代器顺序地遍历整个容器，每次前进一个元素知道容器的末尾，每次能而且只能对一个元素进行一次输出操作。例如，ostream-iterator类是一个输出迭代器，该类的copy函数要求用一个输出迭代器作为函数的第三个参数。

输入迭代器	可以通过该迭代器顺序地遍历整个容器，每次前进一个元素，在前进到下一个元素之前，可以按照需要对当前元素进行任意次读操作。例如，istream-iterator是一个输入迭代器，该类的copy函数要求使用一个输入迭代器作为其前两个参数。
正向迭代器	可以通过该迭代器数序地遍历整个容器，每次前进一个元素直到容器末尾，可以用该迭代器在此访问前一个迭代器指向的元素，而且可以按照需要对该元素进行任意次读或写操作。例如，replace就要求使用正向迭代器作为参数。
双向迭代器	可以通过该迭代器双向地遍历整个容器，每次可以前进或者后退一个元素。例如，list和map类提供双向的迭代器，他们的reverse函数要求使用一个双向迭代器作为参数。
随机访问迭代器	可以使用所有的指针支持的操作来访问容器中的某个元素。例如，vector，string类和C++自带的数组都支持随机访问迭代器。Sort函数要求以随机访问迭代器作为参数。

所有迭代器都支持判断是否相等的操作，而随机访问迭代器还支持所有的关系操作。
一个迭代器可以具有以上的几种迭代器类型：例如，一个前进迭代器同时是一个输入迭代器或者输出迭代器。
迭代器支持的操作如下

++p P++	让p指向容器中的下一个元素，然后把p作为左值返回；p++操作返回p的上一个值的一个复制。
*p	p所指向的元素。
p==p2	
p!=p2	
p->x	
--p p--	

p+n n+p	
p2-p	
p[n]	
p<p2	
P<=p2	
p>p2	
p>=p2	

6. 向量(vector)

#include <vector>	申明了vector类及其相关的操作
v.reserve(n)	为v重新分配内存，使得v空间增大，不需要再分配内存也能容纳下至少n个元素。
v.resieze(n)	为v从新分配内存，使得可以容纳n个元素。该操作使得所有指向v中的元素的迭代器都无效。保存v中的前n个元素。

7. 链表(list)

#include <list>	申明了list类及其相关的操作
l.splice(it,l2)	把l2中的所有元素插入到it所指向的元素前面，并且删除l2中的这些元素。 返回void值。
l.splice(it,l2,it2) l.splice(it,l2,b,e)	在l中it指向的元素的位置前面，插入l2中it2指向的元素，或者[b,e）范围内的元素序列，然后从l2中把相应的元素删除。 返回void值。
l.remove(t	在l中删除所有等于t的元素，或者删除所有令p判

) l.remove_if(p)	断为真的元素。返回void。
l.sort(cmp) l.sort()	对l中的元素进行排序。使用<，或者使用参数中提供的cmp来对元素进行比较。

8. 字符串(string)

#include <string>	申明了string类及其相关的操作
string s(cp);	把s定义为字符串，并把它初始化为cp所指向字符串的一个复制。
os<<s;	把s中的字符输出到os，返回一个指向os的引用；
Is>>s;	从is中读取一个量到s中，覆盖s中原来的内容，返回一个指向is的引用。is中的词使用空白字符（包括空格符、制表符和换行符）来分割。
getline(is,s);	从输入流is中读取一行字符（包括换行符），并且把这行字符（包括换行符）保存到s中，覆盖s中原来的内容。返回一个指向is的引用。
s+=s2;	把s2字符串加到s字符串的末尾，并返回一个指向is的引用。
s+s2	返回一个s字符串和s2字符串连接后的值。
s relop s2	返回一个布尔值，表示这个关系是否为真。String库定义了所有的关系运算符和等式运算符。
s.substr(n,len)	返回一个新的字符串类型对象，该对象中的字符串由s对象中第n个元素开始的len个元素组成。
s.c_str()	生成一个指向一个空字符结尾的字符数组的长来那个指针，该数组中包含s中的字符的一个复制。

s.data()	与s.c_str()类似，不过该函数中的数组不是以空字符结尾。
s.copy(cp, n)	从s中复制前n个字符（不包括空结束符），然后存储到cp指向的字符数组中。

9. 对(pair)

#include <utility>	申明pair类及相关操作。
x.first	一个名为x的pair类对象的第一个元素。
x.second	x对的第二个元素。
pair<K,v> x(k,v);	定义x为一个pair，这个pair的K类型值为k，V类型值为v.
Make_pair(k,v)	生成一个新的pair<K,V>类对象。

10. 图(map)

#include <map>	申明map类及相关操作。
map<K,V> m(cmp)	把m定义为一个新的map。
m[k]	生成一个m中由k索引的元素的一个应用。
m.insert(make_pair(k,v))	在m中键为k指示的位置插入v值。如果相应的值已经存在，该值不会改变。
m.find(k)	返回一个指向键值为k的元素的迭代器。如果这个元素不存值，返回m.end()。
*it	在it迭代器所指向的位置生成一个pair<const K,V>对。It->first代表键，it->second代表值。

S.3 算法：这部分没有完成。

#include <algorithm>	包括常用算法的申明。
accumulate(b,e,t) accumulate(b,e,t,f)	在<numeric>头文件中定义。生成一个临时对象obj，这个临时对象与t具有相同的类型和值。对于b,e范围内的每个输入迭代器it，计算obj=obj+*it或者obj=f(obj,*it)。结果是obj对象的一个复制。注意，+可能被重载。
binary_search(b,e,t)	返回一个布尔类型的值。表示t的值是否存在与由两个正向迭代器b和e所界定的范围内。
copy(b,e,d)	