# 1.cin和cout

C++应当有自己的最高级别标准库

## 1.1 >> <<的操作符重载

>> （流的）插入符 << 提取符

流：格式化并保存字节的对象

Istream ostream

Cin 一个输入流**对象 >>**

Cout 输出流对象 << 发现各种变量里有什么

Endl 插入一个新行并清空流

Cout << flush 只清空流

Cout << hex << “0x” << I << endl 把基数变为16进制的算子（hex，oct，dec）

Cin >> ws 用于提取的操纵算子跳过空格

***extern istream*** cin;

cin object(standard input) normally expects input from the console, but this input **can be redirected** from other sources.

Usage cin >> lvalue; //lvalue左值表达式

①左值表达式必须是基本数据类型

②**不能是void**

③若是指向char的指针，所提取的是一串字符

***extern ostream*** cout;

cout is an object of class ostream (他的类型)that represents the standard output stream. By default, most systems have their standard output set to the console, where text messages are shown, **although this can generally be redirected.**

Usage cout << Expressions;

①表达式的类型必须是基本数据类型

②**不能是void**

③若是指向char的指针，所插入的是一串字符

int **main**()

{

char bob[100];

cin >> bob;

cout << bob << endl;

return 0;

}

//char \*bob = "hello world"; 这种赋值方法不再被支持，虽然C是支持的

可知是cin以空格切断字符串的输入！ 但是cout可正常输出！

获取行输入 get() getline()

流是C++为输入/输出提供的一组类，都放在流库中。流对象总是与某一设备相联系（例如，键盘、显示器、硬盘或打印机），通过使用流类中定义的方法，就可完成对这些设备的输入/输出操作。

输入流：要从流中读取数据，这个流为输入流，输入流库头文件为istream。输入流对象cin对应为键盘。

输出流：要在流中存储数据，这个流为输出流。输出流库头文件为ostream。输出流对象cout对应为屏幕，硬盘既是输入流对象，又是输出流对象。

# 2.string类

## 2.1 构造方法

1) 注意包含<string>而不是<string.h>

2) 或者 using std::string 或者 using namespace std

3) 构造函数

string(const char\*s) // C风格字符串

string(int n, char c)// n个字符c初始化

支持默认构造函数和拷贝构造函数

构造string太长会有length\_error异常

4) 字符操作

支持[]下标，.at(const char& 类型const函数和普通函数)函数返回某位置一个引用，at提供范围检查而【】不提供

Data（）const 返回非NULL的字符数组 c\_str（）const 返回null结尾的字符串

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**（pos默认0）

特性：capacity()当前容量(不必增加内存即可存放的数目), max\_size()可存放最多数量size()当前大小 length()字符串长度 **empty() 当前是否为空**（不是清空）

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

5) 串操作

支持+= assign =赋值 append（好几种append具体参见用法总结）

几种特殊的

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,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

6)比较支持 > < >= <= == 等等 还有int compare（s）

几种特殊的:  
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(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

7) 子串 substr（pos = 0， n = npos）

8) swap(string &s2); 交换

9) 查找函数 几种find（s的xx在当前串的位置）,rfind（倒着查找），

find\_first\_of, //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 （当前串在s中出现的字符 **在当前串中的**位置）

Find\_first\_not\_of,

//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

find\_last\_of（第一个不在串中出现的位置）

find\_last\_not\_of

//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

10) 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

**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**

# 3.存储模式

## 3.1 全局

Static变量

函数中的static变量

类静态变量

不管创建多少类对象，静态成员都只有一个拷贝

**静态数据在类中声明，必须的在类外初始化**

**通常将静态成员声明为私有的**

**静态成员函数只能访问类的静态数据成员，不是该类的成员函数，在类中只有语法的作用**

## 3.2 堆栈

here are three memory allocation means:

-static memory allocation (静态区)

-automatic memory allocation（自动区）

-dynamic memory allocation（动态区 程序员申请）

从高向下是栈local object 从下向上是堆

Global static static-global还有用户自己申请的

## 3.3 堆

t

A\* p = new A;

A\* q = new A(10); // 可以赋初值，调用初始化函数；

Delete p；

Delete[] p; （对自定义类数组什么的要这样删除，不过如果是内置类型也无所谓，最好的风格是都加上）

int \*pi = new int[10]; //array

//sometime later

**delete[] pi;**

# 4. 指向对象的指针

## 4.1 通过指针调用函数->

# 5.动态内存申请

## 5.1 new delete

## 5.2 带[]的（前面已经说过了）

从运行结果中我们可以看出，delete p1 在回收空间的过程中，只有 p1[0] 这个对象调用了析构函数，其它对象如 p1[1]、p1[2] 等都没有调用自身的析构函数，这就是问题的症结所在。如果用 delete[]，则在回收空间之前所有对象都会首先调用自己的析构函数。

基本类型的对象没有析构函数，所以回收基本类型组成的数组空间用 delete 和 delete[] 都是应该可以的；但是对于类对象数组，只能用 delete[]。对于 new 的单个对象，只能用 delete 不能用 delete[] 回收空间。

**所以一个简单的使用原则就是：new 和 delete、new[] 和 delete[] 对应使用。（记住这个就行了！）**

# 6. 引用

a reference is just a alias of an exist object（就是一个既存对象的别名，没有创建新对象）

什么时候使用？函数参数？返回类型？

引用作为参数可以修改函数外面的值，这比指针更明晰好用

**对象而不是对象引用作为函数参数会再调用一次构造函数！注意！**

**引用必须被初始化；且被初始化到一个之后不可以再转换到另一个，始终绑定！**

int a=1;

int& refint1 = NULL; //×

**int& refint2 = 5; //× because 5 is constant**

**const int& refint2 = 5; //对于字面常量的引用可以用const int&**

const int &refint3 = a; **//常量引用可以引用变量，但是不能通过这个引用修改常量**

**返回函数中局部变量的局部引用是不允许的（但是不会error），除非他是静态的**

**有的时候返回引用不一定是真的把引用返回了（可能是值），要看返回类型是不是引用**

**实际上元定义变量只要你按照引用返回也是左值，因为引用和元定义本身没区别……**

Member function不可以返回非常量引用

有一种操作：

**Int &getA();**

**然后在类外int& a = getA（）；**

**可以用引用访问私有变量！！！！**

# 7.常量

Have type information

-value **must be initialized**

-Compiler won't let you change it

-internal linkage,unless you make an explicit extern declaration:

extern const int bufsize = 100;

* **const int bufsize=100;**
* **Outline:**

**-Normally, bufsize is only a symbol，have no storage space, be a compile-time constant（编译时就初始化好了）**

但对于自定义对象来说是要调用构造函数的；

用非常量初始化数组是不可行的；

区别 const int\*(也可以指向int，只是不能通过指针修改) int const\*（可以修改指向的值，但不能改成别的值）

可以用另一个非const的指针指向过去，在数组中，来代替他的功用；

const char \*cp = "hello world";

cp = "hahaha";

这种情况，指向的值是可以变的，但是字符串中的元素不可以变

在一些编译器里定义时不加const会报错

**const char\*** v() 返回**// Returns address of static character array**

For built-in types, it doesn’t matter whether you return by value as a const. Returning by value as a const becomes important when you’re dealing with user-defined types.

使用初始化列表初始化常量成员

class bob{

private:

~~const int size = 20;~~

int array[size]; // 

}; 、、错误

class bob{

private:

enum {size = 20};

int array[size]; //ok

}; //正确

**只有const函数才能调用const对象**

# 8.封装，类和对象关系，类的定义（略）

# 9.头文件

头文件应该只用于声明对象、函数声明、类定义、类模板定义、typedef和宏，而不应该包含或生成占据存储空间的对象或函数的定义。

// 头文件header.h

extern void Foo1(); //function declaration

extern int a1; //object declaration

class A; // forward declaration

class B

{

private: A\* pa;

};

void Foo2() //function definition error

{ }

int a2; //object definition error

所有头文件都应该使用#define 防止头文件被多重包含(multiple inclusion)。

我们倾向于减少包含头文件，尤其是在头文件中包含头文件。

//oneHeadFile.h

class File; //can’t be: #include "file/base/file.h"

class A{

private:

File \*pf;

**File &rhf;**

**static File m\_f; //引用和静态都可以**

public:

File oneFunction(File f);

};

//oneHeadFile.h

#include “file/base/file.h” //can’t be class File;

class A{

private:

File f;

public:

File oneFunction(File f)

{ …..}

};

class FileClass; //前置声明类FileClass

class MyClass

{

public:

FileClass GetFileClass(); //允许

private:

FileClass\* fileClass; //允许

};

<http://blog.csdn.net/fallStones/article/details/6266632>

假设有一个Date类

Date.h

[cpp] view plain copy

class Date {

private:

int year, month, day;

};

如果有个Task类的定义要用到Date类，有两种写法

其一

Task1.h

[cpp] view plain copy

class Date;

class Task1 {

public:

Date getData();

};

其二

Task2.h

[cpp] view plain copy

#include "Date.h"

class Task2 {

public:

Date getData();

};

一个采用前置声明，一个采用#include<Date.h>加入了Date的定义。两种方法都能通过编译。但是 Task1.h 这种写法更好。如果Date.h 的 private 成员变量改变，比如变成 double year, month, day; ，Task1.h 不需要重新编译，而 Task2.h 就要重新编译，更糟的是如果 Task2.h 还与其他很多头文件有依赖关系，就会引发一连串的重新编译，花费极大的时间。可是事实上改变一下写法就可以省去很多功夫。

所以能用前置声明代替#include 的时候，尽量用前置声明

有些情况不能用前置声明代替#include

比如Task1.h改成

[cpp] view plain copy

class Date;

class Task1 {

public:

Date d;

};

会编译错误，因为Date d定义了一个Date类型变量，编译器为d分配内存空间的时候必须知道d的大小，必须包含定义Date类的Date.h文件。

这是可以采用指针来代替

[cpp] view plain copy

class Date;

class Task1 {

public:

Date \*d;

};

指针的大小是固定的。在32位机上是4字节，64位机上是8字节。这时编译Task1的时候不需要Date的大小，所以和Date的定义无关。

1. A继承至C
2. A有一个类型为C的成员变量
3. A有一个类型为C的指针的成员变量
4. A有一个类型为C的引用的成员变量
5. A有一个类型为std::list<C>的成员变量
6. A有一个函数，它的签名中参数和返回值都是类型C
7. A有一个函数，它的签名中参数和返回值都是类型C，它调用了C的某个函数，代码在头文件中
8. A有一个函数，它的签名中参数和返回值都是类型C(包括类型C本身，C的引用类型和C的指针类型)，并且它会调用另外一个使用C的函数，代码直接写在A的头文件中

1，没有任何办法，必须要获得C的定义，因为我们必须要知道C的成员变量，成员函数。

2，需要C的定义，因为我们要知道C的大小来确定A的大小，但是可以使用Pimpl惯用法来改善这一点，详情请

看Hurb的Exceptional C++。

3，4，不需要，前置声明就可以了，其实3和4是一样的，引用在物理上也是一个指针，它的大小根据平台不同，可能是32位也可能是64位，反正我们不需要知道C的定义就可以确定这个成员变量的大小。

5，不需要，有可能老式的编译器需要。标准库里面的容器像list， vector，map，

在包括一个list<C>，vector<C>，map<C, C>类型的成员变量的时候，都不需要C的定义。因为它们内部其实也是使用C的指针作为成员变量，它们的大小一开始就是固定的了，不会根据模版参数的不同而改变。

6，不需要，只要我们没有使用到C。

7，需要，我们需要知道调用函数的签名。

1. 视情况而定。

# 友元

友员函数在类中声明，但不是该类的成员函数，不能通过this指针调用。 声明为友元的函数可以访问该类的私有变量。