程序：数据处理

调试：就是看数据对不对，正不正确，在那个地方错的，

打断点：更能看数据

看log：更能看流程

断点能跟数据，

Log能看流程

能不放在头文件中的东西尽量不要放在头文件中，放的越少，代表引入的越少，依赖的越少，影响的越少

Ui一般是创建，更新

阅读代码的时候一定要注意交互的部分，数据流向的部分

C++:  
面向过程

面向对象：封装，多态（满足不同的需求），继承（复用）

泛型（模板）编程：stl

数组这东西太危险，还是用vector比较好，能扩展，能取得大小，能够根据大小来存取，更容易扩展和安全

传指针有传指针的好处，可以修改原来的值，

传值有传值的好处，可以不依赖原来的东西，避免crash

堆上有堆上的好处：长久存在，但需要手动管理内存

栈上有栈上的好处：函数结束会进行释放，不用去考虑内存管理

类型：属于一种抽象，

对象：是真正有作用的实例

抽象一定要尽量抽象出能够复用的东西，比如新手引导，如果能把具体的行为抽象出来，我们只要在每一步去组合它的行为就行啦

代码应该是尽量无重复的，所以，用于映射，用于转化的代码应该是无重复的

重复的越多，bug越多

对一个变量的操作，最好在它的作用域中进行操作

一般不要企图把一个变量传到函数中去，然后在这个函数中改变这个变量的值

如果我们非要把一个变量传到函数中去，然后在这个函数中改变这个变量的值，那么我们应该传这个变量的引用，或者地址进去

但最好不要用这样的方式去改变一个变量的值，因为变量的改变最好在它的作用域之内去改变，而不应该让它在外部改变，这样可能会让原来的地方存在许多未知，除非我们知道在做什么，否则一般不要这么用

引用我们一般用来减少复制，增快速度，还有就是避免浅拷贝，所以：

引用一般用const引用，这样的话，减少了复制的过程，加速了程序的运行速度

组合，把它放进来，

继承，就是它，能够进行具体的实现，能实现多态，基类一般是包含基本的东西（共同的属性和方法）

聚合，把它传进来，聚合属于强依赖的，最好不要用，要用也最好传入的是个基类就行

集成：继承

组合：在类里面包含具体的对象，需要去创建一个具体的实例，生命周期一定比整体小

聚合：我们只是把它传进来（只是用到它，不持有它），不需要去创建一个具体的实例；生命周期可以比整体长（一般比整体长），对象已经存在，

多用局部变量可以减少所占的内存空间，但运行的效率会受损，

多用静态变量，会多占用内存空间，但运行的效率会更高（不用同时建立两次）

能放在.cpp中的尽量放在.cpp中，放在.h中可能会引起重复定义，.h中不要有命名空间，否则会被多处引用

代理代替处理，创建它的代替它进行处理。

一般用在父亲替孩子处理，孩子用到父亲中的某些东西，为了避免环形的结构，就可以用代理的形式，让父亲成为代理，代替孩子进行处理

代理能够让代码不是写死的，不是完全依赖一个的，能够实现多态，能够通过代理进行不同的处理

用delegate的好处是不用相互依赖，让它是一个单向的

函数一定要考虑它的重入性

String: 是属于堆上的

能用引用的地方就用引用，避免复制带来的消耗

能用const的地方就用const

If-else 中相同的东西就尽量放在外面，在块中我们只是去设置它的值就行啦

绝对路径：是从盘符开始的路径，形如C:\windows\system32\cmd.exe,　绝对路径：绝对路径就是你的主页上的文件或目录在硬盘上真正的路径

相对路径：是从当前路径开始的路径，如当前路径为C:\windows,相对路径：相对与某个基准目录的路径

单列一般会有个getInstance用来new，有一个delete用来销毁

纯虚函数和虚函数还是有一些差别的;

虚函数需要有实现，只有实现了才能进行重写

而纯虚函数是不需要有实现的。

虚函数具有多态的效果，重写就能跑到重写的函数中，没有重写就跑到原来的函数中，这就能满足不同的需求

比如多个客户端，都继承自一个，重写就跑到自己客户端中，没有重写，就用它原来的。

VC++ Directories中设置头文件搜索路径和库文件搜索路径

在C/C++–>Preprocesser中设置全局的预处理常量。

在debug中可以设置账号和连接的服务器ip和端口

动画都应该有回调，有时甚至不只要一个回调

回调是个好东西，比如动画什么的，应该都有一个回调

回调函数：对结果进行处理的函数

1. 函数指针，指向函数的指针，只要是这样类型的函数就能用这样的指针,函数具有一定的作用域，所以在定义函数指针时如果不是全局的应该指明它的作用域  
     
   在一般情况下确实是这样的。在这种情况下，test、可自由转换为函数指针，尽管正规的用函数名来表示函数指针的方式是：&test。  
     
   但是，当作为sizeof操作数时是一个例外：函数不会转化为函数指针。因此，不要指望 sizeof(test)、 sizeof(&test)会得到期望的结果。只有 sizeof(&test)是有定义的（因为&test是一个函数指针类型），其它的就看编译器是怎么处理的了（尽管不符合标准规定）。

sizeof是计算[数据类型](http://zhidao.baidu.com/search?word=%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B&fr=qb_search_exp&ie=utf8)所占的字节数的[运算符](http://zhidao.baidu.com/search?word=%E8%BF%90%E7%AE%97%E7%AC%A6&fr=qb_search_exp&ie=utf8)，sizeof的结果等于对象或者类型所占的内存字节数。sizeof（）中，（）后为“空[结构体](http://zhidao.baidu.com/search?word=%E7%BB%93%E6%9E%84%E4%BD%93&fr=qb_search_exp&ie=utf8)”（不含数据成员）的大小不为0，而是1。试想一个“不占空间”的变量如何被取地址、两个不同的“空[结构体](http://zhidao.baidu.com/search?word=%E7%BB%93%E6%9E%84%E4%BD%93&fr=qb_search_exp&ie=utf8)”变量又如何得以区分呢于是，“空[结构体](http://zhidao.baidu.com/search?word=%E7%BB%93%E6%9E%84%E4%BD%93&fr=qb_search_exp&ie=utf8)”变量也得被存储，这样编译器也就只能为其分配一个字节的空间用于占位

 函数名会退化成函数指针，

变量名和数组名、函数名都代表了一片存储区，数组名和函数名在运行时会自动退化为指针，即首地址，

每个函数都只有一份定义，函数名就代表了这一片存储空间，在运行时，函数名会自动退化为指针，每个函数只要有它的入口地址就能调用到这个函数，通常这个入口地址就是函数名，可以用一个函数指针来保存

对于类中的函数，静态函数是属于整个类的，可以直接调用，非静态成员函数，必须通过它的对象来调用，所以函数名应该带上具体的对象才能算是真正的函数名

在sizeof中，sizeof（数组名）数组名不会退化为指针表示整个数组大小，而sizeof（函数名），函数名不会退化为指针，而且函数名不是一种数据类型，所以相当于为空，返回1，只有取地址后，真正表示函数指针，才能用sizeof

Sizeof（test（））是调用对返回值求大小

当类后有<>的很有可能它是一个模板类，是一个由类模板实例化的类

头文件包含关系：

1. 要防止重复引用:条件编译(ifndef def …endif), #parogam once 只引用一次
2. 要说明谁先开始 谁先开始就在谁中先说明一下，说明引用到的数据类型
3. 只能用指针（因为不知道大小，所以只能用指针）

#define CC\_SAFE\_RETAIN(p) do { if(p) { (p)->retain(); } } while(0)

在宏中用while（0）可以在宏中可以由多个语句，多个处理

C++中头文件相互包含的几点问题

**一、类嵌套的疑问**

C++头文件重复包含实在是一个令人头痛的问题，前一段时间在做一个简单的数据结构演示程序的时候，不只一次的遇到这种问题。假设我们有两个类A和B，分别定义在各自的有文件A.h和B.h中，但是在A中要用到B，B中也要用到A，但是这样的写法当然是错误的：  
class B;

class A  
{  
      public:  
          B b;  
};

class B  
{  
      public:  
          A a;  
};  
因为在A对象中要开辟一块属于B的空间，而B中又有A的空间，是一个逻辑错误，无法实现的。在这里我们只需要把其中的一个A类中的B类型成员改成指针形式就可以避免这个无限延伸的怪圈了。为什么要更改A而不是B？因为就算你在B中做了类似的动作，也仍然会编译错误，表面上这仅仅上一个先后顺序的问题。  
      为什么会这样呢？因为C++编译器自上而下编译源文件的时候，对每一个数据的定义，总是需要知道定义的数据的类型的大小。在预先声明语句class B;之后，编译器已经知道B是一个类，但是其中的数据却是未知的，因此B类型的大小也不知道。这样就造成了编译失败，VC++6.0下会得到如下编译错误：  
      error C2079: 'b' uses undefined class 'B'  
将A中的b更改为B指针类型之后，由于在特定的平台上，指针所占的空间是一定的（在Win32平台上是4字节），这样可以通过编译。

**二、不同头文件中的类的嵌套**

      在实际编程中，不同的类一般是放在不同的相互独立的头文件中的，这样两个类在相互引用时又会有不一样的问题。重复编译是问题出现的根本原因。为了保证头文件仅被编译一次，在C++中常用的办法是使用条件编译命令。在头文件中我们常常会看到以下语句段（以VC++6.0自动生成的头文件为例）：

#ifndef(AFX\_STACK\_H\_\_1F725F28\_AF9E\_4BEB\_8560\_67813900AE6B\_\_INCLUDED\_)  
#define AFX\_STACK\_H\_\_1F725F28\_AF9E\_4BEB\_8560\_67813900AE6B\_\_INCLUDED\_  
      //很多语句……  
#endif

意思是如果没有定义过这个宏，那么就定义它，然后执行直到#endif的所有语句。如果下次在与要这段代码，由于已经定义了那个宏，因此重复的代码不会被再次执行。这实在是一个巧妙而高效的办法。在高版本的VC++上，还可以使用这个命令来代替以上的所有：  
      #pragma once  
它的意思是，本文件内的代码只被使用一次。

      但是不要以为使用了这种机制就全部搞定了，比如在以下的代码中：

//文件A.h中的代码  
#pragma once

#include "B.h"

class A  
{  
      public:  
          B\* b;  
};

//文件B.h中的代码  
#pragma once

#include "A.h"

class B  
{  
      public:  
          A\* a;  
};

这里两者都使用了指针成员，因此嵌套本身不会有什么问题，在主函数前面使用#include "A.h"之后，主要编译错误如下：  
      error C2501: 'A' : missing storage-class or type specifiers  
仍然是类型不能找到的错误。其实这里仍然需要前置声明。分别添加前置声明之后，可以成功编译了。代码形式如下：

//文件A.h中的代码  
#pragma once

#include "B.h"

class B;

class A  
{  
      public:  
          B\* b;  
};

//文件B.h中的代码  
#pragma once

#include "A.h"

class B;

class B  
{  
      public:  
          A\* a;  
};

这样至少可以说明，头文件包含代替不了前置声明。有的时候只能依靠前置声明来解决问题。

Static 在.h中，.h是等效替换，所以所有引用这个.h的文件中都会声明一份该变量，他们同名但是是不同的份，也只作用于改文件中

Stl:

对于map而言

如果你不期望没有就插进去的话，最好不要用[]，

除非你真的期望没有就插进去