**C++与C的区别**

1、C语言面向过程、逐步求精，主要执行结构为顺序、选择、循环

2、C++是面向对象，将各个数据和方法放在一起作为一个整体，从现实世界中抽象分离各个元素之间的关系。通过暴露在外（public）的接口与外接发生关系，程序流程有运行中决定

3、面向对象使用计算机的逻辑来描述客观世界的存在

4、C++实现了封装、继承、多态。

5、C++具有异常处理

6、C++ STL

**多态性**

函数重载、虚函数（对应着编译时的多态性和运行时的多态性）（虚函数的重写）

判定他是不是动态多态性即是判断他是不是在运行时判断的（晚绑定（用纯虚函数实现））

多态性的作用即是可以对模块进行适当的封装，并且实现代码复用和接口重用

纯虚函数：在基类中声明，但没有任何定义，要求任何派生类都要实现自己的响应方法（有点像java里面的接口）定义方法：virtual void function=0

为什么会有纯虚函数：通常基类本身生成对象是很不合理的，比如基类可以派生出老虎、狗等动物。但动物本身形成对象是不合理的。

含有纯虚函数的类就叫做抽象类，他不能够实例化生成对象

**动态绑定如何实现**

使用对象指针+‘->’+函数名的方式调用C++类里面的虚函数才会执行动态绑定。（非虚函数不具备动态绑定的特征）

所谓动态绑定，就是基类的指针或引用可能指向不同的派生类对象

虚函数是动态绑定的基础，动态绑定是实现运行时多态性的基础

触发动态绑定需要有两个条件

1、虚函数

2、通过基类类型的引用或者指针进行函数调用

**C++中四种类型转换方式**

先说一下c风格的：（Type）EXPRESSION （有时候不安全）

C++风格的：

1、static\_cast：最常用的诸如int a;float b; int c=(int)b、float d=static\_cast<float>(a)

2、const\_cast：用于取出const属性，把const类型的指针变为非const类型的指针

如const int \*fun(int x,int y){} 转换为非const的 int \*ptr=const\_cast<int \*>(fun(2,3))

3、dynamic\_cast：用于检查该转换是否类型安全，但只在多态类型时合法，即该类至少具有一个虚拟方法。Dynamic\_cast与static\_cast具有相同的基本语法，dynamic\_cast主要用于类层次间上行转换和下行转换，还可以用于类之间的交叉转换。在上层转换时，他等同于static\_cast（或者说效果一样），下行转换的时候，dynamic\_cast具有类型检查功能，相比于static\_cast更加安全。

class C

{

　　//…C没有虚拟函数

}；

class T{

　　//…

}

int main()

{

　　dynamic\_cast<T\*> (new C);//错误

}

此时如改为以下则是合法的：

class C

{

public:

　　virtual void m() {};// C现在是 多态

}

  4、reinterpret\_cast：interpret是解释的意思，reinterpret即为重新解释，此标识符的意思即为数据的二进制形式重新编译，但是不改变其值。

如int i;char \*ptr='hello friend';i=reinterpret\_cast<int>(ptr);

这个转换方法很少使用

**5、操作符重载**

定义方法：

1、类成员函数实现重载：类名+operator+操作符名

2、友元函数：

用途：比如实现某个点类操作符重载，实现‘+’完成坐标x和x相加，y和y相加

**6、内存对齐的原则**

1、结构体变量的首地址能够被其最宽基本类型成员大小与对齐基数中的较小者所整除

2、结构体每个成员相对于结构体首地址的偏移量（offset）都是该成员大小与对齐基数中的较小者的整数倍，如有需要编译器会在成员之间加上填充字节（internal adding）

3、结构体的总大小为结构体最宽基本类型成员大小与对齐基数中的较小者的整数倍，如有需要编译器会在最末一个成员后加上填充字节。

有点绕口？

换一个解释方式

1、数据成员对齐原则：结构（struct或联合union）的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员存储的起始位置要从该成员大小的整数倍开始（比如第一个成员是int，在32位机上是4字节，则要从4的整数倍地址开始存储）

2、结构体作为成员，如果一个结构体有某些结构体的成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。（struct a里存有struct b,b里面有char，int，double等元素，那么元素b应该从8的整数倍开始存储）

3、收尾工作，结构体的总大小也就是sizeof的结果，必须是其内部最大成员的整数倍，不足的要补齐。

比如

struct A

{

Char c1;

Int I;

Short s;

Int j;

}a;

和

Struct B

{

Int I;

Int j;

Short s;

Char c1;

}b

结构A没有遵守字节对齐的原则，结构B遵守了。

Sizeof(a)=16

Sizeof(b)=12

**7、模板的实现**

定义：

Template <class(也可以用typename) T>

返回类型 函数名（形参表）

{函数体}

举例：

Template <class T>

T min(T x,T y)

{return (x<y)?x:y;

功能：使用了一套代码就解决了因为重载函数定义不全而带来的调用错误。模板实现了参数类型化，即把参数类型定义为参数

**8、指针和const的用法**

const是一个c语言的关键字，他限定一个变量不允许被改变。使用const在一定程度上可以提高程序的安全性和可靠性

指向常量的指针：

Const int \*pa;

Int const \*pa;

两者等价。因为指向常量的指针有时候会指向常量（这什么鬼句子……），所以它具有这个性质：“不能靠解引用改变它指向的对象的值”，以此保护他所指向的常量的常量性。

如：\*pa=d //不可行 d是已经声明过的整形

Pa=&d; //可行，因为指针本身的值是可以改变的

指向常量的指针有时候也会指向变量

Int t,u;

Const int \*pa;

Pa=&t;

Pa=&u

常量指针

Int \*const pa=&n //n是之前已经声明过的整型变量，注意，必须是变量

‘常量指针’即指针本身的值是常量，但能靠解引用改变指向它的对象的值

Pa=&d //不行，d是以经声明过的整形

\*pa=d //可行，d是已经声明过的整形

因为常量指针也是一种const常量，所以他同样必须在第一次声明时就初始化，不过他的初始值缩小为只能是变量（的地址），因为只有变量才能确保以后能靠解引用而改变它指向的对象的值。这使得常量指针不像一般的const常量，用常量或变量初始化都可以。

也就是说，常量指针反而总是指向变量的。

是不是有点绕

我自己的总结就是

const修饰的是谁，谁就不可变

Const char\* a;在这里，const修饰的char\*，则char\*也就是地址不可一变，即\*a不可以变。但是a的是可变的，对a++是合法的，但是\*a++是不合法的

Char\* const a修饰的是a也就是char，则对a是不可变的，a++不合法

有个总结表

1)、const在前面

const int nValue； //nValue是const

const char \*pContent; //\*pContent是const, pContent可变

const (char \*) pContent;//pContent是const,\*pContent可变

char\* const pContent; //pContent是const,\*pContent可变

const char\* const pContent; //pContent和\*pContent都是const

2)、const在后面，与上面的声明对等

int const nValue； // nValue是const

char const \* pContent;// \*pContent是const, pContent可变

(char \*) const pContent;//pContent是const,\*pContent可变

char\* const pContent;// pContent是const,\*pContent可变

char const\* const pContent;// pContent和\*pContent都是const

**9、虚函数，纯虚函数、析构函数**

析构函数：当一个对象的生命周期结束时，会自动执行析构函数（要注意的是析构函数并不是用来删除类的对象，而是在删除对象所申请的内存之前做一些清理工作）

定义

Class A{

Public:

A(void);

A(int x);

~A(void) //析构函数

}

为什么析构函数要设置为虚函数：它所存在的意义是基类的指针指向派生类的对象，可以用基类的指针删除派生类的对象。

New a //执行构造函数

Delete a //执行析构函数

另外要说明的是：定义一个函数为虚函数，不代表该函数未被实现，只是用来实现多态的。

定义一个函数为纯虚函数，才表示函数未被实现，定义它只是为了实现一个接口，起一个规范作用。当一个类中有一个函数是纯虚函数的时候，这个类叫做抽象类（这个类不能被实例化），所有继承自这个抽象类的派生类都要实现这个纯虚函数。

**10、内联函数**

当编译器发现某端代码在调用一个内联函数的时候，她并不是调用该函数，而是将该函数的代码整段插入到当前位置。内联函数的本质是，节省时间但是消耗空间。

一些限制：

1、递归函数不能为inline

2、循环语句和switch语句，无法设置为inline（如果有的话，编译器不会把当做内联函数）

3、由于增加体积这个特性，建议inline函数内的代码量应该很小，最好不要超过5行。

4、在调用一个内联函数之前，这个函数一定要已经声明为inline

如

Void foo();

Foo();

之后代码段才有

Inline void foo()

程序最终将没能实现inline

Inline void foo():

Void foo(){}

这样程序也是没能实现inline的

Void foo(int x,int y)

Inline void foo(int x,int y)

这时候如果类声明中的成员函数将自动的变为内联函数

Class a

{

Public:

Void foo(int x,int y){…} //自动成为内联函数

}

实现了inline

5、在一个文件中定义的内联函数不能在另一个文件中使用。他们通常放在头文件中共享。

内联函数的优点：既有宏代码的效率，又增加了安全性。还可以自由的操控类成员。

内联函数与宏定义的区别：

1、宏定义只是简单的文本替换，不做任何安全性的检查也不像函数那样在栈上创建新的变量空间。

如：

#define F(x) x+x

#define F3(x) x+x+x

int i = 0;

cout<<F(i++)<<endl;//1 <=> (i++ + i ++) i is added twice

cout<<i<<endl;//2

int j = 0;

cout<<F3(j++)<<endl; //3

cout<<j<<endl; //3

2、宏定义没有类型检查，可能传进来任意的类型

3、宏定义无法操作类的私有成员，inline可以

**11、const和typedef**

未完待续……

**13、链接指示： extern "C"的作用**

使用extern "C"的修饰的变量和函数是按照的C语言的方式编译和连接的

比如C++支持函数重载，而C语言则不支持。

C++被编译后在符号库中的名字与C语言的不同,比如同样一个

void foo(int x,int y)

函数经过C编译器后在符号库中的名字为\_foo，而C++编译器则会像\_foo\_int\_int之类的名字（不同编译器可能生成的名字不同，但是都采用了相同的机制，生成的名字称为mangled naem）

正是通过包含函数名\_函数参数\_类型信息等等，C++实现了函数重载（是不是可以理解先绑定是什么意思了~）

**15、为什么构造函数不能为虚函数**

构造函数不能声明为虚函数，析构函数可以声明为虚函数，而且有时是必须声明为虚函数。

不建议在构造函数和析构函数里面调用虚函数。

构造函数不能声明为虚函数的原因是:

1 构造一个对象的时候，必须知道对象的实际类型，而虚函数行为是在运行期间确定实际类型的。而在构造一个对象时，由于对象还未构造成功。编译器无法知道对象 的实际类型，是该类本身，还是该类的一个派生类，或是更深层次的派生类。无法确定。。。

2 虚函数的执行依赖于虚函数表。而虚函数表在构造函数中进行初始化工作，即初始化vptr，让他指向正确的虚函数表。而在构造对象期间，虚函数表还没有被初 始化，将无法进行。

虚函数的意思就是开启动态绑定，程序会根据对象的动态类型来选择要调用的方法。然而在构造函数运行的时候，这个对象的动态类型还不完整，没有办法确定它到底是什么类型，故构造函数不能动态绑定。（动态绑定是根据对象的动态类型而不是函数名，在调用构造函数之前，这个对象根本就不存在，它怎么动态绑定？）

编译器在调用基类的构造函数的时候并不知道你要构造的是一个基类的对象还是一个派生类的对象。

**16、动态链接库的优缺点**

1 静态链接库的优点

(1) 代码装载速度快，执行速度略比动态链接库快；

(2) 只需保证在开发者的计算机中有正确的.LIB文件，在以二进制形式发布程序时不需考虑在用户的计算机上.LIB文件是否存在及版本问题，可避免DLL地狱等问题。

2 动态链接库的优点

(1) 更加节省内存并减少页面交换；

(2) DLL文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性；

(3) 不同编程语言编写的程序只要按照函数调用约定就可以调用同一个DLL函数；

(4)适用于大规模的软件开发，使开发过程独立、耦合度小，便于不同开发者和开发组织之间进行开发和测试。

3 不足之处

(1) 使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费；

(2) 使用动态链接库的应用程序不是自完备的，它依赖的DLL模块也要存在，如果使用载入时动态链接，程序启动时发现DLL不存在，系统将终止程序并给出错误信息。而使用运行时动态链接，系统不会终止，但由于DLL中的导出函数不可用，程序会加载失败；速度比静态链接慢。当某个模块更新后，如果新模块与旧的模块不兼容，那么那些需要该模块才能运行的软件，统统撕掉。这在早期Windows中很常见。

**33、智能指针**

由于C++没有自动内存回收机制，每次new出来的内存都要手动delete。经常会出现忘记delete导致内存泄露等等情况。智能指针便可以有效地解决缓解这类问题。包括**std::auto\_ptr、boost::scoped\_ptr、boost::shared\_ptr、boost::scoped\_array、boost::shared\_array、boost::weak\_ptr、boost:: intrusive\_ptr**

对于编译器来说，智能指针实际上是一个栈对象，并非一个指针类型。在栈对象生命周期结束的时候，智能指针通过析构函数释放它管理的堆内存。所有的智能指针都重载了“operator->"操作符，直接返回对象的引用，用以操作对象。访问智能指针原来的方法需要用"."操作符.

访问智能指针里面的裸指针可以使用get()函数，要判断裸指针是否为空，可以使用if(ptr.get()).

智能指针包含了一个reset()方法，如果不传递参数（或者传递NULL），则智能指针会释放当前管理的内存。如果传递一个对象，则指针会释放当前的对象，来管理新传入的对象。

stl库中包含了auto\_ptr这个智能指针，但是他有以下一些问题

1、使用operator=赋值之后，之前的对象就无效了

2、使用release函数不会释放对象，仅仅是归还所有权

3、std::auto\_ptr最好不要当成参数传递

4、由于std::auto\_ptr的oprator=的问题，在其对象中不能放入std::vector等容器（尽管感觉一般也不会放入vector……）

总结来说，auto\_ptr仅仅是服务一个对象堆的内容

这些问题通常情况下可以简单地，使用boost::scoped\_ptr来代替，scoped\_ptr也是服务单个内存堆对象。

同样，boost库里面有也有服务单个数组的scoped\_array，对象需要共享的share\_ptr和share\_array

总结：

1、在可以使用boost库的情况下，拒绝使用std::auto\_ptr，因为其不仅不符合C++的编程思想，而且极容易出错

2、在确定对象无需共享的情况下，使用bosst::scoped\_ptr（动态数组使用boost::scoped\_array)

3、在对象需要共享的情况下，使用boost::shared\_ptr（动态数组用share\_array）

4、在需要访问boost::share\_ptr对象，而又不想改变其引用计数的情况下，使用boost::weak\_ptr，一般常用于软件框架设计中

5、使用智能指针尽量的替代delete

**34、C++中类对象的内存结构**

简述：C++中，类如果有虚函数，就会有一个虚函数表的指针，在类对象最开始的内存数据中。之后是类中的成员变量的内存数据。对于子类，最开始的内存数据记录着父类对象的拷贝（包括父类虚函数表指针和成员变量）。之后是子类自己的成员变量数据。对于子类的子类，也是同样的道理，无论继承了多少个子类，对象中始终只有一个虚函数表指针。

图示

Child 
Base 
Base-*-gm 
C 一 三 冖 一 
Base 
'Jfptr 
Base 
'Jfptr 

现在看一个简单的例子

首先写一个基类

class Base

{

public:

virtual void f() { cout << "Base::f" << endl; }

virtual void g() { cout << "Base::g" << endl; }

virtual void h() { cout << "Base::h" << endl; }

int base;

protected:

private:

};

接下来，我们通过多种不同的继承情况来说明子类的内存对象结构

1、无虚函数集的继承

考虑以下的这个子类

class Child1 : public Base

{

public:

virtual void f1() { cout << "Child1::f1" << endl; }

virtual void g1() { cout << "Child1::g1" << endl; }

virtual void h1() { cout << "Child1::h1" << endl; }

int child1;

protected:

private:

};

这个子类没有继承任何一个基类的虚函数，反映在虚函数表中的效果就是

intchfl 冖 11, 
int base; 
int base; 
Childl 
Base 
'Jfptr 
'Jfptr 
C 一 三 冖 一 一 … … h 一 C) 
C 一 三 冖 一 一 … … g 一 C) 
C 一 三 冖 一 一 … … f 一 C) 

可以看到，在子类的虚函数表中，先存放积累的虚函数表，再存放子类自己的虚函数

2、有一个虚函数的继承

class Child2 : public Base

{

public:

virtual void f() { cout << "Child2::f" << endl; }

virtual void g2() { cout << "Child2::g2" << endl; }

virtual void h2() { cout << "Child2::h2" << endl; }

int child2;

protected:

private:

};

这里子类覆盖了父类的虚函数f()，对应的，编译器会将子类的虚函数表中对应的父类的虚函数替换为子类的函数

v h)tr 
int base, 
Lit base, 
int childl; 
Base O 
Base K) 
Base u) 
Base, •go 

3、全部虚函数都继承

class Child3 : public Base

{

public:

virtual void f() { cout << "Child3::f" << endl; }

virtual void g() { cout << "Child3::g" << endl; }

virtual void h() { cout << "Child3::h" << endl; }

protected:

int x;

private:

};

mt base, 
frit base; 
int ctuldl, 
CYLId2X) 

4、多重继承的情况

多重继承的情况下，子类的内存结构和单一继承有所不同

Basel 
Ufptr 
Basel 
Child 
Basel 
Basel 
Child 
Base2 
Ufptr 
Basel 
Ufptr 
Base 
Ufptr 
Base 

Base3 
Vfptr 
int base3; 
Child5 
Vfptr 
int base3; 
Vfptr 
int base3; 
int chfld5; 
Basel: f3() 
Basel z 
Child2 
Base4: 
Base4: 
Base4 
Vfptr 
int base4; 
Child' f3() 
Basel z 
Child' f5() 
Child5 
Child5 h5() 
Base4::h4() 

5、虚继承

int base; 
v fptr 
int chM, 
v mtr 
int base, 
Base: O 
Base.Ä) 
Ch11d:O 
Child X) 
Base<) 
Base 

虚集成的内存继承顺序和普通继承不同。虚拟继承的子类，有单独的虚函数表，另外也单独保存一份父类的虚函数表，两部分之间用一个4个字节的0x00000000来作为界。子类的内存中，首先是自己的虚函数表，然后是子类的数据成员，然后是0x0，之后是父类的虚函数表，父类的数据成员。如果子类没有自己的虚函数，那么子类就不会有虚函数表，但是子类数据和父类数据之间，还是需要0x0来分割。因此，在虚继承中，子类和父类的数据，是完全间隔的，先存放子类自己的虚函数表和数据，中间以0x0分界，最后保存父类的虚函数和数据。如果子类重载了父类的虚函数，那么则将子类内存中父类的虚函数表的相应函数替换。

**35、C++空间配置器**

STL空间配置器，主要分为三个文件实现，stl\_construct.h（定义了全局函数construct()和destroy()，负责对象的构造和析构。stl\_alloc.h文件中定义了一、二两级配置器，彼此合作。主要思想是先申请一个大块内存池，然后小块内存直接从这个内存池中申请，当不够用的时候在申请新的内存池。这个做法有两个优点：

1、小对象的快速分配。小对象是从内存池分配的，这个内存池是系统调用一次malloc分配一块足够大的区域给程序备用，当内存池耗尽的时候再向系统申请一块新的区域。整体过程类似于批发和零售。

2、避免了内存碎片的生成。程序中的小对象的分配极易造成内存碎片，给操作系统的内存管理带来压力。使用内存池的方法，一操作系统的角度来看，只是一大块内存使用，看不到小对象的分配和释放，避免了内存碎片的生成。

**37、C++中一种实现单例模式的方式**

class CS

{

public:

static CS \*ins()

{

return m\_pCS;

}

int func(){return 0;}

private:

CS()//构造函数私有

{

if (0==m\_pCS)//保证一个实例

{

m\_pCS = new CS();

}

}

static CS \*m\_pCS;

};

CS\* CS::m\_pCS=0;

void main()

{

CS\* pCS = CS::ins();

int n = pCS->func();

}

此方法要注意的是要提供一个Release方法来实现单例对象的析构，析构完成之后要给

\*m\_pCS指针置为空

**38、strcpy与memcpy的区别**

strcpy和memcpy都是标准C函数

strcpy是字符串复制，同时仅仅用于字符串复制（因为函数声明 char \* strcpy(char \*,const char \*)

memcpy提供了内存的复制，而且对于复制的内容没有限制（比如字符数组，整形，结构体等等），并且可以指定复制的大小

strcpy因为不需要指定长度，所以复制的是整个字符数组，如果字符数组没有'\0'则可能发生溢出。memcpy则是需要指定复制长度  
memcpy声明

void \***memcpy**(void \*memTo, const void \*memFrom, **size\_t** size)

{

if((memTo == NULL) || (memFrom == NULL)) //memTo和memFrom必须有效

         return NULL;

**char** \*tempFrom = (**char** \*)memFrom;             //保存memFrom首地址

**char** \*tempTo = (**char** \*)memTo;                  //保存memTo首地址

while(size -- > 0)                //循环size次，复制memFrom的值到memTo中

       \*tempTo++ = \*tempFrom++ ;

return memTo;

}

**39、C++中静态成员函数和普通成员函数有什么区别**

静态成员函数不依赖类成员的实例，所以静态成员函数并不存在this指针。

在类加载的时候，静态成员函数就会分配内存，通过类名+"::"方式访问静态成员函数

因为静态成员函数没有this指针，所以他只可以访问类的静态成员变量

同时静态成员变量在类外必须进行初始化

**40、C++中的字符串分割**

使用strtok函数

strtok函数的实质上处理的是，strtok在s中查找包含在delim中的字符串并用NULL来替换，直到找到整个字符串。这也就意味着每次调用strtok函数都只能获得一个分割单位，要获得所有的分割单元必须反复调用strtok函数

#include<string.h>

#include<stdio.h>

int main()

{

char s[]="Golden Global View,s disk\*desk";

const char \*d=" ,\*";

char \*p;

p=strtok(s,d); //首次调用需要指定要切割的字符串

while(p)

{

printf("%s\n",p);

p=strtok(NULL,d); //再次调用则需要将第一参数设为NULL

}

}

**44、STL中分配空间内存不足的时候发生的情况**

当new失败的时候，会调用set\_new\_handler设置的函数

比如定义了一下这个函数

void faild()

{

…

}

之后就可以std::set\_new\_handler(faild);设置之后，当new失败的时候，便会自动执行faild函数（相当于回调函数了）

**45、迭代器it++,++it效率（很tricky的问题~）**

定义一个STL容器和其迭代器，遍历这个容器

for(…;…;it++);

for(…;…;++it);

这两个for语句在最终执行的效果是一致的，但是会有一个微小的效率差距

即it++和++it的差别。

it++返回的是临时对象

++it返回的是自身的引用

即在it++的情况下，每进行一次循环，编译器就会创建且销毁一个无用的临时对象

可以看一个简单的代码，就可以了解it++和++it的区别了

这是a++

**int** **int**::operator++(**int**)

{

**int** oldvalue = \***this**;

   ++\***this**;

**return** oldvalue;

}

这是++a

**int**& **int**::operator ++()

{

  \***this** = \***this**+1;

**return** \***this**;

}

**46、C++ STL迭代器失效的问题**

当通过迭代器向一个容器insert或者erase元素时候，如果不获取返回的迭代器，可能会导致迭代器失效

比如已经有了一个vector v

std::vector<int>::iterator iter=v.begin()+2;

v.erase(iter); //此处编写会很可能导致迭代器失效

Iter=v.erase(iter); //正确写法

vector中迭代器几种失效的情况

1、当插入一个元素后 end()返回的迭代器肯定失效

2、当插入一个元素后，first end以及各种迭代器可能失效（capacity空间不够，vector地址重加载迁移）

3、当进行删除操作后，指向删除点和之后的迭代器全部失效

deque中的失效情况

1、deque首尾插入元素不会使任何迭代器失效

2、首尾删除元素只会使指向被删除元素的迭代器失效

3、在deque任何其他位置插入和删除元素都会是指向改容器元素的所有迭代器失效

list删除元素会使指向当前删除节点的迭代器失效