1. 关于赋值
2. 类的赋值运算符重载不可以继承，其他运算符重载可继承。即派生类中的默认赋值运算符与基类中重载的赋值运算符没有任何关系，而派生类中的其他运算符从基类中得到继承。
3. 对象的赋值操作通过“=”实现。默认的赋值运算符，只是将成员对象挨个进行赋值操作（包括私有对象）。
4. 关于默认构造函数
5. 如果没有在类中定义任何构造函数，则编译器会自动创建一个默认无参构造函数。改构造函数会对成员对象进行简单的初始化，即调用他们的无参构造函数，此时必须保证任何对象都具有无参构造函数。
6. 在类中定义了任意一个构造函数，则编译器就不再创建默认无参构造函数。
7. 如果没有定义拷贝构造函数，则会自动生成。因此拷贝构造函数和无参构造函数的默认生成规则是不一样的。默认构造函数与默认的赋值语句一样。
8. 关于函数重写和函数重定义
9. 一般情况下，基类中重载的函数都能集成到子类中。但是如果子类中对这个函数进行任意方式的重定义，则所有继承的其他版本都不可见了，即只能调用子类中重定义的函数，不能调用其他继承的版本。即通过子类指针不能调用继承版本，通过基类指针仍然可以调用。
10. 如果基类中有重载的虚函数，一般情况下子类也能继承这些函数。但是如果子类中以任意方式重写了这个重载的虚函数（包括正常的重写），那么所有继承的版本都不可用。规则同（1）。但是子类中不能改变改重载函数的返回值。
11. C++编译器认为，既然在子类中以任意方式重定义了基类的函数或者虚函数，那么基类中所有重载版本的该函数都在子类中被隐藏，即不能通过子类指针调用，可以用基类指针调用这些隐藏的版本，如果调用时，会根据多态性进行调用。
12. 总之，基类中有一堆重载的函数，有的是虚函数，有的不是，那么派生类中只要出现该函数的同名函数，那么基类中的函数版本都将被隐藏，如果出现的这个函数是基类中虚函数的重写，那么这个函数就具有多态性，即可以通过基类指针进行多态调用。
13. 关于构造函数的初始化参数列表
14. 构造活动从最根处开始，即最顶层的基类开始。在每一层，不管参数表的顺序如何，首先调用基类的构造函数，然后将列表中的其余部分，按照成员对象的声明顺序进行初始化构造。
15. 关于析构函数
16. 默认的析构函数会依次调用成员对象的析构函数。调用的顺序与初始化参数表严格相反。比如成员对象A比B先构造，那么在析构B的时候也应该认为A仍然是可用的，这叫做相关性。
17. 析构函数为什么是虚的？析构是自底层向上层开始析构的，如果以基类指针指向子类对象，那么删除这个基类指针的时候，如果析构函数不是虚的，那么将只会调用基类的析构函数，在子类中定义的变量就不会被删除，造成内存泄露。
18. 析构函数可以是纯虚函数，而且基类的纯析构函数必须提供函数体，因为自底向上析构必然会调用到基类的析构函数。
19. 析构函数中的虚函数调用会发生与构造函数中的虚函数调用同样的问题。析构函数是自底层一层一层向上调用，在调用到父类的时候，子类已经被析构掉，此时如果调用一个虚函数并且成功调用，那么在这个函数中可能访问一个已经销毁的成员，导致发生灾难。
20. 关于虚函数
21. 多态是通过虚函数来实现的。每个具有虚函数的类中都有一个指针指向虚函数表，这个虚函数表中存放着所有虚函数的地址。虚函数表具有继承性质，即如果子类没有重写基类中的虚函数，那么虚函数表中改函数的地址就会使用继承下来的虚函数地址。所以调用一个没有重写虚函数的对象的该函数时，总是调用离他最近的上层的该函数版本。
22. 在构造函数内部调用虚函数，虚机制将不起作用。理由有两个，一是构造函数将首先构造父类的对象，如果在父类的构造函数中进行了虚调用，并且成功调用了子对象的中的函数，那么在该函数中可能有对子对象成员的访问，而这些访问还没有被初始化，可能会造成灾难性的行为。二是，VPTR在构造函数调用时首先被初始化，然而此时它只知道它是属于那个类的------构造函数所在的类，所以虚函数表中的内容是本地函数，进入子类构造函数调用时，VPTR知道自己处于子类，会将虚函数表中的内容进行修改，但写入虚函数表中的仍然不是最底层的函数，而是子类的版本，所以在构造函数中进行虚函数调用，只能调用到本地版本的函数。正因为如此，许多编译器把构造函数中的虚函数设置成本地版本的早捆绑。
23. 虚函数表在编译期间就确定了。在运行期间，给VPTR赋值为虚函数表的地址。
24. 纯虚函数是在虚函数最后加上“=0”。具有纯虚函数的类不能实例化对象，具有纯虚函数的类可以有其他类型的函数，用来供子类继承。如果子类没有实现基类中的纯虚函数，那么子类依然是纯虚函数，不能实例化对象。
25. 纯虚函数不能内联。
26. 待解决问题：如果A的子类B实现了A中的虚函数，那么B的子类C的内存布局会怎样？
27. 默认参数不具有多态性。即基类的虚函数具有多态性，但是函数中的默认参数不具有多态性，下面展示了函数a的多态性及其默认参数的非多态性。

**class** Base

{

**public**:

**virtual** **void** **a**(**int** t=1){cout<<"Base::a(int) "<<t<<endl;};

};

**class** Drived :**public** Base

{

**public**:

**virtual** **void** **a**(**int** t=2 ){cout<<"drived:a(double) "<<t<<endl;}

};

**int** **main**() {

Base \* a=**new** Drived;

a->a(); //输出 drived:a(double) 1

Drived \*b=**new** Drived;

b->a(); //输出 drived:a(double) 2

**return** 0;

}

1. 关于static和extern
2. static修饰局部变量：只是改变了变量的生存期及存储空间，不改变其作用域；static修饰全局变量：表明该变量具有文件作用域，即只在该文件中起作用。
3. static修饰函数时与上面相同，只是改变作用域。
4. static修饰类成员时，表示该成员采用了全局变量的存储方式，但又明确了该变量与类相联系，从而避免了别人对该变量的修改，避免发生名字冲突，保证了安全性。
5. extern的作用就是表示声明，其修饰的变量和函数在其他模块中实现，他的好处就是避免了使用include，加快了编译速度。用于链接阶段。Extern “C” 通知编译器以C风格进行函数的链接。
6. 关于静态成员
7. 某些静态变量、常量和数组，在类内部声明，但要在类外部定义。定义语法：  
   Object A:object（）；
8. 如果静态的变量是内置的“整型”，则可以在声明的地方进行初始化，否则一律在类外初始化。
9. 关于友元
10. 可以声明类或结构外部的其它函数、其他结构、其他结构中的函数为友元，这些友元函数或者结构可以访问该类或结构对象的私有成员。
11. 关于protected
12. protected和private类似，只有类本身或者友元能访问private成员，声明为protected的成员除了这些能访问之外，派生类也能访问。
13. protected和private的一个用途是可以将构造函数声明为private或protected，从而禁止在外部实例化对象，只能在类内部实例化对象（singleton模式），如果只想在子类中实例化，则应声明为protected。
14. 关于多重继承
15. 多重继承中的模板。允许用户在编译时指定要混入的类（mixin）-----第2卷，P361.
16. 多重继承中的名字冲突。在一个继承层次中如果发生了名称冲突，编译器会选择那个“占优势”的名字。第2卷P373.
17. 使用多重继承的理由。需要新类来显示两个或两个以上的公共接口类；需要向上转换为两个或两个以上的基类类型。
18. 关于接口
19. 接口技术通过纯虚函数来实现。
20. 接口的调用有两种方式：1种是通过基类指针，2种是通过模板。第2卷,P358.
21. 关于const
22. const与#define值替代的区别，#define在进行值替代的过程中，属于预处理阶段，编译器只进行简单的文本替换，不进行类型检查。Const定义的常量在进行值替代的时候，保存了类型信息，可以进行类型检查。
23. 指向常量的指针：const int \* p 或者int const \* p，指针指向一个常量，指针的值可以改变，而指向的内存数据不能改变；常量指针：int \* const p，指针是常量的，即不能改变指针，但是可以改变指针指向的数据；指向常量的常量指针：const int \* const p 或者 int const \* const p。const指针可以给非const指针赋值，如：int \* const p， int \* q=p；只是给q保存地址的拷贝。
24. Const的赋值：可以把一个非const对象地址赋给指向const的指针，不能把const对象赋给非指向const的指针。
25. Char \* p=”Hello world”。Hello world被编译器看作是一个const字符数组，p指向这块内存的首地址，因此不能改变Hello world中的任何字符。可以定义char [] p 这样就可以改变内容了。
26. Const int a=5；作用于内部类型时，在编译期间会进行常量折叠，即将所有出现a的地方用5替代，除非有取a的地址的操作，才会给a分配内存。
27. 临时量为const，因此不能修改。临时量经常出现在函数参数中，如f(g());
28. Const成员变量的生存期为对象的生存期，在对象创建的时候由初始化参数表初始化，static const 可以定义编译期间就存在的常量成员。
29. 动态数组
30. int \*p=new int[size]; 不能使 int p[]=new int[size]，因为new返回的是分配内存的首地址。
31. sizeof(void \*)将得到所指对象的完整大小。将指针强制转换为void \* 将得到所指对象的首地址，这相比于类继承层次中向上的类型转换。
32. new和delete
33. new所做的操作包括使用malloc从堆中申请内存，调用构造函数，然后返回这块内存的首地址。Delete所做的操作首先是调用指向对象的析构函数（这就是为什么析构函数要是虚的），然后释放对象所占的内存空间。所以delete void \*只会释放掉所指的内存空间，由于void \*没有类型信息，就不会调用其析构函数，可能会造成内存泄露。这也是为什么容器中存的指针时，删除容器不会触发delete的原因，因为可能存储的是void \*。
34. new 和delete的重载。重载了全局的new和delete将完全隐藏默认的new和delete，甚至在重载的new和delete里面也不能访问。创建对象的数组时，始终调用的是全局的new。重载new 并将其设为私有，可以限制对象只在栈里创建。
35. 模板参数

（1）类模板参数：  
参数类型1：无类型参数，指示该参数不是一个类型，而是一个编译时的常数  
  
参数类型2：默认参数，参数可以有一个默认值。  
参数类型3：模板参数，参数是一个模板。注意模板参数和普通类型参数的区别。模板参数表示该参数是一个模板，而模板形式的普通类型参数表示该参数是一个具体类型的模板。

**template** <**class** **T**,**bool** **N**=**true**>//N不是一个类型，只是一个常数

**class** Stack

{

};

1. 指针运算
2. 指针可以进行加减运算，加减运算将导致编译器改变指针的值：根据指针类型进行偏移计算。如A \*a=0; a-=1000; a+=1000;编译器将执行a=a-sizeof(A) \* 1000;注意：由于void没有类型信息，即sizeof(void)为非法运算，所以不能给void指针进行加减运算。Char \* \* p;则p++，编译器执行p+=sizeof(char \*);
3. 两个指针不可以进行加法运算，编译器认为这是非法的。可以进行减法运算，但是必须是同类型的指针表示两个指针的偏移量。这跟（1）中的指针操作是逆运算。
4. 取址运算符“&”，表示的是所取对象类型的一个指针，如：int a，则&a为int \*，int \* a，则&a为int \* \*，若A派生自B，A \*a=new B，则&（\*a）为A \*，因为\*a为A的一个对象。
5. 函数返回对象
6. 在函数中返回对象，函数会利用被返回的对象本身，调用接收返回对象的拷贝构造函数。如果没有接收返回对象的变量，编译器会自动生成一个临时的对象来进行拷贝函数的调用。
7. 函数返回的对象，必然会被拷贝构造函数进行拷贝操作，如果此时正在初始化一个对象（这种情况其实就替代了临时对象的创建），则直接调用该对象的拷贝构造函数，否则将创建临时对象进行拷贝操作。
8. 当拷贝构造函数中出现函数作为参数时如：A a(foo())，不会生成临时变量，直接利用foo中欲返回的对象对a进行拷贝构造。
9. 如果不是在进行对象的初始化工作，那么函数返回的对象必然是一个临时对象，语句执行完后就被销毁。

**class** A

{

**public**:

string id;

**virtual** **~A**(){cout<<id<<" destructed"<<endl;}

**A**(**const** string & i):id(i){cout<<id<<" constructed"<<endl;}

**A**(**const** A &a):id(a.id+" copy"){cout<<id<<" constructed"<<endl;}

A & **operator=**(**const** A &a)

{

cout<<"="<<endl;

**return** \***this**;

}

};

A **foo**(**const** A a)

{

**return** a;

}

**int** **main**()

{

A a(" a");

A b(" b");

b=foo(a);//此处要对b进行赋值操作，

//则首先创建一个临时变量，

//然后进行临时对象和b的赋值操作

A c=foo(a);//此处是在初始化c，

//foo中直接用返回值进行初始化，

//而不生成临时对象,

cout<<"after foo"<<endl;

}

1. 关于自动类型转换
2. 自动类型转换有两种形式：通过构造函数进行自动类型转换，目的类执行转换操作---如

**class** A

{

};

**class** B

{

**public**:

/\*explicit\*/

**B**(**const** A &){};

};

**void** **f**(**const** B& ){}

**int** **main**()

{

A a;

f(a);//调用B的构造函数进行自动类型转换

//在该构造函数前面加上explicit可以避免

}

1. 另一种形式是通过重载运算符，源类执行转换操作。如：

**class** B

{

};

**class** A

{

**public**:

**operator B**()**const**{**return** B();}

};

**void** **f**(**const** B& ){}

**int** **main**()

{

A a;

f(a);//a.operator B()进行自动类型转换

}

1. 关于数组
2. 数组变量不是左值，但是动态创建的数组可以是左值如：

**char** \* \*a=**new** **char** \*[4];

\*a="abc";

\*(a+1)="bcd";

cout<<\*a++<<endl;//正确

**char** \* b[4]={"abc","bcd"};

cout<<\*b++<<endl;//错误，b不是左值

1. 字节对齐
2. 字节对齐遵循三个基本原则：  
   ------结构的首地址能被结构中最宽基本数据类型的sizeof整除。  
   ------每个成员的地址能被该成员的sizeof整除。  
   ------结构体的长度要能被最宽基本数据类型的sizeof整除。

**struct** B //最宽基本数据类型为double=8，

//所以每个B对象的首地址都要被8整除

//可以假设首地址为0（因为能被8整除）

{

**int** a;//a存储在对象的首地址中，0-3

**char** b;//b的首地址为4,

**unsigned** **int** c;//c的首地址为5，因为5不能被

//sizeof(unsigned int)4整除，所以

//在b后面添加3个字节，使c的首地址为8

**double** d; //c为8-11，d首地址为12，不能被8整除，在

//c后面填充4个字节使d的首地址为16

//此时整个长度为4+1+3+4+4+8=24

**char** e; //到此，整个长度为24+1=25,由于25不能被8整除，

//所以需要在最后添加7个字符，使整个长度能被8整除

};

1. 如果定义了#pragma pack(n) ，则上面考虑的有关sizeof的数值，都变成min(sizeof,n)，如整个长度被min(n,最宽类型长度)整除，每个成员的首地址要被min(n,成员的sizeof)整除。
2. 数组的引用

**Int a[3] = {1, 2, 3};**

**Int &b[3] = a;//b为数组a的一个引用**

1. 临时对象

按值返回的函数，是否需要临时对象与函数的调用方式有关。如果调用函数时，左侧是一个尚未初始化的对象，则直接将该对象当作临时对象，不需要再次创建临时对象：

QString a = getString();

编译器很聪明，这时候a的值没有新内容，可以随意更改，就可以把a当成临时对象，如果是下面的，情况就不一样了，必须要创建临时对象保存中间结果：

QString a;

a = getString();

这是因为a的内容不可以随意更改。

此外，getString()函数也可以决定临时对象的形式：

QString getString(){return QString();} //不会产生临时对象

QString getString(){QString a; return a;}// 会产生临时对象，然后用a拷贝构造该临时对象

注意析构的时机。

class A

{

public:

A(){qDebug()<<"A::A()"<<this;}

A(const A& other){qDebug()<<"A:A(const A&)"<<&other<<this;}

A& operator= (const A& other){qDebug()<<"A::="<<&other<<this;}

~A(){qDebug()<<"A::~()"<<this;}

A returnExistint(){A a, b;qDebug()<<"aa()"<<&a;return a;}

A returnNew(){return A();}

};

int main(int argc, char \*argv[])

{

A bb;

bb.returnExistint();

bb.returnNew();

qDebug()<<"\*\*\*\*\*\*\*\*\*";

A cc;

A dd = cc.returnExistint();

A ee = cc.returnNew();

return 0;

}

A::A() 0x3dfcf2

A::A() 0x3dfcbf

A::A() 0x3dfcbe

aa() 0x3dfcbf

A:A(const A&) 0x3dfcbf 0x3dfcef

A::~() 0x3dfcbe

A::~() 0x3dfcbf

A::~() 0x3dfcef

A::A() 0x3dfcee

A::~() 0x3dfcee

\*\*\*\*\*\*\*\*\*

A::A() 0x3dfcf3

A::A() 0x3dfcbf

A::A() 0x3dfcbe

aa() 0x3dfcbf

A:A(const A&) 0x3dfcbf 0x3dfcf0

A::~() 0x3dfcbe

A::~() 0x3dfcbf

A::A() 0x3dfcf1

A::~() 0x3dfcf1

A::~() 0x3dfcf0

A::~() 0x3dfcf3

A::~() 0x3dfcf2