

# C++


date:2022/2/23

## 参考

[C++笔试](#)

C++语言程序设计 郑莉，董渊，何江舟 第四版

## 说明

1. 总结绝大部分来源于上面所说的参考以及网上。
2. 程序结果可能与运行环境有关，尤其是对于sizeof()来说，该文档对应程序的运行结果在程序末尾以注释的方式给出，对应的运行环境为 windows10 | x64 | gcc version 8.1.0。
3. 擅用ctrl+f搜索，如果你用的是Typora的话，它对于万字以上的大文件处理经常力不从心，特别是Ctrl+F搜索可能直接就无响应不跳转。可以点击底部  图标进入源代码编辑模式，再进行搜索就能正常跳转。

## 常见问题

1. 面向对象和面向过程的编程思想，他们有什么区别
  - 面向过程：面向过程语言也就是称为结构化程序和设计，他的核心思想就是认为任何程序都可由顺序、选择、循环三种基本控制结构构造，现在常见的语言基本上就是c语言。
  - 面向对象：具有四个特性：抽象性，封装性，继承性，多态性。他的核心思想是面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。现在常用的语言很多都是面向对象的了。（面向对象的三大特性：封装性，继承性，多态性）
  - 形象的区别：以将大象放进冰箱为例。
  - 优缺点：

	面向过程	面向对象
优点	性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素	易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护
缺点	没有面向对象易维护、易复用、易扩展	性能比面向过程低

2. 对象都具有的二方面特征是什么？分别是什么含义？
  - 对象都具有的特征是：静态特征和动态特征。静态特征是指能描述对象的一些属性，动态特征是指对象表现出来的行为。
3. 在头文件中进行类的声明，在对应的实现文件中进行类的定义有什么意义？
  - 一个头文件被多个CPP包含时编译链接正确，一个CPP文件被多个其它CPP文件包含时编译正确，链接出错，报错为 XXX 已经在 xxx.obj中定义。

- 因为C++中有函数的实现体，每被包含一次就多了一个实现，导致一个函数在不同C++文件中被多次实现，重复了。

头文件被多次包含为什么没问题？关键是每个头文件开头都有宏 `#pragma once`，该宏确保了头文件只会被包含一次。在.h文件中声明一个类，对类的成员函数做定义的时候你可能会有这样一个疑问：“在.h中定义成员函数，被多个cpp include的时候难道不会重复定义吗？”一般我们的写法都是借鉴C中的规范，在.h中对类只做声明，对类的定义放到相应的.cpp文件中去。这样做无疑是最好的也是最规范的做法，但是在.h中对类做定义，是不会像基础数据类型变量那样发生重复定义报错的。我们知道，类的声明和普通变量声明一样都是不产生目标代码的。但是类的定义也不产生目标代码。因此它和普通变量的声明唯一的区别是不能在同一编译单元内出现多次。

- 也利于文件管理，提高文件效率。
- 模板文件的特殊性。
  - 模板文件只有在实例化时才能确定其具体的实现体，所以如果将模板文件的声明和函数体分开在.h和.cpp中，当编译cpp时，并不会产生函数的具体实现体。当在其它文件中 `#include "template.h"`时，会提示找不到函数的定义。
  - 解决方法：在需要使用模板函数的地方，`#include "template.cpp"`，即包含它的C++文件，而不是.h文件。原因：使用模板函数的地方，比如 `addobj<cube>()`，传了具体的模板类型给函数，这样模板函数就能到C++文件中找到对应的实现体将cube传给模板参数而实例化了。

#### 4. 成员函数通过什么来区分不同对象的成员数据？为什么它能够区分？

- 通过 `this` 指针来区分的，因为它指向的是对象的首地址。（比如类的静态成员是属于整个类的，该类的所有对象共享，所以是没有this指针的）

#### 5. C++编译器自动为类产生的四个缺省函数是什么？

- 默认构造函数，复制构造函数，析构函数，重载赋值运算符函数
- 复制构造函数与重载赋值运算符函数调用的时机。重载赋值运算符函数的调用是在当两个对象之间进行赋值时，会自动调用重载赋值运算符函数，它不同于复制构造函数，复制构造函数是用已有对象给新生成的对象赋初值的过程。

```
1 Point a1;
2 Point a2 = a1; // 复制构造函数
3 a2 = a1;       // 重载赋值运算符函数
```

```
1 class A{
2 private:
3     int data;
4 public:
5     A(int val=1){
6         data = val;
7         cout << "调用构造函数" << endl;
8     }
9     A(const A& a){
10        data = a.data;
11        cout << "调用复制构造函数" << endl;
12    }
13    A test(A a){
14        return A(2);
15        // return A(a);
16    }
17    void show(){
18        cout << "data=" << data << endl;
19    }
```

```

20 };
21
22 int main(){
23     A a1, a2;
24     A a3 = a2.test(a1);
25     a3.show();
26     return 0;
27 }
28
29 /*
30 调用构造函数
31 调用构造函数
32 调用复制构造函数
33 调用构造函数
34 data=2
35 */

```

结果解释：对于 `A test(A a){return A(2);}` 来说，只有形实结合时会调用复制构造函数，`A(2)` 调用的是构造函数，而返回该对象时，由于做了优化，已经不在调用复制构造函数了。

```

1  class B
2  {
3  public:
4      B()
5      {
6          cout<<"默认构造函数"<<endl;
7      }
8      B(const B& b){
9          cout << "复制构造函数" << endl;
10     }
11     ~B()
12     {
13         cout<<"destructed"<<endl;
14     }
15     B(int i):data(i)
16     {
17         cout<<"constructed by parameter " << data <<endl;
18     }
19 private:
20     int data;
21 };
22
23 void test(B b){
24     cout << "test" << endl;
25 }
26 B Play( B b)
27 {
28     return b ;
29 }
30
31
32 int main()
33 {
34     // 调用默认构造函数
35     B t3;
36
37     // 调用一次构造函数

```

```

38     test(4);
39
40     // 调用一次构造函数
41     // 一次复制构造函数(初始化t1时)
42     B t1 = Play(5);
43
44     // 调用两次复制构造函数(形参结合时, 初始化t2时)
45     B t2 = Play(t3);
46
47     return 0;
48 }
49
50 /*
51 默认构造函数
52 constructed by parameter 4
53 test
54 destructed
55 constructed by parameter 5
56 复制构造函数
57 destructed
58 复制构造函数
59 复制构造函数
60 destructed
61 destructed
62 destructed
63 destructed
64 */

```

#### 6. 复制构造函数在哪几种情况下会被调用?

- 当类的一个对象去初始化该类的另一个对象时;
- 如果函数的形参是类的对象 (不能是对象的引用), 调用函数进行形参和实参结合时;
- 如果函数的返回值是类对象, 函数调用完成返回时 (注: 当函数返回值是类对象时, 现在的gcc编译器已经做了优化, 返回值为对象时, 不再产生临时对象, 因而不再调用复制构造函数)。 [C++返回值为对象时复制构造函数不执行怎么破。](#)

```

1  class Point{
2      private:
3          int x;
4          int y;
5      public:
6          Point(){
7              x = 3;
8              y = 5;
9          }
10         Point(int _x, int _y){
11             x = _x;
12             y = _y;
13         }
14
15         // 复制构造函数
16         Point(Point& p);
17
18         void show();
19         void setXY(int x, int y);
20         int getX();
21         int getY();

```

```
22 };
23
24 Point::Point(Point& p){
25     x = p.x;
26     y = p.y;
27     cout << "调用复制构造函数" << endl;
28 }
29
30 void Point::show(){
31     cout << "x:" << x << " y:" << y << endl;
32 }
33
34
35 void Point::setXY(int _x, int _y){
36     x = _x;
37     y = _y;
38 }
39
40 int Point::getX(){
41     return x;
42 }
43
44 int Point::getY(){
45     return y;
46 }
47
48 void fun1(Point p){
49     cout << p.getX() << endl;
50 }
51
52 Point fun2(){
53     Point a;
54     return a;
55 }
56
57
58
59 int main(){
60     Point p1, p2;
61
62     cout << "调用复制构造函数的三种情况" << endl;
63     cout << "用类的对象初始化该类的另一个对象" << endl;
64     Point p3(p1);
65     cout << endl;
66
67     cout << "函数形参为Point类" << endl;
68     fun1(p2);
69     cout << endl;
70
71     cout << "返回值为Point类" << endl;
72     Point p4;
73     p4 = fun2();
74     cout << endl;
75     p4.show();
76     return 0;
77 }
78
79 /*
```

```

80 调用复制构造函数的三种情况
81 用类的对象初始化该类的另一个对象
82 调用复制构造函数
83
84 函数形参为Point类
85 调用复制构造函数
86 3
87
88 返回值为Point类
89
90 x:3 y:5
91 */

```

#### 7. 构造函数与普通函数相比在形式上有什么不同？

- 特点：构造函数一般声明为共有的；构造函数的名字必须与类名相同；它不具有任何类型，不返回任何值。
- 作用：构造函数是类的一种特殊成员函数，一般情况下，它是专门用来初始化对象成员变量的。

#### 8. 什么时候必须重写复制构造函数？

- 当构造函数涉及到动态存储分配空间时，要自己写复制构造函数，并且要深拷贝。

#### 9. 构造函数的调用顺序是什么？

- 如果该类有直接或间接的虚基类，则先执行虚基类的构造函数。(虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的)。所谓最远派生类，就是建立对象时所指定的类。
- 如采该类有其他基类，则按照它们在继承声明列表中出现的次序，分别执行它们的构造函数，但构造过程中，不再执行它们的虚基类的构造函数。
- 按照在类定义中出现的顺序，对派生类中新增的成员对象进行初始化。对于类类型的成员对象，如果出现在构造函数初始化列表中，则以其中指定的参数执行构造函数，如未出现，则执行默认构造函数;对于基本数据类型的成员对象，如果出现在构造函数的初始化列表中，则使用其中指定的值为其赋初值，否则什么也不做。
- 执行该类的构造函数的函数体。

#### 10. 哪几种情况必须用到初始化成员列表？

- 类的成员是常量成员初始化；
- 类的成员是对象成员初始化，而该对象没有无参构造函数。
- 类的成员为引用时。

注：复制构造函数是一种特殊的构造函数，所以上面的情况对于复制构造函数同样适用。

```

1  class Point{
2      private:
3          int x;
4          int y;
5      public:
6          Point(int _x, int _y){
7              x = _x;
8              y = _y;
9              cout << "调用Point类构造函数" << endl;
10         }
11
12         // 复制构造函数
13         Point(Point& p);
14
15         void show();

```

```

16         void setXY(int x, int y);
17         int getX();
18         int getY();
19     };
20
21     Point::Point(Point& p){
22         x = p.x;
23         y = p.y;
24         cout << "调用Point类复制构造函数" << endl;
25     }
26
27     void Point::show(){
28         cout << "x:" << x << " y:" << y << endl;
29     }
30
31
32     void Point::setXY(int _x, int _y){
33         x = _x;
34         y = _y;
35     }
36
37     int Point::getX(){
38         return x;
39     }
40
41     int Point::getY(){
42         return y;
43     }
44
45     class Line{
46     public:
47         Line(Point _p1, Point _p2);
48         Line(Line& line);
49         double getLen();
50     private:
51         Point p1, p2;
52         double len;
53     };
54
55     Line::Line(Point _p1, Point _p2): p1(_p1), p2(_p2){
56         cout << "调用Line类构造函数" << endl;
57         double x = p1.getX() - p2.getX();
58         double y = p1.getY() - p2.getY();
59         len = pow(x*x+y*y, 0.5);
60     }
61
62     Line::Line(Line& line): p1(line.p1), p2(line.p2){
63         cout << "调用Line类复制构造函数" << endl;
64         len = line.len;
65     }
66
67     double Line::getLen(){
68         return len;
69     }
70
71     int main(){
72         Point p1(1,1);
73         Point p2(4,5);

```

```

74
75     Line l1(p1, p2);
76     // Line::Line(Point _p1, Point _p2): p1(_p1), p2(_p2)
77     // 会调用四次Point类的构造函数，前两次属于类作为参数，后两次属于用类的对象初始化该类的另一对象
78
79     Line l2(l1);
80     return 0;
81 }
82
83 /*
84 调用Point类构造函数
85 调用Point类构造函数
86 调用Point类复制构造函数
87 调用Point类复制构造函数
88 调用Point类复制构造函数
89 调用Point类复制构造函数
90 调用Line类构造函数
91 调用Point类复制构造函数
92 调用Point类复制构造函数
93 调用Line类复制构造函数
94 */

```

#### 11. 静态函数存在的意义？

- 静态私有成员在类外不能被访问，可通过类的静态成员函数来访问；
- 类的构造函数是私有的时候，不像普通类那样实例化自己，只能通过静态成员函数来调用构造函数。

#### 12. 在类外有什么办法可以访问类的非公有成员？

- 友元函数，类的公有成员函数，继承（对于protected来说）

#### 13. 什么叫抽象类？

抽象类是带有纯虚函数的类。抽象类是为了设计的目的而建立的，它为一个类族提供统一的操作界面。一个抽象类自身无法实例化，也就是说我们无法定义一个抽象类的对象，只能通过继承机制，生成抽象类的非抽象派生类，然后再实例化。

抽象类不能实例化，即不能定义一个抽象类的对象，但是可以定义一个抽象类的指针和引用，以此来实现多态性。

#### 14. 不允许重载的 5 个运算符是哪些？

- .\* (成员指针访问运算符)
- 作用域标识符
- sizeof 长度运算符
- ?: 条件运算符
- . (成员访问符/类属关系运算符)

#### 15. 运算符重载的形式？

- 共有两种，非静态成员函数和重载为非成员函数。其中非成员函数包括普通函数和友元函数，由于一般都是需要访问类的私用成员，所以对于非成员函数来说，一般都是会重载为友元函数。
- 对于成员函数来说，表达式 **obj1 运算符 obj2** 相当于函数调用 **obj1.operator 运算符 (obj2)**。
- 对于非成员函数来说表达式 **obj1 运算符 obj2** 相当于函数调用 **operator 运算符 (obj1, obj2)**。

#### 16. 流运算符为什么不能通过类的成员函数重载？一般怎么解决？



- 因为通过类的成员函数重载必须是运算符的第一个是自己，而对流运算的重载要求第一个参数是流对象。
- 一般通过友元来解决。

#### 17. 赋值运算符和复制构造函数的区别与联系？

- 相同点：都是将一个对象 copy 到另一个中去。
- 不同点：复制构造函数涉及到要新建一个对象。

#### 18. 友元关系有什么特性？

- 单向的，非传递的，不能继承的。

#### 19. const char \*p, char \* const p; 的区别

- 如果 const 位于星号的左侧，则 const 就是用来修饰指针所指向的变量，即指针指向为常量；如果 const 位于星号的右侧，const 就是修饰指针本身，即指针本身是常量。

#### 20. 是不是一个父类写了一个 virtual 函数，如果子类覆盖它的函数不加 virtual ,也能实现多态？

- virtual 修饰符会被隐形继承的。
- virtual 可加可不加,子类覆盖它的函数不加 virtual ,也能实现多态。具体地，系统会根据名称、参数及返回值 3 个方面检查进行检查。
- 该函数是否与基类的虚函数有相同的名称。
- 该函数是否与基类的虚函数有相同的参数个数及相同的对应参数类型。
- 该函数是否与基类的虚函数有相同的返回值或者满足赋值兼容规则的指针、引用型的返回值。
- 派生类的函数满足了上述条件，就会自动确定为虚函数。这时，派生类的虚函数便覆盖了基类的虚函数。不仅如此，派生类中的虚函数还会隐藏基类中同名函数的所有其他重载形式。
- 如果理解类覆盖的概念，虚函数就是派生类对父类函数的覆盖。

#### 21. 函数重载是什么意思？它与虚函数的概念有什么区别？

- 函数重载是一个同名函数完成不同的功能，编译系统在编译阶段通过函数参数个数、参数类型不同来区分该调用哪一个函数，即实现的是静态的多态性。不能仅仅通过函数返回值不同来实现函数重载。
- 而虚函数实现的是在基类中通过使用关键字 virtual 来申明一个函数为虚函数，含义就是该函数的功能可能在将来的派生类中定义或者在基类的基础之上进行扩展，系统只能在运行阶段才能动态决定该调用哪一个函数，所以实现的是动态的多态性。它体现的是一个纵向的概念，也即在基类和派生类间实现。**虚函数是动态绑定的基础，而虚函数的具体实现依赖于赋值兼容规则。虚函数必须是非静态的成员函数。**
- 构造函数与析构函数是最适合的例子，构造函数可以重载，当不能声明为虚构造函数；析构函数不可以重载，但是可以声明为虚析构函数。

#### 22. 什么叫多态，多态分为哪些类型？

- 多态是指同样的消息（[这里所说的消息主要是指对类的成员函数的调用，而不同的行为是指不同的实现](#)）被不同类型的对象接收时导致完全不同的行为，是对类的特定成员函数的再抽象。
- C++ 支持的多态又可以分为 4 类：重载多态（函数的重载，运算符的重载实质也是函数的重载）、强制多态（强制类型转换）、包含多态（主要是指用虚函数实现的多态）和参数多态（类模板实现的多态）。
- 多态从实现的角度来讲可以划分为两类，即编译时的多态（重载多态、强制多态和参数多态）和运行时的多态（包含多态）。前者是在编译的过程中确定了同名操作的具体操作对象，而后者则是在程序运行过程中才动态地确定操作所针对的具体对象。**这种确定操作的具体对象的过程就是绑定/关联(binding)。**

#### 23. 构造函数和析构函数是否可以被重载,为什么？

- 构造函数可以被重载，析构函数不可以被重载。因为构造函数可以有多个且可以带参数，而析构函数只能有一个，且不能带参数。

#### 24. 虚函数动态绑定的实现原理

- 主要是通过虚表实现的，虚表的内容是虚函数的指针，保留每个虚函数的对应的函数入口地址。类的每个对象只需要保存指向虚表首地址的指针。每个对象指针的个数等于该类继承的具

有虚函数类的个数。如果给类继承的不具有虚函数，而自己本身有虚函数，那么该类也会有一个虚表指针。

```
1  class Base1 {
2  private:
3      int v;
4      int* p;
5  public:
6      Base1(){
7          v = 1;
8          cout << "默认构造" << endl;
9      }
10     Base1(const Base1& a){
11         v = a.v;
12         cout << "默认拷贝函数" << endl;
13     }
14     virtual void f(int val){
15         cout << val << endl;
16     }
17     Base1& operator=(const Base1& a){
18         v = a.v;
19         cout << "默认重载赋值运算符函数" << endl;
20         return *this;
21     }
22 };
23
24 class Base2 {
25 private:
26     int val;
27 public:
28     Base2(int v){
29         val = v;
30         cout << "默认构造" << endl;
31     }
32     Base2(const Base2& b){
33         cout << "默认拷贝函数" << endl;
34     }
35     virtual void test(int val){
36         cout << "test" << endl;
37     }
38 };
39
40 class Base3: public Base1, public Base2{
41
42 };
43
44 int main(){
45     cout << "sizeof(Base1)=" << sizeof(Base1) << endl;
46     cout << "sizeof(Base2)=" << sizeof(Base2) << endl;
47     cout << "sizeof(Base3)=" << sizeof(Base3) << endl;
48     return 0;
49 }
50
51 /*
52 sizeof(Base1)=24
53 sizeof(Base2)=16
54 sizeof(Base3)=40
```

### 对上述程序输出的一种错误解释

- `sizeof(Base1)=24`,  $8*2 + 4 = 20$ , 内存对齐 24。
  - 有两个指针, 一个虚表指针, 一个 `int*` 指针, 所以是  $8 * 2$ ; 一个 `int`, 4字节。总共  $8*2 + 4 = 20$ , 后面内存对齐, 24。
- `sizeof(Base2)=16`,  $4+8 = 12$ , 内存对齐 16。
- `sizeof(Base3)=40`,  $8*2 + 8 + 4*2 = 32$ , 两个虚表指针 (注意, 继承了几个含有虚函数的类, 就有几个虚表指针), 所以是  $8 * 2$ ; 一个 `int*` 指针, 一个 `int`, 4字节。总共  $8*2 + 8 + 4*2 = 32$ , 发现和结果对不上。

### 正确的解释

为了能够进行正确的解释, 必须看到在编译器这些类的内存布局。

- 命令行输入 `g++ t.cpp -fdump-lang-class`, (`t.cpp` 是我的程序文件名) 可以专门查看内存布局
- 生成 `t.cpp.0011.class` 文件, 下面是一部分

```

1  vtable for Base1
2  Base1::_ZTV5Base1: 3 entries
3  0      (int (*)(...))0
4  8      (int (*)(...))(& _ZTI5Base1)
5  16     (int (*)(...))Base1::f
6
7  Class Base1
8      size=24 align=8
9      base size=24 base align=8
10 Base1 (0x0x66fc120) 0
11     vptr=((& Base1::_ZTV5Base1) + 16)
12
13
14 vtable for Base2
15 Base2::_ZTV5Base2: 3 entries
16 0      (int (*)(...))0
17 8      (int (*)(...))(& _ZTI5Base2)
18 16     (int (*)(...))Base2::test
19
20 Class Base2
21     size=16 align=8
22     base size=12 base align=8
23 Base2 (0x0x66fcba0) 0
24     vptr=((& Base2::_ZTV5Base2) + 16)
25
26 vtable for Base3
27 Base3::_ZTV5Base3: 6 entries
28 0      (int (*)(...))0
29 8      (int (*)(...))(& _ZTI5Base3)
30 16     (int (*)(...))Base1::f
31 24     (int (*)(...))-24
32 32     (int (*)(...))(& _ZTI5Base3)
33 40     (int (*)(...))Base2::test
34
35 Class Base3
36     size=40 align=8

```

```

37     base size=36 base align=8
38     Base3 (0x0x64d2690) 0
39     vptr=((& Base3::_ZTV5Base3) + 16)
40     Base1 (0x0x6749120) 0
41     primary-for Base3 (0x0x64d2690)
42     Base2 (0x0x6749180) 24
43     vptr=((& Base3::_ZTV5Base3) + 40)

```

解释一下：vtable 是虚表的意思，vptr 是虚指针的意思

```

1  size=24 align=8 // size表示内存对齐之后的大小
2  base size=24 base align=8 // base size表示实际大小

```

看懂了这个，就能明白，错误的原因在于编译器 Base1 实际的大小就是24，可以理解为将类的私有成员内存对齐之后认为是该类的实际大小，之后有虚表指针了需要对齐的话继续对齐。那么可以得到 Base3 的实际大小就是36，之后内存对齐，结果是40。

25. 判断：

- 抽象类不会产生实例，所以不需要有构造函数。✗
  - 一个类需不需要构造函数要看具体情况，和是不是抽象类（虚基类）没有关系。一个类的构造函数的作用是对其成员进行初始化，抽象类也有可能包含需要初始化的成员，也需要进行初始化。
  - 而且每一个类都至少会有一个默认构造函数。
- 从一个模板类可以派生新的模板类，也可以派生非模板类。✓

26. 当一个类 A 中没有声明任何成员变量与成员函数,这时 sizeof(A)的值是多少，如果不是零，请解释一下编译器为什么没有让它为零。

- 此时为1，C++标准规定，完整对象的大小为正数，而且就算是一个空类也可以实例化，如果为空，那么它的存放就不需要分配内存，这是不符合逻辑的。

27. 子类析构时要调用父类的析构函数吗？

- 会调用，析构函数调用的次序是先派生类的析构后基类的析构，也就是说在基类的析构调用的时候,派生类的信息已经全部销毁了，析构函数的调用顺序与构造函数的调用顺序刚好相反。

28. 继承的优缺点

- 在面向对象的语言中，继承是必不可少的、非常优秀的语言机制，它有如下优点：
  - 代码共享，减少创建类的工作量，每个子类都拥有父类的方法和属性；
  - 提高代码的重用性，可扩展性、开放性；
- 缺点：
  - 降低代码的灵活性。子类必须拥有父类的属性和方法，让子类自由的世界中多了些约束；
  - 增强了耦合性。当父类的常量、变量和方法被修改时，需要考虑子类的修改，而且在缺乏规范的环境下，这种修改可能带来非常糟糕的结果——大段的代码需要重构。

29. 解释堆和栈的区别。

- 从C/C++的内存分配（与操作系统相关）上来说，堆（heap），栈（stack）属于内存空间的一段区域。堆用于动态分配内存，由程序员申请分配和释放，采用链式存储结构。栈由编译器自动释放，存放函数的参数值、局部变量等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存放到栈中，这个被调用的函数再为它的自动变量和临时变量在栈上分配空间。每调用一个函数一个新的栈就会被使用。在32位系统下，堆内存可以达到4G的空间（虚拟内存的大小，有面试官问过），从这个角度来看堆内存大小可以很大。但对于栈来说，一般都是有一定的空间大小的（在VC6默认的栈空间大小是1M，也有默认2M的）。

- 从效率方面来看，栈是机器系统提供的数据结构，计算机会在底层对栈提供支持（有专门的寄存器存放栈的地址，压栈出栈都有专门的机器指令执行），这就决定了栈的效率比较高。堆则是C/C++函数库提供的，由于机制复杂，效率较低。

### 30. 何时需要预编译

- 对于需要很长时间生成的大项目，可能需要考虑创建自定义预编译文件。Microsoft C 和 C++ 编译器提供预编译任何 C 或 C++ 代码（包括内联代码）的选项。使用此性能功能，可以编译稳定的代码正文，在文件中存储已编译的代码状态，并在后续编译过程中将预编译代码和仍在开发的代码合并在一起。每个后续编译的速度都更快，因为无需重新编译稳定的代码。
- 预编译代码在开发周期中非常有用，可缩短编译时间，尤其是在以下情况下：
  - 始终使用不常更改的一大段代码。
  - 程序包含多个模块，所有这些模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，所有包含文件都可以预编译为一个预编译标头。
- 第一个编译（创建预编译标头 (PCH) 文件）所花的时间比后续编译长一点。通过包括预编译的代码，后续编译可以更快地进行。
- 可以预编译 C 和 C++ 程序。在 C++ 编程中，通常的做法是将类接口信息分离成头文件。这些头文件以后可以包含在使用类的程序中。通过预编译这些标头，可以缩短程序进行编译所需的时间。

### 31. 虚拟函数与普通成员函数的区别

- 区别：虚拟函数有 virtual 关键字，有虚拟指针和虚函数表，虚拟指针就是虚拟函数的接口，而普通成员函数没有。内联函数和构造函数不能为虚拟函数。

### 32. 内联函数、构造函数、静态成员函数为甚不可以定义为虚函数

- 因为虚函数的作用是为了实现多态（父类指针可以访问子类的函数），即运行时候才知道这个函数具体是什么，那么也就是说提前不把这个函数给写死的函数才能设为虚函数，而static修饰的函数也就是静态函数属于类，编译时候就已经确定它的函数体具体内容了，以后不能再变（即已经被写死了），所以它不能充当虚函数。而内联函数也是编译时候展开的函数实体，即也就是给写死了，所以也不能动态改变，所以也不能作虚函数。此外构造函数也不能作虚函数，因为首先得构造出一个对象，然后才能在这个对象里找到虚函数表，才能调用到对应的函数实体，而构造函数自己就是一个虚函数的话，那么自己都还没创建出来又怎么能找到这个虚函数表呢，所以构造函数是不能设为虚函数的。
- 内联函数和普通函数最大的区别在于内部的实现方面，当普通函数在被调用时，系统首先跳跃到该函数的入口地址，执行函数体，执行完成后，再返回到函数调用的地方，函数始终只有一个拷贝；而内联函数则不需要进行一个寻址的过程，当执行到内联函数时，此函数展开（很类似宏的使用），如果在 N 处调用了此内联函数，则此函数就会有 N 个代码段的拷贝。

### 33. 有时基类的析构函数为什么要声明为虚函数？

- 有可能通过基类指针调用对象的析构函数(通过delete)，就需要让基类的析构函数成为虚函数，
- 如用派生类对象去实例化基类对象的指针，而派生类又申请了内存空间，此时基类的析构函数必须要声明为虚函数。

### 34. 类中 private, protected, public 三种访问限制类型的区别

- 友元函数都可以访问，其中private和protected基本一样，区别仅在于protected在派生类中可以被访问，public可以被类和类对象访问。

### 35. 指针与引用

- 共同点：
  - 使用它们作为形参，都可以通过该参数修改主调函数中的变量以达到参数双向传递的目的。
  - 都可以避免值复制的发生从而减少函数调用时数据传递的开销。
- 区别：
  - 引用必须被初始化，指针不必。



- 引用初始化以后不能被改变，指针可以改变所指的对象。
- 只有常引用，而没有引用常量，也就是说，不能用 `&const` 作为引用类型。这是因为引用只能在初始化时指定它所引用的对象，其后则不能再更改，这使得引用本身(而非被引用的对象)已经具有常量性质了。注意这里说的常量性质的意思，并不是说引用的值不可以改变。
- 不存在指向空值的引用，但是存在指向空值的指针。
- 引用的是指是被引用变量的别名，而指针是内容是变量的首地址。

```

1  int main(){
2      int v1 = 9;
3      int& v2 = v1;
4      int* p = &v1;
5
6      // 从结果可以看出引用本身的地址被隐藏了
7      cout << &v1 << " " << &v2 << endl;
8
9      cout << &v1 << " " << &p;
10     return 0;
11 }
12
13 /*
14 0x61fe14 0x61fe14
15 0x61fe14 0x61fe08
16 */

```

可以说，引用可以做的指针都可以做，但是有些情况下只能使用指针，如：

- 中途需要改变所指向对象的。
- 有时候不能明确所指向的对象，而只能声明为空时。

### 36. 全局变量和局部变量的区别？

- 全局变量是整个程序都可访问的变量，生存期从程序开始到程序结束；局部变量存在于模块中(比如某个函数)，只有在模块中才可以访问，生存期从模块开始到模块结束。
- 全局变量分配在全局数据段，在程序开始运行的时候被加载。局部变量则分配在程序的栈中。因此，操作系统通过变量的分配地址就可以判断出是局部变量和全局变量，编译器可以通过语法规则的分析，判断出是全局变量还是局部变量。
- 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期（在下一次调用的时候还可以保持原来的赋值）。把全局变量改变为静态变量后是改变了它的作用域（仅在本文件可见），限制了它的使用范围。因此static 这个说明符在不同的地方所起的作用是不同的。

### 37. 堆栈溢出一般是由什么原因导致的？

- 函数调用层次太深。函数递归调用时，系统要在栈中不断保存函数调用时的现场和产生的变量，如果递归调用太深，就会造成**栈溢出**，这时递归无法返回。再有，当函数调用层次过深时也可能导致栈无法容纳这些调用的返回地址而造成**栈溢出**。
- 动态申请空间使用之后没有释放。由于C语言中没有垃圾资源自动回收机制，因此，需要程序主动释放已经不再使用的动态地址空间。申请的动态空间使用的是堆空间，动态空间使用不会造成**堆溢出**。
- 数组访问越界。C语言没有提供数组下标越界检查，如果在程序中出现数组下标访问超出数组范围，在运行过程中可能会内存访问错误。
- 指针非法访问。指针保存了一个非法的地址，通过这样的指针访问所指向的地址时会产生内存访问错误。

### 38. 如何引用一个已经定义过的全局变量？

- 用extern关键字

```

2  #include <stdio.h>
3  /*外部变量声明*/
4  extern int g_X ;
5  extern int g_Y ;
6  int max()
7  {
8      return (g_X > g_Y ? g_X : g_Y);
9  }
10 /***main.c***/
11 #include <stdio.h>
12 /*定义两个全局变量*/
13 int g_X=10;
14 int g_Y=20;
15 int max();
16 int main(void)
17 {
18     int result;
19     result = max();
20     printf("the max value is %d\n",result);
21     return 0;
22 }

```

- 用引用头文件的方式(可以在不同的C文件中声明同名的全局变量，前提是其中只能有一个C文件中对此变量赋初值，此时连接不会出错)

```

1  fu1.h:
2  #include<stdio.h>
3  void setone();
4
5  fu1.c:
6  #include"fu1.h"
7
8  int i;
9
10 void setone()
11 {
12     printf("%d/n", i);
13 }
14
15 fu2.h
16 #include<stdio.h>
17
18 void settwo();
19
20 fu2.c
21 #include"fu2.h"
22
23 int i;
24
25 void settwo()
26 {
27     printf("%d/n", i);
28 }
29
30
31 test.c
32 #include"fu1.h"

```

```

33 #include "fu2.h"
34
35 int i=36;    多个文件中唯一的一个初始化赋值，因此不会出现连接问题
36
37 int main(void)
38 {
39     printf("%d/n", i);
40     i =3;
41     setone();
42     i = 6;
43     settwo();
44 }
45 /*运行结果：
46 36
47 3
48 6*/

```

- 区别：已经定义过全局变量，假如用引用头文件的方式引用，两个cpp以上引用这个头文件会出现重复定义。用extern引用只是声明，不会分配内存，不会重复定义。所以用extern引用定义过全局变量。

39. 对于一个频繁使用的短小函数,在 C 语言中应用什么实现,在 C++中应用什么实现?

- c用宏定义（也可以用inline），c++用 inline。

40. C++是不是类型安全的?

- 不是（存在reinterpret\_cast转换）。类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。类型安全的编程语言与类型安全的程序之间，没有必然联系。绝对类型安全的编程语言暂时还没有。
- 所谓类型不安全，是指同一段内存可以用不同的数据类型来解释。
- [C++ 中static cast、dynamic cast、const cast和reinterpret cast总结](#)。

关键字	说明
static_cast	用于良性转换，一般不会导致意外发生，风险很低。
const_cast	用于 const 与非 const、volatile 与非 volatile 之间的转换。通俗地说，const_cast 可以用来将数据类型中的 const 属性去除，const_cast 只用于将常指针转换为普通指针，将常引用转换为普通引用，
reinterpret_cast	高度危险的转换，这种转换仅仅是对二进制位的重新解释，不会借助已有的转换规则对数据进行调整，但是可以实现最灵活的 C++ 类型转换。
dynamic_cast	借助 RTTI，用于类型安全的向下转型（Downcasting）。专门用于指针或者引用从一般到特殊的转换，虽然这个过程也可以通过显示的 static_cast 转换，但是dynamic_cast在转换前会检查指针(或引用)所指向对象的实际类型是否与转换目的类型兼容，如果兼容转换才会发生，才能得到派生类的指针(或引用)。否则会抛出空指针或者异常

对于 static\_cast 和 reinterpret\_cast 这两种转换来说，之所以说前者更加安全，是因为前者使用的是**基于内容的转化**，比如对于整型的 和双精度浮点型的 2. 在内存中是由不同的二进制序列表示的。我们默认转换是基于内容的转换，所以没有问题。而当我们尝试用 reinterpret\_cast 来做转换时，肯定是会出错的。c++ 标准只保证用 reinterpret\_cast 操作符将 A类型的p转换为B类型的q，再用reinterpret\_cast 操作符将B类型的q转换为A类型的r后电应当有 (p= =r) 成立。

static\_cast 同时也不是那么的安全，比如像下面这样：



```

1  int main(){
2      double v1 = 1.1;
3      void* p1 = &v1;
4      int* p2 = static_cast<int*>(p1);
5      cout << *p2 << endl;
6      return 0;
7  }
8
9  /*
10 -1717986918
11 */

```

这个问题就和上面提到的用 `reinterpret_cast` 来做转换时遇到的问题是一样的。

- [C++ static cast、dynamic cast、const cast和reinterpret cast \(四种类型转换运算符\)](#)。

- `dynamic_cast`的使用示例

```

1  class Base1{
2  public:
3      virtual void show(int i = 3)const{
4          cout << "Base1::display()" << i << endl;
5      }
6  };
7
8  class Base2:public Base1{
9  public:
10     void show(int i = 6)const{
11         cout << "Base2::display()" << i << endl;
12     }
13     void test()const{
14         cout << "test" << endl;
15     }
16 };
17
18
19 int main(){
20     // wrong
21     // Base1* p = &b2;
22     // p->test(); // 基类指针调用派生类的函数，只能调用基类中用virtual声明过的
函数，
23
24     // 或者把基类指针转换为派生类指针
25
26     Base2 b2;
27     Base1* p = &b2;
28     Base2* p2 = static_cast<Base2*>(p); // 一般到特殊，必须要显示转换
p2->test();
29
30     // 更好的方法
31     Base1* pp = &b2;
32     Base2* p3 = dynamic_cast<Base2*>(pp); // 一般到特殊，必须要显示转换
33     if(p3 != NULL){
34         p3->test();
35     }
36     else{
37         cout << "类型不兼容" << endl;
38     }

```

```

39
40     return 0;
41 }
42
43 /*
44 test
45 test
46 */

```

◦ const\_cast的使用示例

```

1  int main(){
2      const int val = 5;
3      // int& v = val; // 这一步是不行的，必须要使用常引用
4      int& v = const_cast<int&>(val);
5      v++;
6      cout << val << " " << v << endl;
7      return 0;
8  }
9
10 /*
11 5 6
12 */

```

#### 40. 内存的分配方式

◦ 程序占用的内存分为五个区域：

- 静态区/全局区 (static)  
存放静态变量、全局变量，内存存在程序编译的时候就已经分配好，这块内存存在程序的整个运行期间始终不变。
- 栈区 (stack)  
存放函数的参数值、局部变量的值等，由编译器自动分配释放。管理简单，空间使用效率高，但是生命周期很短暂，分配的内存容量有限。用来存储函数的参数和非静态局部变量。
- 堆区 (heap)  
也叫动态内存分配。程序在运行的时候new申请任意大小的内存，一般由程序员分配释放，如果程序员没有释放掉，程序会一直占用内存，导致内存泄漏，在程序结束后，系统会自动回收。适用范围广，容易出现碎片。由new和delete运算符产生释放的存储空间都是堆空间。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
- 文字常量区 (constant)  
存放常量，不允许修改，程序结束后由系统释放。
- 代码区 (code)  
存放函数体的二进制代码。

#### 41. 在定义一个宏的时候要注意什么？

- 定义部分的每个形参和整个表达式都必须用括号括起来，以避免不可预料的错误发生。因为宏替换实际上就是文本替换。

```

1  #define mul(a, b) (a*b)
2
3  int main(){
4      int val = mul(2+4, 5);
5      printf("%d", val);
6      return 0;
7  }

```

- 正确的做法应该是

```

1  #define mul(a, b) ((a)*(b))
2
3  int main(){
4      int val = mul(2+4, 5);
5      printf("%d", val);
6      return 0;
7  }

```

#### 42. strcpy()和 memcpy()的区别

strcpy和memcpy主要有以下3方面的区别。

- 复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、结构体、类等。
- 复制的方法不同。strcpy不需要指定长度，它遇到被复制字符串的串结束符“\0”才结束，所以容易溢出。memcpy则是根据其第3个参数决定复制的长度。
- 用途不同。通常在复制字符串时用strcpy，而需要复制其他类型数据时则一般用memcpy。

```

1  int main()
2  {
3      char a1[6] = "hello";
4      char a2[10] = "1234567";
5      strcpy(a1, a2);
6      printf("%s %d\n", a1, sizeof(a1));
7      memcpy(a1, a2, sizeof(a2));
8      printf("%s %d\n", a1, sizeof(a1));
9
10     int a3[5], a4[5];
11     for(int i = 0; i < 5; i++){
12         a4[i] = i+1;
13     }
14     printf("%d\n", sizeof(a3));
15     memcpy(a3, a4, sizeof(a3));
16     for(int i = 0; i < 5; i++){
17         printf("%d ", a3[i]);
18     }
19     return 0;
20 }
21
22 /*
23 1234567 6
24 1234567 6
25 20
26 1 2 3 4 5
27 */

```

43. 说明 define 和 const 在语法和含义上有什么不同?

- (相同点就是他们都被视为常量, 在其他地方不可改变)
- 编译器处理方式不同  
#define宏是在预处理阶段展开。  
const常量是编译运行阶段使用。
- 类型和安全检查不同  
#define宏没有类型, 不做任何类型检查, 仅仅是展开。  
const常量有具体的类型, 在编译阶段会执行类型检查。
- 存储方式不同  
#define宏仅仅是展开, 有多少地方使用, 就展开多少次, 不会分配内存。(宏定义不分配内存, 变量定义分配内存。)  
const常量会在内存中分配(可以是堆中也可以是栈中)。
- const 可以节省空间, 避免不必要的内存分配。例如:  
#define NUM 3.14159 //常量宏  
const double Num = 3.14159; //此时并未将Num放入ROM中 .....  
double i = Num; //此时为Num 分配内存, 以后不再分配!  
double l = NUM; //编译期间进行宏替换, 分配内存  
double j = Num; //没有内存分配  
double J = NUM; //再进行宏替换, 又一次分配内存!

44. 说出字符常量和字符串常量的区别, 并使用运算符 sizeof 计算有什么不同?

- 形式上: 字符常量是单引号引起的一个字符 字符串常量是双引号引起的 若干个字符
- 含义上: 字符常量相当于一个整形值(ASCII 值),可以参加表达式运算 字符串常量代表一个地址值(该字符串在内存中存放位置)
- 占内存大小: 字符常量只占1个字节 字符串常量占若干个字节(至少一个 字符结束标志)

```
1  int main()
2  {
3      char* str = "";
4      char arr[] = "";
5
6      // 需要注意第一个计算的是指针本身的大小
7      printf("%d %d %d\n", sizeof(str), sizeof(""), sizeof(arr));
8      return 0;
9  }
10
11 /*
12 8 1 1
13 */
```

45. 简述全局变量的优缺点?

- 全局变量也称为外部变量, 它是在函数(包括main函数)外部定义的变量, 它属于一个源程序文件, 它保存上一次被修改后的值, 便于数据共享, 但不方便管理, 易引起意想不到的错误。

46. 总结 static 的应用和作用

- 函数体内 static 变量的作用范围为该函数体, 不同于 auto 变量, 该变量的内存只被分配一次, 因此其值在下次调用时仍维持上次的值;
- 在模块内的 static 全局变量可以被模块内所用函数访问, 但不能被模块外其它函数访问;
- 在模块内的 static 函数只可被这一模块内的其它函数调用, 这个函数的使用范围被限制在声明它的模块内;
- 在类中的 static 成员变量属于整个类所拥有, 对类的所有对象只有一份拷贝;

- 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能 访问类的 static 成员变量。

#### 47. 总结 const 的应用和作用

- 欲阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要 对它进行初始化，因为以后就没有机会再去改变它了；
- 对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或 二者同时指定为 const；
- 在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；
- 对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类 的成员变量；
- 对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。

#### 48. 什么是常指针，指针常量？

- 指针常量含义是该指针所指向的地址不能变，但该地址所指向的内容可以变化，使用指针常量可以保证我们的指针不能指向其它的变量，

```
1 int a = 5;
2 int* const p = &a;
```

- 常指针的含义是指向常量的指针，是指该指针的变量本身的地址可以变化，可以指向其它的变量，但是它 所指的内容不可以被修改。

```
1 int a = 5;
2 const int* p1 = &a;
3 int const* p2 = &a;
```

#### 49. 函数指针和指针函数的区别？

- 在程序运行时，不仅数据要占据内存空间，执行程序的代码也被调入内存并占据一定 的空间。每一个函数都有函数名，**实际上这个函数名就表示函数的代码在内存中的起始地址**。由此看来，调用函数的通常形式"函数名(参数表)"的实质就是"函数代码首地址(参数表)"
- 函数指针就是专门用来存放函数代码首地址的变量**。在程序中可以像使用函数名一 样使用指向函数的指针来调用函数。

```
1 int f1(int val){
2     return ++val;
3 }
4
5 int f2(int val){
6     return val+3;
7 }
8 int main(){
9     int (*pf)(int);
10    pf = f1;
11    int v1 = pf(2);
12    pf = f2;
13    int v2 = pf(2);
14    printf("%d %d", v1, v2);
15    return 0;
16 }
17
18 /*
```

```
19 | 3 5
20 | */
```

- 指针函数是指函数的返回值是一个指针类型。

#### 50. 指针的几种典型应用情况?

- `int *p[n];`-----指针数组, 每个元素均为指向整型数据的指针。
- `int (*)p[n];`-----`p` 为指向一维数组的指针, 这个一维数组有 `n` 个整型数据。
- `int *p();`-----函数带回指针, 指针指向返回的值。
- `int (*)p();`-----`p` 为指向函数的指针。

#### 51. static 函数与普通函数有什么区别?

- 用`static`修饰的函数, 本限定在本源码文件中, 不能被本源码文件以外的代码文件调用。而普通的函数, 默认是`extern`的, 也就是说, 可以被其它代码文件调用该函数。。因此定义静态函数有以下好处:
  - 其他文件中可以定义相同名字的函数, 不会发生冲突。
  - 静态函数不能被其他文件所用

#### 52. struct(结构) 和 union(联合)的区别?

- 结构和联合都是由多个不同的数据类型成员组成, 但在任何同一时刻, 联合中只存放了一个被选中的成员 (所有成员共用一块地址空间), 而结构的所有成员都存在 (不同成员的 存放地址不同)。
- 对于联合的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于结构 的不同成员赋值是互不影响的。

```
1  union Data
2  {
3      int i;
4      float f;
5      char str[20];
6  };
7  int main(){
8      union Data data;
9      data.i = 10;
10     data.f = 220.5;
11     printf( "data.i : %d\n", data.i);
12     printf( "data.f : %f\n", data.f);
13     strcpy( data.str, "C Programming");
14     printf( "data.i : %d\n", data.i);
15     printf( "data.f : %f\n", data.f);
16     printf( "data.str : %s\n", data.str);
17     printf("%d", sizeof(data));
18     return 0;
19 }
20
21 /*
22 data.i : 1130135552
23 data.f : 220.500000
24 data.i : 1917853763
25 data.f : 41223605803277949000000000000000.000000
26 data.str : C Programming
27 20
28 */
```

#### 53. class 和 struct 的区别?

- 在C++中struct与class除了struct的成员默认是公有的，而类的成员默认是私有的以外。其他的用法均一样。
- 在c中，没有class的概念，struct只是数据的组合，struct内只能声明变量，不能定义变量，不能出现函数。

54. 简述枚举类型？

枚举方便一次定义一组常量，使用起来很方便；

```
1  enum DAY
2  {
3      MON=1, TUE, WED, THU, FRI, SAT, SUN
4  };
5
6  int main()
7  {
8      enum DAY day;
9      day = WED;
10     printf("%d",day);
11     return 0;
12 }
13
14 /*
15 3
16 */
```

55. 局部变量和全局变量是否可以同名？

- 能，局部会屏蔽全局。要用全局变量，需要使用 "::"(域运算符)。

```
1  int global = 9;
2
3  int main()
4  {
5      int global = 5;
6      cout << global << " " << ::global << endl;
7      return 0;
8  }
9
10 /*
11 5 9
12 */
```

56. 在什么时候使用常引用？

- 如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。

57. 怎样消除多重继承中的二义性？

- 作用域分辨符

```
1  class Base1{
2  public:
3      int var;
4      void fun(){
5          cout << "Base1" << endl;
6      }
7  };
```

```

8
9 class Base2{
10 public:
11     int var;
12     void fun(){
13         cout << "Base2" << endl;
14     }
15 };
16
17 class Derived:public Base2, public Base1{
18 // public:
19 //     int var;
20 //     void fun(){
21 //         cout << "Derived" << endl;
22 //     }
23 };
24
25
26 int main(){
27     Derived d;
28     // d.var = 3; // 出错
29     // d.fun(); // 出错
30     d.Base1::var = 3;
31     d.Base2::fun();
32     return 0;
33 }

```

上面出错的原因在于，派生类没有声明与基类同名的成员，那么使用"对象名.成员名"就无法访问到任何成员，来自 Base1 Base2 类的同名成员具有相同的作用域，系统根本无法进行唯一标识，这时就必须使用作用域分辨符。

- 使用using关键字进行澄清

```

1 class Base1{
2 public:
3     int var;
4     void fun(){
5         cout << "Base1" << endl;
6     }
7 };
8
9 class Base2{
10 public:
11     int var;
12     void fun(){
13         cout << "Base2" << endl;
14     }
15 };
16
17 class Derived:public Base2, public Base1{
18 public:
19     using Base1::var;
20     using Base2::fun;
21 };
22
23
24 int main(){

```



```

25     Derived d;
26     d.var = 3;
27     d.fun();
28     d.Base1::var = 3;
29     d.Base1::fun();
30     return 0;
31 }

```

58. 运行过程中的多态（包含多态）需要满足的3个条件？

- 类之间满足赋值兼容规则.
- 要声明虚函数.
- 要由成员函数来调用或者是通过指针、引用来访问虚函数。如果是使用 对象名来访问虚函数，则绑定在编译过程中就可以进行(静态绑定)，而无须在运行过程中 进行。

59. 什么叫智能指针？

- [本文介绍c++里面的四个智能指针: auto\\_ptr, shared\\_ptr, weak\\_ptr, unique\\_ptr 其中后三个是c++11支持，并且第一个已经被c++11弃用。为什么要使用智能指针：我们知道c++的内存管理是让很多人头疼的事，当我们写一个new语句时，一般就会立即把delete语句直接也写了，但是我们不能避免程序还未执行到delete时就跳转了或者在函数中没有执行到最后的delete语句就返回了，如果我们不在每一个可能跳转或者返回的语句前释放资源，就会造成内存泄露。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。](#)
- 在C++中，我们知道，如果使用普通指针来创建一个指向某个对象的指针，那么在使用完这个对象之后我们需要自己删除它，例如：

```

1  ObjectType* temp_ptr = new ObjectType();
2  temp_ptr->foo();
3  delete temp_ptr;

```

很多材料上都会指出说如果程序员忘记在调用完temp\_ptr之后删除temp\_ptr，那么会造成一个悬挂指针(dangling pointer)，也就是说这个指针现在指向的内存区域其内容程序员无法把握和控制，也可能非常容易造成内存泄漏。

可是事实上，不止是“忘记”，在上述的这一段程序中，如果foo()在运行时抛出异常，那么temp\_ptr所指向的对象仍然不会被安全删除。

在这个时候，智能指针的出现实际上就是为了可以方便的控制对象的生命期，在智能指针中，一个对象什么时候和在什么条件下要被析构或者是删除是受智能指针本身决定的，用户并不需要管理。

60. 什么时候需要用虚析构函数？

- 当基类指针指向用 new 运算符生成的派生类对象时，delete 基类指针时，派生类部分没有释放掉而造成释放不彻底现象，需要虚析构函数。

61. 派生新类的过程要经历三个步骤

- 吸收基类成员、改造基类成员、添加新的成员。面向对象的继承和派生机制，其**最主要目的是实现代码的重用和扩充**。因此，吸收基类成员就是一个重用的 过程，而对基类成员进行调整、改造以及添加新成员就是原有代码的扩充过程，二者是相辅相成的。
- 吸收基类成员：继承基类中除构造和析构函数之外的所有非静态成员。
- 改造基类成员：涉及到两个问题
  - 一个是基类成员的访问控制问题，主要依靠派生类 定义时的继承方式来控制。
  - 另一个是对基类数据或函数成员的覆盖或隐藏。

62. memset与fill的区别？

```

1  int main()
2  {
3      int len = sizeof(int)*5;
4      int* arr = new int[5];
5      memset(arr, 1, len); // memset是以字节为单位进行赋值的
6      for(int i = 0; i < 5; i++){
7          cout << arr[i] << " ";
8      }
9      cout << endl;
10     fill(arr, arr+5, 1);
11     for(int i = 0; i < 5; i++){
12         cout << arr[i] << " ";
13     }
14     cout << endl;
15     return 0;
16 }
17
18 /*
19 16843009 16843009 16843009 16843009 16843009
20 1 1 1 1 1
21 */

```

### 63. 类成员函数的重载、覆盖和隐藏区别

- 成员函数被重载的特征：
  - 相同的范围（在同一个类中）；
  - 函数名字相同；
  - 参数不同；
  - virtual 关键字可有可无。
- 覆盖是指派生类函数覆盖基类函数，特征是：（函数名称，参数一模一样才叫覆盖）
  - 不同的范围（分别位于派生类与基类）；
  - 函数名字相同；
  - 参数相同；
  - 基类函数必须有 virtual 关键字。
- “隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：
  - 如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。
  - 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）

### 64. 链表各自有序，请把它们合并成一个链表依然有序。

- 要善于运用伪头节点的做法

```

1  struct Node{
2      Node* next;
3      int val;
4  };
5
6  Node* merge(Node* head1, Node* head2){
7      if(head1 == NULL){
8          return head2;
9      }
10     if(head2 == NULL){
11         return head1;

```

```

12     }
13     Node* pHead = new Node; // 伪头节点
14     Node* pSave = pHead; // 保留首节点
15     while (head1 && head2)
16     {
17         if(head1->val > head2->val){
18             pHead->next = head2;
19             head2 = head2->next;
20             pHead = pHead->next;
21         }
22         else{
23             pHead->next = head1;
24             head1->next = head1->next;
25             pHead = pHead->next;
26         }
27     }
28     Node* p = pSave->next;
29     delete pSave;
30     return p;
31 }
32
33 int main()
34 {
35     Node* h1 = new Node;
36     Node* h2 = new Node;
37     h1->val = 3;
38     h2->val = 2;
39     h1->next = NULL;
40     h2->next = NULL;
41     Node* p = merge(h1, h2);
42     while (p)
43     {
44         cout << p->val << " ";
45         p = p->next;
46     }
47     cout << endl;
48     int a = 5;
49     cout << a << endl;
50
51     return 0;
52 }

```

## 65. 程序运行结果

- ```

1  class B
2  {
3  public:
4      B()
5      {
6          cout<<"default constructor"<<endl;
7      }
8      ~B()
9      {
10         cout<<"destructed"<<endl;
11     }
12     B(int i):data(i)
13     {

```

```

14         cout<<"constructed by parameter" << data <<endl;
15     }
16 private:
17     int data;
18 };
19 B Play( B b)
20 {
21     return b ;
22 }
23
24
25 int main()
26 {
27     B b(5); // 构造函数
28     B temp = Play(b); // 执行了两次复制构造函数
29     return 0;
30 }
31
32 /*
33 constructed by parameter5
34 destructed
35 destructed
36 destructed
37 */

```

- 

```

1 class B
2 {
3 public:
4     B()
5     {
6         cout<<"default constructor"<<endl;
7     }
8     ~B()
9     {
10        cout<<"destructed"<<endl;
11    }
12    B(int i):data(i)
13    {
14        cout<<"constructed by parameter" << data <<endl;
15    }
16 private:
17     int data;
18 };
19 B Play( B b)
20 {
21     return b ;
22 }
23
24
25 int main()
26 {
27     B temp = Play(5); // 执行了一次构造函数， 一次复制构造函数
28     return 0;
29 }
30
31 /*
32 constructed by parameter5

```

```
33     destructed
34     destructed
35     */
```

66. 将“引用”作为函数参数有哪些特点？

- 传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对 其相应的目标对象（在主调函数中）的操作。
- 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用复制构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。
- 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用“\*指针变量名”的形式进行运算，这很容易产生 错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰

```
1  class A{
2  public:
3      A(){
4          cout << "A constructed" << endl;
5      }
6      A(A&){
7          cout << "A copy constructed" << endl;
8      }
9  };
10
11 void f1(A a){
12     cout << "f1" << endl;
13     return;
14 }
15
16 void f2(A& a){
17     cout << "f2" << endl;
18     return;
19 }
20
21 void f3(A* a){
22     cout << "f3" << endl;
23     return;
24 }
25
26 int main()
27 {
28     A a;
29     f1(a);
30     f2(a);
31     f3(&a);
32     return 0;
33 }
34
35 /*
36 A constructed
37 A copy constructed
38 f1
39 f2
```

```
40 f3
41 */
```

67. 复制构造函数为什么参数必须是引用？

- 如果复制构造函数中的参数不是一个引用，即形如 `CClass(const CClass c class)`，那么就相当于采用了传值的方式 (pass-by-value)，而传值的方式会调用该类的复制构造函数，从而造成无穷递归地调用复制构造函数。因此复制构造函数的参数必须是一个引用。其实之前也说过，C++ 中引用能做的事儿，指针都能做，这里必须传引用而不用指针，可以看作是 C++ 中的一种习惯。

68. 赋值操作符 (=) 的返回值、赋值操作符的参数为什么都是引用。

- 赋值操作符 (=) 的返回值必须是引用，只是为了满足连续赋值 (`a = b = c`)，只有引用可以作为左值，而返回对象的话是不能满足连续赋值的（因为返回的对象在返回以后会被析构，无法找到地址，自然无法被赋值）
- 赋值操作符的是引用，这样可以减少一次复制构造函数的调用。

```
1  class A{
2  private:
3      int val;
4  public:
5      A(){
6          val = 2;
7          cout << "A constructed" << endl;
8      }
9      A(A&){
10         cout << "A copy constructed" << endl;
11     }
12     A& operator=(A& a){
13         cout << "=" << endl;
14         val = a.val;
15         return *this;
16     }
17 };
18
19
20 int main()
21 {
22     A a1;
23     A a2 = a1; // 初始化时调用复制构造函数
24     a2 = a1; // 一个对象复制给另一个已经存在的对象时调用=的重载函数
25     return 0;
26 }
27
28 /*
29 A constructed
30 A copy constructed
31 =
32 */
```

69. 什么时候需要“引用”？

- 流操作符 (<<, >>) 和赋值操作符 (=) 的返回值、复制构造函数的参数、赋值操作符的参数。

70. 运算符的重载既可以是成员函数，又可以是非成员函数，成员函数的重载方式更加方便，那什么时候必须重载为非成员函数？

- 要重载的操作符的第一个操作数不是可以更改的类型，例如"<<"运算符的第一个操作数的类型为 ostream，是标准库的类型，无法向其中添加成员函数。
- 以非成员函数形式重载，支持灵活的类型转换。例如下面的代码中，可以直接使用 5.0+c1，因为 Complex 的构造函数使得实数可以被隐含转换为 Complex 类型。这样 5.0+c1 就会以 operator+ (ComplexC 5.0, c1) 的方式来执行，c1+5.0 也一样，从而支持了实数和复数的相加，很方便也很直观；而以成员函数重载时，左操作数必须具有 Complex 类型，不能是实数(因为调用成员函数的目的对象不会被隐含转换)，只有右操作数可以是实数(因为右操作数是函数的参数，可以隐含转换)。

```

1  class Complex { // 复数类定义
2  public: // 外部接口
3      Complex (double r= 0.0 , double i=0.0) : real(r),imag(i) {} // 构造函数
4      friend Complex operator+ (const Complex &c1 , const Complex &c2); // 运
      算符+重载
5      friend Complex operator- (const Complex &c1 , const Complex &c2); // 运
      算符-重载
6      friend ostream & operator<< (ostream &out, const Complex &c) ; // 运算符
      <<重载
7
8  private: // 私有数据成员
9      double real; // 复数实部
10     double imag; // 复数虚部
11 };
12
13 Complex operator+ (const Complex &c1 , const Complex &c2){ // 重载运算符函数实
      现
14     return Complex (c1.real + c2.real, c1.imag+ c2.imag);
15 }
16
17 Complex operator- (const Complex &c1 , const Complex &c2){ // 重载运算符函数实
      现
18     return Complex (c1.real- c2.real, c1.imag- c2.imag);
19 }
20
21 ostream & operator<< (ostream &out , const Complex &c){ // 重载运算符函数实现
22     out<<"("<<c.real<< " , "<< c.imag<< ")";
23     return out;
24 }
25 int main () {
26     Complex c1(5, 4) , c2(2 , 10) , c3;
27     cout<< "c1= "<< c1<< endl;
28     cout<< "c2= "<< c2<< endl;
29     c3=c1-c2;
30     cout<< "c3= c1- c2= "<< c3<< endl;
31     c3=c1+c2;
32     cout<< "c3= c1 + c2= "<< c3<< endl;
33     return 0;
34 }
35
36 /*
37 c1= (5, 4)
38 c2= (2, 10)
39 c3= c1- c2= (3, -6)
40 c3= c1 + c2= (7, 14)
41 */

```

71. 哪些时候必须重载为成员函数？

语法规则"=", "[]", "()", "->"只能被重载为成员函数，而且派生类中的"="运算符函数总会隐藏基类中的"="运算符函数。

72. 面向对象的三个基本特征，并简单叙述之？

- 封装：将客观事物抽象成类，每个类对自身的数据和方法实行 protection(private, protected, public)
- 继承：广义的继承有三种实现形式：实现继承（指使用基类的属性和方法而无需额外编码的能力）、可视继承（子窗体使用父窗体的外观和实现代码）、接口继承（仅使用属性和方法，实现滞后到子类实现）。前两种（类继承）和后一种（对象组合=>接口继承以及纯虚函数）构成了功能复用的两种方式。
- 多态：是将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象 可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许 将子类类型的指针赋值给父类类型的指针。

73. 一个单向链表，不知道头节点，一个指针指向其中的一个节点，问如何删除这个指针指向的节点？

- 将这个指针指向的 next 节点值 copy 到本节点，将 next 指向 next->next,并随后删除原 next 指向的节点。

74. 字符串 A 和 B,输出 A 和 B 中的最大公共子串

```
1 char* commonStr(char* shortS, char* longS){
2     if(strstr(longS, shortS) != NULL){
3         return shortS;
4     }
5
6     char* subStr = malloc(sizeof(char) * 256);
7
8     // 因为要找到最大公共子字符串，所以i的长度应该从最大的开始。
9     // for(int i = 1; i < strlen(shortS); i++){ // 在shortS中找到长度为i的，看
    是否是子串。
10    for(int i = strlen(shortS); i > 0; i--){ // 在shortS中找到长度为i的，看是否
    是子串。
11        for(int j = 0; j <= strlen(shortS)-i; j++){
12            memcpy(subStr, &(shortS[j]), i);
13            subStr[i] = '\0';
14            if(strstr(longS, subStr) != NULL){
15                return subStr;
16            }
17        }
18    }
19    return NULL;
20 }
21
22 int main(){
23     char s1[] = "cdaocdfecdghj";
24     char s2[] = "pmcdfa";
25
26     char* res;
27     if(strlen(s1) > strlen(s2)){
28         res = commonStr(s2, s1);
29     }
30     else{
31         res = commonStr(s1, s2);
32     }
33     printf("%s\n", res);
34     return 0;
```



```

35 }
36
37 /*
38 cdf
39 */

```

#### 74. C++中的左值与右值的区别？

- 不过左值、右值通常不是通过一个严谨的定义而为人所知的，大多数时候左右值的定义与其判别方法是一体的。一个最为典型的判别方法就是，在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”。比如：

`a = b + c;`

在这个赋值表达式中，`a`就是一个左值，而`b + c`则是一个右值。这种识别左值、右值的方法在C++中依然有效。不过C++中还有一个被广泛认同的说法，那就是可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值。那么这个加法赋值表达式中，`&a`是允许的操作，但`&(b + c)`这样的操作则不会通过编译。因此`a`是一个左值，`(b + c)`是一个右值。或者，也可以简单的认为可以被引用的是左值，不可以被引用的就是右值。当然第三种说法还有些问题，如虽然简单的引用不可以实现，但是可以改为常引用。下面是对纯右值 `a1+a2` 的引用。

```

1  int main(){
2      int a1, a2;
3      a1 = a2 = 1;
4      const int& a3 = a1+a2;
5      cout << a3 << endl;
6      return 0;
7  }
8
9  /*
10  2
11  */

```

- 右值是由两个概念构成的，一个是将亡值（xvalue，eXpiring Value），另一个则是纯右值（prvalue，Pure Rvalue）。

其中纯右值就是C++98标准中右值的概念，讲的是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值（我们在前面多次提到了）就是一个纯右值。一些运算表达式，比如`1 + 3`产生的临时变量值，也是纯右值。而不跟对象关联的字面量值，比如：`2`、`'c'`、`true`，也是纯右值。

- 如在写程序时，函数参数有时会是引用以提高程序效率，这个时候就涉及到程序的引用。

```

1  void f1(string str){
2      cout << str << endl;
3  }
4
5  void f2(string& str){
6      cout << str << endl;
7  }
8
9  void f3(const string& str){
10     cout << str << endl;
11 }
12
13 int main()
14 {

```

```

15     string str = "hello";
16     f1("hello");
17     // f2("hello"); // 会出错 右值（常量）不可以通过引用绑定到左值（可以被赋值的对象/
    可以被引用的对象）
18     f2(str);
19     f3("hello");
20     f3(str); // 左值可以绑定为右值
21     return 0;
22 }
23
24 /*
25 hello
26 hello
27 hello
28 hello
29 */

```

## 75. 实现 strcmp

```

1  int strcmp(const char* s1, const char* s2){
2      while (*s1 == *s2++)
3      {
4          if(*s1++ == 0){ // s1一定会自增
5              return 0;
6          }
7      }
8      return *s1 - *(--s2);
9  }
10 }
11
12 int main()
13 {
14     cout << strcmp("a", "ab") << endl;
15     cout << strcmp("ef", "aa") << endl;
16     cout << strcmp("ef", "ef") << endl;
17     return 0;
18 }
19
20 /*
21 -98
22 4
23 0
24 */

```

## 76. sizeof的使用

```

1  void f(char s[100]){
2      cout << sizeof(s) << endl;
3  }
4
5  int main()
6  {
7      char str[] = "123456789";
8      char* p = (char*)malloc(sizeof(char) * 100);
9      int (*arr)[10][15] = (int (*)[10][15])malloc(sizeof(int)*3*10*15);
10     cout << sizeof(str) << endl;

```

```

11     f(str); // 'sizeof' on array function parameter 's' will return size of
        'char*'
12     cout << sizeof(p) << endl;
13     cout << sizeof(arr) << endl;
14     return 0;
15 }
16
17 /*
18 10
19 8
20 8
21 8
22 */

```

77. sizeof有时结果不同

- 在C++中

```

1  int main()
2  {
3      char s1[3] = "a";
4      int a = 5;
5      char ch = 'a';
6      printf("%d %d %d %d\n", sizeof(s1), sizeof(ch), sizeof('b'), sizeof(a));
7      return 0;
8  }
9
10 /*
11 3 1 1 4
12 */

```

- 在c中

```

1  int main()
2  {
3      char s1[3] = "a";
4      int a = 5;
5      char ch = 'a';
6      printf("%d %d %d %d\n", sizeof(s1), sizeof(ch), sizeof('b'), sizeof(a));
7      return 0;
8  }
9  /**
10 3 1 4 4
11 */

```

可能是在c中，会根据情况把字符转为对应的int。

78. 实现strstr函数

```

1  char* strstr(char* str, char* subStr){
2      int val;
3      if(strlen(subStr) == 0){ // subStr = "" (此时sizeof(subStr) = 8)
4          return str;
5      }
6      for(int i = 0; i < strlen(str); i++){
7          val = i;

```

```

8         int j = 0;
9         while (str[i++] == subStr[j++])
10        {
11            if(subStr[j] == '\0'){
12                return &str[val];
13            }
14        }
15        i = val;
16    }
17    return NULL;
18 }
19 int main(){
20     char s1[3] = "ab";
21     char s2[3] = "ef";
22     char* ans1 = strstr("abcd", "bc");
23     char* ans2 = strstr("abcd", "bc");
24     if(ans1 != NULL){
25         printf("%s\n", ans1);
26     }
27     if(ans2 != NULL){
28         printf("%s\n", ans2);
29     }
30     return 0;
31 }
32
33 /**
34 bcd
35 bcd
36 */

```

79. 为什么 `strstr()` 在字符串 `subStr=""` 时会返回 `str` 的首地址，而不是说返回为空。

这个问题是一个很好的问题，背后数学集合概念的表现，我们把 `str` 和 `subStr` 都看作集合，那么此时 `subStr` 就是空集，在离散中，可以证明空集包含于任何集合，所以 `str` 是包含 `subStr` 的，因此不返回为空，而是返回首地址。

80. 实现 `strcat` 函数

```

1 char* strcat(char* des, char* src){
2     char* d;
3     if(des==NULL || src==NULL){
4         return des;
5     }
6     d = des;
7     while (*d) // '\0' 是和0一样的
8     {
9         d++;
10    }
11    while (*src)
12    {
13        *d = *src++;
14        d++;
15    }
16    *d = '\0';
17    return des;
18 }
19
20 int main()

```

```

21 {
22     char s1[50] = {0};
23     strcat(s1, "hello");
24     strcat(s1, " H");
25     printf("%s\n", s1);
26     return 0;
27 }
28 /**
29 hello H
30 */

```

#### 80. 函数模板与类模板有什么区别？

- 函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。

```

1  template<class T>
2  T Fabs(T v){
3      return v < 0 ? -v : v;
4  }
5  int main()
6  {
7      int v1 = Fabs<int>(2); // 显式指定
8      double v2 = Fabs(-2.5); // 隐式指定
9      cout << v1 << " " << v2 << endl;
10
11     return 0;
12 }
13
14 /**
15 2 2.5
16 */

```

#### 81. 字符串常量与字符数组的存储形式不同

- 字符数组，每个都有其自己的存储区，它们的值则是各存储区首地址，不等。arr5和arr6并非数组而是字符指针，并不分配存储区，其后的“abc”以常量形式存于静态数据区，而它们自己仅是指向该区首地址的指针，相等。

```

1  int main()
2  {
3      char arr1[] = "ab";
4      char arr2[] = "ab";
5      char* arr5 = "ab"; // 字符串常量
6      char* arr6 = "ab"; // 字符串常量
7      printf("%d\n", arr1==arr2);
8      printf("%d\n", arr5==arr6);
9      printf("%d\n", arr1==arr5);
10     printf("%ld %ld", arr5, arr6);
11
12     return 0;
13 }
14 /**
15 0
16 1
17 0
18 4210688 4210688

```

82. 非C++内建型别 A 和 B，在哪几种情况下 B 能隐式转化为 A？

- `class B : public A { ..... }` // B 公有继承自 A，可以是间接继承的
- `class B { operator A( ); }` // B 实现了隐式转化为 A 的转化
- `class A { A( const B& ); }` // A 实现了 non-explicit 的参数为 B（可以有其他带默认值的参数）构造函数
- `A& operator= ( const A& );` // 赋值操作，虽不是正宗的隐式类型转换，但也可以勉强算一个

83. 按默认构造函数定义对象，不需要加括号

```

1  class A{
2  public:
3      A(int){}
4      A(){}
5      void f(){}
6  };
7
8  int main()
9  {
10     A a(1);
11     a.f();
12     A* b = new A();
13     b->f();
14
15     // A c(); // 按默认构造函数定义对象，不需要加括号
16     // c.f();
17     return 0;
18 }
```

84. 程序运行结果

```

1  struct A{
2      int val;
3      A(){
4          val = 4;
5          A(5); // 只会产生一个临时对象
6      }
7      A(int v){
8          val = v;
9      }
10
11 };
12
13 int main()
14 {
15     A a;
16     cout << a.val << endl;
17     return 0;
18 }
19
20 /*
21 4
22 */
```

- 在默认构造函数内部再调用带参的构造函数属用户行为而非编译器行为，亦即仅执行函数调用，而不会执行其后的初始化表达式。只有在生成对象时，初始化表达式才会随相应的构造函数一起调用。

85. [C++中 \(int&\) 和 \(int\) 的区别。](#)

```
1  int main()
2  {
3      float a = 1.0f;
4      cout << (int)a << endl; // 强制类型转换
5      cout << (int &)a << endl; // 同一个数据，不同的解释
6
7      float b = 0.0f;
8      cout << (int)b << endl;
9      cout << (int &)b << endl;
10     return 0;
11 }
12
13 /*
14 1
15 1065353216
16 0
17 0
18 */
```

86. C++中结构和类对象指针都必须使用 new 创建

```
1  class A{
2  private:
3      int val;
4      string str;
5  public:
6      A(): val(5), str("hello"){ }
7      A(int v): val(v), str("hell"){ }
8      void show(){
9          cout << "val=" << val << " str=" << str << endl;
10     }
11 };
12
13 int main()
14 {
15     A* a = new A;
16     a->show();
17     A* b = (A*) malloc(sizeof(A));
18     b->show();
19     A* c = new A(3);
20     c->show();
21     return 0;
22 }
23
24 /*
25 val=5 str=hello
26 val=15622960 str=
27 val=3 str=hell
28 */
```

new出来对象会调用对象的构造函数，但是malloc出来的对象是没有调用构造函数的。

#### 87. 强制类型转换

```
1  int main()
2  {
3      unsigned int a = 3;
4      cout << a * -1 << endl; // 最高位是1的unsigned
5      cout << int(a * -1) << endl;
6      return 0;
7  }
8
9  /*
10 4294967293
11 -3
12 */
```

#### 88. 局部变量覆盖全局变量

```
1  int a;
2  int b;
3  int c;
4  void F1()
5  {
6      b=a*2;
7      a=b;
8  }
9  void F2()
10 {
11     c=a+1;
12     a=c;
13 }
14
15 int main()
16 {
17     a = 5;
18     cout << a << " " << b << " " << c << endl;
19     F1();
20     F2();
21     printf("%d\n", a);
22     return 0;
23 }
24
25 /*
26 5 0 0
27 11
28 */
```

#### 89. 只有fun1()是虚函数

```
1  class A
2  {
3  public:
4      // virtual void func1(); // 虚函数必须实现，如果不实现，编译器将报错
5      virtual void func1(){
6          cout << "fun1 in class A" << endl;
```



```

7     }
8     void func2(){cout << "fun2 in class A" << endl;} // 普通函数可以不实现
9 };
10
11
12 class B: public A
13 {
14 public:
15     void func1(){cout << "fun1 in class B" << endl;}
16     virtual void func2(){cout << "fun2 in class B" << endl;}
17 };
18
19 int main()
20 {
21     A* p = new B();
22     p->func1();
23     p->func2(); // 如果A中未实现func2,这里调用会报错
24     return 0;
25 }
26
27 /*
28 fun1 in class B
29 fun2 in class A
30 */

```

#### 86. 静态成员函数不能被声明为虚函数

```

1 class A
2 {
3 private:
4     static int a; // 这里只是声明，必须要在文件作用域定义，已分配空间
5     int* pA;
6 public:
7     A(){
8         pA = new int;
9         *pA = 5;
10        cout << "A构造" << endl;
11    }
12    // virtual static void showStatic(){
13    // error: member 'showStatic' cannot be declared both 'virtual' and
14    'static'
15    static void showStatic(){
16        cout << "a=" << a << endl;
17    }
18    virtual void show(){
19        cout << "pA=" << *pA << endl;
20        cout << "a=" << a << endl;
21    }
22    }
23    virtual ~A(){ // 一定要是虚析构函数，否则基类指针不会调用派生类的析构函数
24        cout << "A 析构" << endl;
25        delete pA;
26    }
27 };
28
29 int A::a = 1;

```

```

30
31
32 class B: public A
33 {
34 private:
35     static int b;
36     int *pB;
37 public:
38     B(){
39         pB = new int;
40         *pB = 6;
41         cout << "B构造" << endl;
42     }
43     static void showStatic(){
44         cout << "b=" << b << endl;
45     }
46     void show(){
47         cout << "pB=" << *pB << "  b=" << b << endl;
48     }
49     ~B(){
50         cout << "B 析构" << endl;
51         delete pB;
52     }
53 };
54
55 int B::b = 2;
56
57 int main()
58 {
59     A* p = new B;
60     p->show();
61     p->showStatic();
62     delete p; // 这里一定要delete p否则即便程序运行完毕，也不会调用析构函数
63     return 0;
64 }
65
66 /*
67 A构造
68 B构造
69 pB=6  b=2
70 a=1
71 B 析构
72 A 析构
73 */

```

## 87. sizeof 应用在结构上的情况

```

1 struct strc{
2     double a;
3     char cha;
4     int b;
5 };
6
7 int main()
8 {
9     cout << sizeof(strc) << endl;
10    return 0;

```

```

11 }
12
13 /*
14 16
15 */

```

- 原理

| 类型     | 对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量） |
|--------|-------------------------------|
| char   | 偏移量必须为 sizeof(char)即 1 的倍数    |
| int    | 偏移量必须为 sizeof(int)即 4 的倍数     |
| float  | 偏移量必须为 sizeof(float)即 4 的倍数   |
| double | 偏移量必须为 sizeof(double)即 8 的倍数  |

- 为了提高 CPU 的存储速度，VC 对一些变量 的起始地址做了"对齐"处理。在默认情况下，VC 规定各成员变量存放的起始地址相对于结 构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数。确保结构的大小为结构的字节边界数（即该结构中占用最大空间的类型所占用的字节数）的倍数。

88. sscanf和scanf一样，都需要给出存放地址

```

1 void DoSomething(char* p)
2 {
3     char str[16];
4     int n;
5     assert(NULL != p);
6     sscanf(p, "%s%d", str, &n); // 而不是sscanf(p, "%s%d", str, n);
7 }

```

89. strcpy导致数组越界问题

```

1 int main()
2 {
3     char arr1[10], arr2[10];
4     for(int i = 0; i < 10; i++){
5         arr1[i] = i+'a';
6     }
7     printf("%s\n", arr1);
8     strcpy(arr2, arr1); // arr2内容未知
9     printf("%s\n", arr2);
10    return 0;
11 }
12
13 /*
14 abcdefghij
15
16 abcdefghij
17
18 */

```

90. 静态变量上一次的值会被保留。

```

1  int add_n(int n)
2  {
3      static int i=100;
4      i+=n;
5      return i;
6  }
7
8  int main()
9  {
10     int a = add_n(1);
11     int b = add_n(1);
12     cout << a << " " << b << endl;
13     return 0;
14 }
15
16 /*
17 101 102
18 */

```

91. 当声明静态数组（不是运行时使用new或者malloc申请的动态数组）时，在C++中，可以向下面这样用

```

1  int main(){
2      const int size1 = 10;
3      int size2 = 10;
4
5      int arr1[size1]; // 完全可以
6      int arr2[size2]; // 会提示出错，但可以运行
7      arr2[5] = 9;
8      cout << arr2[5] << endl;
9
10     return 0;
11 }
12
13 /*
14 9
15 */

```

在c中，情况有些不同，但结果没有影响

```

1  int main()
2  {
3      const int size1 = 10;
4      int size2 = 10;
5
6      int arr1[size1]; // 会提示出错，但可以运行
7      int arr2[size2]; // 会提示出错，但可以运行
8      arr1[5] = 8;
9      arr2[5] = 9;
10     printf("%d %d", arr1[5], arr2[5]);
11
12     return 0;
13 }
14
15 /**
16 8 9

```

在c中声明静态数组，[它的大小在编译的时候已经确定，且大小在整个数组生命期内不会改变](#)，通过查阅可知c中声明静态数组的方法有以下几种：

```
1  int n[10]; // integer constants are constant expressions
2  char o[sizeof(double)]; // sizeof is a constant expression
3  enum { MAX_SZ=100 };
4  int n[MAX_SZ]; // enum constants are constant expressions
5  #define MAX_SZ 100 // 使用宏定义
```

[Array declaration.Can a const variable be used to declare the size of an array in C?](#)

## 92. [内联函数是怎么提高效率的](#)

上面博客中介绍的内联函数的工作原理也就告诉了我们为什么只有短小的函数适合作为内联函数，为什么直接递归调用函数是不可以作为真正意义上的内联函数的（因为内联函数是在编译阶段在调用处直接把内联函数复制过来的，这样就不用跳转以及函数调用是的入栈出栈了，但是递归函数在编译阶段，编译器并不知道这个函数会被实际调用多少次），也就能理解为什么下面的程序依然可以出结果了（因为我们只是声明该函数为内联函数，具体该函数是不是内联函数，与我们声明或者不声明没有多大关系，一个函数最终是否能成为内联函数，由编译器决定）。

```
1  inline void hanluota(char a, char b, char c, int n){
2      if(n==1){
3          cout << a << "->" << c << endl;
4      }
5      else{
6          hanluota(a, c, b, n-1);
7          hanluota(a, b, c, 1);
8          hanluota(b, a, c, n-1);
9      }
10 }
11 int main(){
12     hanluota('A', 'B', 'C', 3);
13     return 0;
14 }
15
16 /*
17 A->C
18 A->B
19 C->B
20 A->C
21 B->A
22 B->C
23 A->C
24 */
```

## 93. C++中const修饰函数形参列表中的形式参数。

在C++的类中，很多普通成员函数的形参列表中如果参数是一个定义类对象，基本上都会被const修饰（const修饰了之后可以作为右值的引用），而且传的值都是引用尤其是在运算符重载那里。具体来说，const可以放在三个位置上，[运算符重载关于const的分析（超详细）](#)。如 `const Point operator+(const Point &point) const { return Point(m_x + point.m_x, m_y + point.m_y); }`，其中Point是我们自己定义的一个类。

- 第1个const，表示**返回值是常量**

- 第2个 const，即括号里的 `const Point &point`，是为了扩大接受参数的范围。
- 第3个 const，即括号后面的 `const`，它的作用是使得该函数可以被 `const` 对象所调用。

先看下面示例1：

实例1：

```
1  class A{
2  private:
3      int data;
4  public:
5      A(int val=1){
6          data = val;
7          cout << "调用构造函数" << endl;
8      }
9      A(const A& a){ // 用const修饰
10         data = a.data;
11         cout << "调用复制构造函数" << endl;
12     }
13     A test(A a){
14         // return A(2);
15         return A(a);
16     }
17     void show(){
18         cout << "data=" << data << endl;
19     }
20 };
21
22 int main(){
23     A a1, a2;
24     A a3 = a2.test(a1);
25     a3.show();
26     return 0;
27 }
28
29 /*
30 调用构造函数
31 调用构造函数
32 调用复制构造函数
33 调用复制构造函数
34 data=1
35 */
```

示例1的复制构造函数加了const修饰，是没有问题的。

示例2：

```
1  class A{
2  private:
3      int data;
4  public:
5      A(int val=1){
6          data = val;
7          cout << "调用构造函数" << endl;
8      }
9      A(A& a){ //
10         data = a.data;
11         cout << "调用复制构造函数" << endl;
```

```

12     }
13     A test(A a){
14         return A(a);
15     }
16     void show(){
17         cout << "data=" << data << endl;
18     }
19 };
20
21
22 int main(){
23     A a1, a2;
24     A a3 = a2.test(a1);
25     a3.show();
26     return 0;
27 }
28
29 /*
30 error: cannot bind non-const lvalue reference of type 'A&' to an rvalue
of type 'A'
31 return A(a);
32 */

```

#### 出错原因

- [C++类成员函数返回类的对象](#)中提到，“必须使用复制构造函数 A(const A &p)，使返回的临时对象转为持久对象 s，从而具有正常功能”。
- [拷贝构造函数中的参数是引用，因为不是引用就会无限制的调用拷贝构造函数，所以编译器强制报错提醒我们必须使用引用，引用传递是地址传递，和指针一样还发现一个事实，当拷贝构造函数参数没有const时无法通过临时对象创建新对象。](#)

#### 实例3——重载赋值运算符函数没有const修饰

```

1 class Complex{
2 private:
3     double real, imag;
4 public:
5     Complex(double r=1, double i=1): real(r), imag(i){
6         cout << "构造函数调用完成" << endl;
7     };
8     Complex(const Complex& c){
9         real = c.real;
10        imag = c.imag;
11        cout << "复制构造函数调用完成" << endl;
12    }
13    Complex operator+(const Complex& c){
14        // Complex tmp;
15        // tmp.real = real + c.real;
16        // tmp.imag = imag + c.imag;
17        // return tmp;
18
19        //上面的写法等价于
20        return Complex(real+c.real, imag+c.imag);
21    }
22    Complex& operator=(Complex& c){
23        real = c.real;
24        imag = c.imag;
25        cout << "赋值构造函数调用完成" << endl;

```

```

26         return *this;
27     }
28     void show(){
29         cout << "real=" << real << " imag=" << imag << endl;
30     }
31     friend ostream& operator<<(ostream& out, const Complex& c);
32 };
33
34 ostream& operator<<(ostream& out, const Complex& c){
35     out << "real=" << c.real << " imag=" << c.imag << endl;
36     return out;
37 }
38
39 int main()
40 {
41     Complex c1(2, 3), c2, c3;
42     cout << (c1+c2);
43     c3 = c1 + c2;
44     c3.show();
45     return 0;
46 }
47
48 /*
49 error: cannot bind non-const lvalue reference of type 'Complex&' to an
rvalue of type 'Complex'
50     c3 = c1 + c2;
51 */

```

解释: `c1 + c2` 为右值, 要想对他引用必须加`const`修饰。

实例4——复制构造函数没有`const`修饰。

```

1  class Complex{
2  private:
3      double real, imag;
4  public:
5      Complex(double r=1, double i=1): real(r), imag(i){
6          cout << "构造函数调用完成" << endl;
7      };
8      Complex(Complex& c){
9          real = c.real;
10         imag = c.imag;
11         cout << "复制构造函数调用完成" << endl;
12     }
13     Complex operator+(const Complex& c){
14         // Complex tmp;
15         // tmp.real = real + c.real;
16         // tmp.imag = imag + c.imag;
17         // return tmp;
18
19         //上面的写法等价于
20         return Complex(real+c.real, imag+c.imag);
21     }
22     Complex& operator=(const Complex& c){
23         real = c.real;
24         imag = c.imag;
25         cout << "赋值构造函数调用完成" << endl;

```



```

26         return *this;
27     }
28     void show(){
29         cout << "real=" << real << " imag=" << imag << endl;
30     }
31     friend ostream& operator<<(ostream& out, const Complex& c);
32 };
33
34 ostream& operator<<(ostream& out, const Complex& c){
35     out << "real=" << c.real << " imag=" << c.imag << endl;
36     return out;
37 }
38
39 int main()
40 {
41     Complex c1(2, 3), c2, c3;
42     cout << (c1+c2);
43     c3 = c1 + c2;
44     c3.show();
45     return 0;
46 }
47
48 /*
49 error: cannot bind non-const lvalue reference of type 'Complex&' to an
rvalue of type 'Complex'
50     return Complex(real+c.real, imag+c.imag);
51 */

```

如果仔细看下它的提示出错位置，`Complex(real+c.real, imag+c.imag)`，可是这一句在调用构造函数哎，之前已经提到过返回值为类对象时不在调用复制构造函数，好吧，我还是给复制构造函数加上const修饰吧。

```

1  class Complex{
2  private:
3      double real, imag;
4  public:
5      Complex(double r=1, double i=1): real(r), imag(i){
6          cout << "构造函数调用完成" << endl;
7      };
8      Complex(const Complex& c){
9          real = c.real;
10         imag = c.imag;
11         cout << "复制构造函数调用完成" << endl;
12     }
13     Complex operator+(const Complex& c){
14         // Complex tmp;
15         // tmp.real = real + c.real;
16         // tmp.imag = imag + c.imag;
17         // return tmp;
18
19         //上面的写法等价于
20         return Complex(real+c.real, imag+c.imag);
21     }
22     Complex& operator=(const Complex& c){
23         real = c.real;
24         imag = c.imag;
25         cout << "赋值构造函数调用完成" << endl;

```

```

26     return *this;
27 }
28 void show(){
29     cout << "real=" << real << " imag=" << imag << endl;
30 }
31 friend ostream& operator<<(ostream& out, const Complex& c);
32 };
33
34 ostream& operator<<(ostream& out, const Complex& c){
35     out << "real=" << c.real << " imag=" << c.imag << endl;
36     return out;
37 }
38
39 int main()
40 {
41     Complex c1(2, 3), c2, c3;
42     cout << (c1+c2);
43     c3 = c1 + c2;
44     c3.show();
45     return 0;
46 }
47
48 /*
49 构造函数调用完成
50 构造函数调用完成
51 构造函数调用完成
52 构造函数调用完成
53 real=3 imag=4
54 构造函数调用完成
55 赋值构造函数调用完成
56 real=3 imag=4
57 */

```

**结论：**能加const修饰就别偷懒。☹️☹️☹️☹️☹️☹️☹️☹️☹️☹️

#### 94. 面向对象设计之魂的六大原则。

## 95. 泛型程序设计

泛型程序设计是编写不依赖于具体数据类型的程序。c++中，模板是泛型程序设计的主要工具。

## 96. STL简介

STL (Standard Template Library)，标准模板库，提供了一些常用的数据结构和算法。如vector容器就是一种线性存储的数据结构，sort()函数就是使用快排的排序算法。

## 97. STL 的4种基本组件

容器(container)、迭代器(iterator)、函数对象(function object)、算法(algorithm)

○ 容器

可以分为两种基本类型：顺序容器 (sequence container) 和关联容器 (associative container)。顺序容器将一组具有相同类型的元素以严格的线性形式组织起来，向量、双端队列和列表容器就属于这一种。关联容器具有根据一组索引来快速提取元素的能力，集合和映射容器就属于这一种。

- vector容器
- deque双端队列
- list链表容器
- set容器
- map容器

对于stack栈容器，queue队列容器，priority\_queue优先队列，multiset容器，multimap容器等我们认为他是一种适配器，即他们的很多功能由上述提到的顺序容器和关联容器而来。

- 这里对顺序容器和关联容器做一个解释，比如，为什么认为列表是一个顺序容器呢？
- C++中，顺序容器是指，这个容器逻辑上是一个线性的容器，逻辑上容器中的a元素的下一个元素是b，具有这个概念的容器就是顺序容器。在链表中我们也有这个概念。而对于关联容器，就没有这个概念，关联容器更多的是一个hash映射的概念。
- 迭代器
  - 迭代器提供了顺序访问容器中每个元素的方法。对迭代器可以使用"++"运算符来获得指向下一个元素的迭代器，可以使用"\*"运算符访问一个迭代器所指向的元素。如果元素类型是类或结构体，还可以使用"-->"运算符直接访问该元素的一个成员，有些迭代器还支持通过"--"运算符获得指向上一个元素的迭代器。指针也具有同样的特性，因此指针本身就是一种迭代器，迭代器是泛化的指针。
- 函数对象
  - 函数对象是一个行为类似函数的对象，对它可以像调用函数一样调用。任何普通的函数和任何重载了"()"运算符的类的对象都可以作为函数对象使用，函数对象是泛化的函数。

## • 算法

- 1 包括查找算法、排序算法、消除算法、计数算法、比较算法、变换算法、置换算法和容器管理等。这些算法的一个最重要的特性就是它们的统一性，并且可以广泛用于不同的对象和内置的数据类型。

```
1  #include <iostream>
2  #include <vector>
3  #include <iterator>
4  #include <queue>
5  #include <algorithm>
6  #include <functional>
7
8  using namespace std;
9
10 int main(){
11     const int N = 5;
12     vector<int> s(N);
13     for(int i = 0; i < N; i++){
14         s[i] = rand()%10 + 1;
15     }
16     for(int val: s){
17         cout << val << " ";
18     }
19     cout << endl;
20
21     // transform          算法      <algorithm>
22     // negate<int>()       函数对象  <functional>
23     // ostream_iterator<int>(cout, " ") 迭代器  <iterator>
24     // s.begin(), s.end()   迭代器
25     transform(s.begin(), s.end(), ostream_iterator<int>(cout, " "),
26 negate<int>());
27     cout << endl;
28     for(int val: s){
29         cout << val << " ";
30     }
31     cout << endl;
32     return 0;
```

```

32     }
33
34     /*
35     2 8 5 1 10
36     -2 -8 -5 -1 -10
37     2 8 5 1 10
38     */
39

```

注：对于普通函数对象来说，传递时不需要()，对于类的对象来说，传递时需要()。

```

1     bool cmp(int& a, int& b){ // 注意这里的写法，引用
2         return a > b;
3     }
4
5     int main()
6     {
7         const int N = 5;
8         vector<int> s(N);
9         for(int i = 0; i < N; i++){
10             s[i] = rand()%10 + 1;
11         }
12         for(int val: s){
13             cout << val << " ";
14         }
15         cout << endl;
16         sort(s.begin(), s.end(), cmp);
17
18         for(int val: s){
19             cout << val << " ";
20         }
21         cout << endl;
22         return 0;
23     }
24
25     /*
26     2 8 5 1 10
27     10 8 5 2 1
28     */

```

- 算法通过迭代器而非直接通过容器来操作数据，函数对象用来描述算法对单个数据执行的具体运算，迭代器充当算法和容器的桥梁。

## 98. vector容器（向量容器）

向量容器是一种支持高效的随机访问和高效向尾部加入新元素的容器。vector容器的实质是一个动态分布的数组。

- vector容器的空间问题：

vector容器在初始化的时候，会申请一定大小的空间内存(capacity)。当添加元素空间占满的时候，会重新申请一个更大的空间，将vector容器的原来元素——复制到新的数组中，并且添加新的元素，之后释放原来的内存空间。

```

1  int main(){
2      vector<int> v(5, 2);
3      v.push_back(3);
4      cout << v.capacity() << " " << v.size() << endl;
5      return 0;
6  }
7
8  /*
9  10 6
10 */

```

- vector容器内存空间的释放问题

```

1  int main(){
2      vector<int> v(5, 2);
3      v.push_back(3);
4      cout << v.capacity() << " " << v.size() << endl;
5      while (!v.empty())
6      {
7          v.pop_back();
8      }
9      cout << v.capacity() << " " << v.size() << endl;
10
11     // swap函数会让临时对象vector<int>()和v对象交换空间
12     // 这时候v对象的空间就会是空的，而临时对象只是在这一句表达式中具有一定的内存空间
13     // 表达式结束之后，临时对象所有的内存空间自动销毁
14     vector<int>().swap(v);
15     cout << v.capacity() << " " << v.size() << endl;
16
17     return 0;
18 }
19
20 /*
21 10 6
22 10 0
23 0 0
24 */

```

- swap () 的工作原理：

swap () 函数只是两个要交换对象的内存首地址，size，capacity进行了交换。

#### 99. deque双端队列

双端队列是一种支持向两端高效地插入数据、支持随机访问的容器。双端队列的底层实现可以看成是这样，有多个存储元素的数组，还要维护一个存放这些数组首地址的数组。

- 两端高效插入，删除元素

不需要对数组元素进行任何移动

- 此向中间插入元素的效率较低

需要将插入点到某一端之间的所有元素向容器的这一端移动

#### 100. list链表容器

列表是一种不能随机访问但可以高效地在任意位置插入和删除元素的容器。

#### 101. priority\_queue优先队列

```

1  int main(){
2      // priority_queue<int, vector<int>, less<int>> bigHeap;
3      priority_queue<int> bigHeap; // 默认就是大顶堆
4      priority_queue<int, vector<int>, greater<int>> smallHeap; // 小顶堆
5      int arr[10];
6      srand(time(NULL));
7      for(int i = 0; i < 10; i++){
8          arr[i] = rand()%10 + 5;
9      }
10     for(int i = 0; i < 10; i++){
11         cout << arr[i] << " ";
12     }
13     for(int i = 0; i < 10; i++){
14         bigHeap.push(arr[i]);
15         smallHeap.push(arr[i]);
16     }
17
18     cout << endl << endl << "大顶堆" << endl;
19     while (!bigHeap.empty())
20     {
21         cout << bigHeap.top() << " ";
22         bigHeap.pop();
23     }
24
25     cout << endl << "小顶堆" << endl;
26     while (!smallHeap.empty())
27     {
28         cout << smallHeap.top() << " ";
29         smallHeap.pop();
30     }
31     cout << endl;
32
33     return 0;
34 }
35
36 /*
37 11 10 9 13 7 7 6 12 14 5
38
39 大顶堆
40 14 13 12 11 10 9 7 7 6 5
41 小顶堆
42 5 6 7 7 9 10 11 12 13 14
43 */

```

## 102. set容器

set, 就是用来描述一个集合的, 满足数学中的集合的三个特征 (无序性、互异性、确定性)

```

1  int main(){
2
3      int arr[10];
4      srand(time(NULL));
5      for(int i = 0; i < 10; i++){
6          arr[i] = rand()%10 + 5;
7      }
8      for(int i = 0; i < 10; i++){

```

```

9      cout << arr[i] << " ";
10     }
11     cout << endl;
12
13     // set<int> st(10, 1); // wrong
14     set<int> st(arr, arr+10);
15     if(st.count(10)){
16         cout << "已经有这个元素" << endl;
17     }
18     else{
19         st.insert(10);
20     }
21     return 0;
22 }
23
24 /*
25 7 12 5 14 9 10 9 10 12 13
26 已经有这个元素
27 */

```

### 103. map容器

在集合中按照键查找一个元素时，一般只是用来确定这个元素是否存在，而在映射中按照键查找一个元素时，除了能确定它的存在性外，还可以得到相应的附加数据。因此，映射的一种通常用法是，根据键来查找附加数据。映射很像是一个“字典”。

```

1  int main(){
2      map<int, string> mp;
3      mp[5] = "元素5";
4      if(mp[8] == "112"){
5          cout << "这样不对" << endl;
6      }
7      if(mp.count(8)){
8          cout << "上一步已经插入" << mp[8] << endl;
9      }
10     mp.insert(pair<int, string>(6, "元素6"));
11     return 0;
12 }
13
14 /*
15 上一步已经插入
16 */

```

### 104. 对象指针

对象指针就是用于存放对象地址的变量，对象指针遵循一般变量指针的各种规则。

注：在类中，对象同时包含了数据和函数两种成员，与一般变量略有不同，但是对象所占据的内存空间只是用于存放数据成员的，函数成员不在每个对象中存储副本。

```

1  class Point{
2  private:
3      int x, y;
4  public:
5      Point(int _x=0, int _y=0): x(_x), y(_y){}
6      int getX()const{

```

```

7         return x;
8     }
9     int getY()const{
10         return y;
11     }
12 };
13
14 int main(){
15     Point p1;
16     cout << sizeof(Point) << " " << sizeof(p1) << endl;
17     return 0;
18 }
19
20 /*
21 8 8
22 */

```

对象指针的声明以及使用

```

1 class Point{
2 private:
3     int x, y;
4 public:
5     Point(int _x=0, int _y=0): x(_x), y(_y){}
6     int getX()const{
7         return x;
8     }
9     int getY()const{
10        return y;
11    }
12 };
13
14 int main(){
15     Point p1(1, 2);
16     Point* pp = &p1;
17
18     // 通过对象指针调用对象成员的两种方法
19     cout << pp->getX() << " " << (*pp).getY() << endl;
20     return 0;
21 }
22
23 /*
24 1 2
25 */

```

如下面的程序是错误的：

```

1 class B; // 前向引用声明
2 class A{
3 private:
4     B b;
5 };
6
7 class B{
8 public:
9     int data;

```



```

10 };
11
12 int main(){
13     A a;
14     return 0;
15 }
16
17 /*
18 error: field 'b' has incomplete type 'B'
19 */

```

尽管使用了前向引用声明，但是在提供一个完整的类定义之前，不能定义该类的对象，也不能在内联成员函数中使用该类的对象

下面的程序语法上是正确的

```

1  class B;    // 前向引用声明
2  class A{
3  private:
4      B* b;
5  };
6
7  class B{
8  public:
9      int data;
10 };
11
12 int main(){
13     A a;
14     return 0;
15 }

```

实际操作会发现，只有这样才行得通。

```

1  class B;    // 前向引用声明
2  class A{
3  private:
4      B* b;
5  public:
6      A(){
7          b = NULL;
8      }
9      ~A();
10     void setData(int val);
11     void showData();
12
13 };
14
15 class B{
16 public:
17     int data;
18 };
19
20 void A::setData(int val){
21     if(b == NULL){
22         b = new B();
23     }

```

```

24     b->data = val;
25 }
26
27 void A::showData(){
28     cout << "data " << b->data << endl;
29 }
30 A::~A(){
31     delete b;
32 }
33
34 int main(){
35     A a;
36     a.setData(3);
37     a.showData();
38     return 0;
39 }
40
41 /*
42 data 3
43 */

```

指针只是告诉编译器，既然我用了前向引用声明你还报错是因为即便有前向引用声明，你编译器还是不知道要申请的内存大小，那我就只是声明一个指针，指针的大小你总知道吧😁😁😁，动手实现了下，我谢谢你噢，说到底编译器还是得知道这个对象到底要占用多大内存，就连析构函数都得放在B的定义之后。

#### 105. this指针

this指针（一定是一个指针常量）是一个隐含于每一个类的非静态成员函数中的特殊指针(包括构造函数和析构函数).它用于指向正在被成员函数操作的对象。this指针实际上是类成员函数的一个隐含参数。在调用类的成员函数时，目的对象的地址会自动作为该参数的值，传递给被调用的成员函数，这样被调函数就能够通过this指针来访问目的对象的数据成员。对于常成员函数来说，这个隐含的参数是常指针类型的。

#### 106. 指向类的非静态成员的指针

```

1  class Point{
2  private:
3      int x, y;
4  public:
5      int val;
6      Point(int _x=0, int _y=0): x(_x), y(_y){
7          val = x+y;
8      }
9      int getX()const{
10         return x;
11     }
12     int getY()const{
13         return y;
14     }
15 };
16
17
18 int main(){
19     Point p(1, 2);
20
21     //指向对象数据成员的指针
22     int Point::*pSum = &Point::val; //★定义时和类关联

```

```

23 //指向对象函数成员的指针
24 int (Point::*pGetX)() const = &Point::getX; //★定义时和类关联 (const也是函数的一部分)
25
26 cout << p.*pSum << endl; //★使用时和对象关联
27 p.*pSum = 5;
28 cout << p.val << endl; //★使用时和对象关联
29
30 cout << (p.*pGetX)() << endl; //★使用时和对象关联
31 return 0;
32 }
33
34 /*
35 3
36 5
37 1
38 */

```

可能会想 &Point::val; 不就是一个 int\* 类型的吗，为什么还要这么麻烦，直接 int\* ps = &Point::val; \*ps = 5; 不就可以了，很遗憾，不是这样的。原因是对于类的非静态成员来说，该成员依赖于对象（即类的非静态成员只能通过类的对象调用，这也就是为什么静态成员函数里面只能涉及到静态成员），因此不可以用普通的指针来指向和访问非静态成员。

```

1 error: cannot convert 'int Point::*' to 'int*' in initialization
2 int* ps = &Point::val;

```

## 107. 指向类的静态成员的指针

```

1 class Point{
2 private:
3     int x, y;
4 public:
5     static int cnt;
6     Point(int _x=0, int _y=0): x(_x), y(_y){
7         cnt++;
8     }
9     int getX()const{
10         return x;
11     }
12     int getY()const{
13         return y;
14     }
15 };
16
17 int Point::cnt = 0;
18
19
20 int main(){
21     Point p1(1, 2);
22     Point p2(3, 4);
23
24     int* pCnt = &Point::cnt;
25     cout << *pCnt << endl;
26     return 0;

```

```

27 }
28
29 /*
30 2
31 */

```

指向类的静态成员的指针的定义以及使用就如同静态成员本身的定义一样，被所有对象所共有。

#### 108. 适配器

在C++中有一个适配器的概念，从名称就可以理解。所谓适配器就是将已有的数据结构进行一个封装，得到另一个数据结构，新的数据结构很多功能大都来源于原有的数据结构。

如，可以将vector容器进行封装得到一个Stack，栈的push(), pop(), empty(), top()都来源于vector的方法，同时这个栈没有vector容器的插入方法。

#### 109. C++中的临时对象

C++中有时程序会产生临时对象，临时对象可以是隐式产生的，也可以是显示产生的。临时对象的特点就是它的生存期，只在产生临时对象的一条表达式中，它的生命周期只有一条语句的时间，作用域也仅仅在一条语句中，当语句执行完就自动被析构了。我们是看不见临时变量的，也没有名称，但是它却实际存在着，并影响着程序运行效率的。

```

1  class A{
2  private:
3      int val;
4  public:
5      A(int v): val(v){}
6      A(){
7          A(3);
8      }
9      A(const A& a){
10         val = a.val;
11         cout << "调用复制构造函数" << endl;
12     }
13     void show(){
14         cout << "val=" << val << endl;
15     }
16 };
17
18 // 隐式的临时对象
19 A f(int val){
20     A a(val);
21     return a;
22 }
23
24
25 int main(){
26     A a;
27     a.show();
28     A b = A(1); // 显示的临时对象
29     b.show();
30     A c = f(3); // f()返回A类型时会产生一个隐式临时对象
31     c.show();
32     return 0;
33 }
34 /*
35 val=0
36 val=1

```

```
37 val=3
38 */
```

对于A a; a.show();语句,为什么结果会是val=0,这个里面就涉及到临时对象的原因,直接调用构造函数,会产生一个临时对象,由于临时变量作用域和生命周期都只在一条一句上有效,即调用构造函数本身这条语句,所以无参构造函数的函数体和空函数并没有什么区别,这也是为什么打印的是随机值了。

对于临时对象,需要知道,临时对象是性能的瓶颈,也是Bug的来源之一。

## 110. 作用域

作用域是一个标识符在程序正文中有效的区域。C++中标识符的作用域有函数原型作用域、局部作用域(块作用域)、类作用域和命名空间作用域。

- 函数原型作用域
  - 函数原型中的参数,其作用域始于"(",结束于")"。
  - 标识符具有最小的作用域,特指函数的声明。
- 局部作用域
  - 函数的形参、在块中声明的标识符;
  - 其作用域自声明处起,限于块中。
  - 具有局部作用域的变量也称为局部变量。
- 类作用域
  - 类的成员具有类作用域
- 命名空间作用域
  - 不属于前面所述各个作用域的标识符,都属于该命名空间作用域。
  - 具有命名空间作用域的变量也称为全局变量。
    - 注:全局变量默认都是外部变量,外部变量可以看作是可以被多个源文件共享的变量,但其它文件在使用这个外部变量时,必须要用extern关键字声明,对外部变量的声明可以是定义性声明,即在声明的同时定义(分配内存,初始化),也可以是引用性声明(引用在别处定义的变量),可以有多个引用性声明(需要共享的文件),但是只能有一次定义性声明。

```
1 void f(int val); // val具有函数原型作用域
2
3 int main(){
4     int val = 5;
5     f(val);
6     cout << "val=" << val << endl;
7     return 0;
8 }
9
10 void f(int val){ // val具有块作用域
11     int& v = val;
12     v = 9;
13     cout << "val=" << val << endl;
14 }
15
16 /*
17 val=9
18 val=5
19 */
```

## 111. 可见性

可见性是标识符是否可以引用的问题

- 主要是指如果在两个或多个具有包含关系的作用域中声明了同名标识符，则外层标识符在内层不可见。

## 112. 生存期

对象从诞生到结束的这段时间就是它的生存期，分为静态生存期和动态生存期两种。

- 静态生存期
  - 如果对象的生存期与程序的运行期相同，则称它具有静态生存期。
  - 有两种对象具有静态生存期
    - 在命名空间作用域中声明的对象。
    - 在函数内部的局部作用域中，使用static声明的对象
  - 静态变量的特点是，它并不会随着每次函数调用而产生一个副本，也不会随着函数返回而失效。也就是说，当一个函数返回后，下一次再调用时，该变量还会保持上一回的值，即使发生了递归调用，也不会为该变量建立新的副本，该变量会在每次调用间共享。

```
1 void f(){
2     static int val = 5;
3     cout << "val=" << val++ << endl;
4 }
5
6 int main(){
7     f();
8     f();
9     f(); // val在整个程序文件中只会被初始化一次，val在每次使用时共享一个内存
10    return 0;
11 }
12
13 /*
14 val=5
15 val=6
16 val=7
17 */
```

在函数内部定义的普通局部变量和静态局部变量的这种差异，实际上是计算机底层将这两种类型的变量存储于不同位置上。普通局部变量存储在栈区，静态局部变量存储在全局区。

- 动态生存期

除了上述两种情况，其余的对象都具有动态生存期。在局部作用域中声明的具有动态生存期的对象，习惯上也称为局部生存期对象。局部生存期对象诞生于声明点，结束于声明所在的块执行完毕之时。

## 113. 静态数据成员

如果某个属性为整个类所共有，不属于任何一个具体对象，则采用 static 关键字来声明为静态成员。静态成员在每个类只有一个副本，由该类的所有对象共同维护和使用，从而实现了同一类的不同对象之间的数据共享。

- 对于静态数据成员，一点更要注意的是，在类的定义中仅仅对静态数据成员进行引用性声明，必须在命名空间作用域的某个地方使用类名限定定义性声明。
- 之所以类的静态数据成员需要在类定义之外再加以定义，是因为需要以这种方式专门为它们分配空间。非静态数据成员不需要以此方式定义，因为它们的空间是与它们所属对象的空间同时分配的。

## 114. 将变量和函数限制在编译单元内

命名空间作用域中声明的变量和函数，在默认情况下都可以被其他编译单元访问，但有时并不希望一个源文件中定义的命名空间作用域的变量和函数被其他源文件引用。这时候有两种做法

- 在定义这些变量和函数时使用 `static` 关键字。 `static` 关键字用来修饰命名空间作用域的变量或函数时，和 `extern` 关键字起相反的作用，它会使得被 `static` 修饰的变量和函数无法被其他编译单元引用。
- 使用匿名的命名空间，在匿名命名空间中定义的变量和函数，都不会暴露给其他编译单元。

```
1 namespace{
2     int n;
3     void f(){
4         cout << "匿名空间" << endl;
5     }
6 }
7
8
9 int main(){
10     f();
11     return 0;
12 }
13
14 /*
15 匿名空间
16 */
```

## 115. C++ 预处理、编译、汇编、链接、执行

C++代码的执行过程可以分为预处理、编译、汇编、链接、执行。

- 预处理
  - 预处理主要将源程序中的宏定义指令、条件编译指令、[头文件](#)包含指令以及特殊符号完成相应的替换工作。不如说将预处理指令 `#include <iostream>`，中的 `iostream` 文件插入到源程序中。
- 编译
  - 以预编译的输出作为输入，利用C++运行库，通过词法分析和语法分析，在确认所有的指令都符合语法规则时，将其翻译成等价的中间代码表示或者是汇编语言。
    - 需要指出的是，目标文件代码段的目标代码中对静态生存期对象的引用和对函数的调用所使用的地址都是未定义的，因为它们的地址在连接阶段才能确定。
- 汇编
  - 将以汇编语言的形式存在的程序转化为机器可识别的二进制代码，从而得到相应的目标文件
- 链接
  - 在连接期间，需要将各个编译单元的目标文件和运行库当中被调用过的单元加以合并。
    - 经过合并后，不同编译单元的代码段和两类数据段就分别合并到一起了，程序在运行时代码和静态数据需要占据的内存空间就全部已知了，因此所有代码和数据都可以被分配确定的地址了。
  - 重定位信息这时也能发挥作用了，各段代码中未定义的地址，都可以被替换为有效地址。
    - 链接分为静态链接和动态链接
      - [静态链接和动态链接两者最大的区别就在于链接的时机不一样，静态链接是在形成可执行程序前，而动态链接的进行则是在程序执行时](#)。静态链接库与动态链接库都是共享代码的方式。
      - 静态链接

- 如果采用静态链接库，则无论你愿不愿意，lib中的指令都被直接包含在最终生成的EXE文件中了。
- 静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。
- 静态链接的缺点就是，一方面是空间浪费，另外一方面是更新困难。
- 动态链接
  - DLL(Dynamic Linkable Library)若使用DLL，该DLL不必被包含在最终的EXE文件中，EXE文件执行时可以“动态”地引用和卸载这个与EXE独立的DLL文件。
  - 动态链接库的优点：（1）更加节省内存；（2）DLL文件与EXE文件独立，只要输出接口不变，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。
- 执行
  - 程序的执行，是以进程为单位的。程序的一次动态执行过程称为一个进程。
  - 程序是存储在磁盘上的，在执行前，操作系统需要首先将它载入到内存中，并为它分配足够大的内存空间来容纳代码段和数据段，然后把文件中存放的代码段和初始化的数据段的内容载入其中。

注：下面的题号记录有误，导致实际题数因该比最后的题号多7，由于markdown的题号改了之后格式都不一样了，所以就这样了

109. 把数组作为参数时，一般不指定数组第一维的大小，即使指定，也会被忽略。

解释：[Why is it allowed to omit the first dimension, but not the other dimensions when declaring a multi-dimensional array?](#)。

110. c++和c中对常量的检查是不同的。

○ c

```

1  int main()
2  {
3      const int a = 9;
4      int* p = &a;
5      *p = 8;
6      printf("%d %d\n", a, *p);
7      return 0;
8  }
9
10 /*
11 t.c:12:14: warning: initialization discards 'const' qualifier from
    pointer target type [-wdiscarded-qualifiers]
12     int* p = &a;
13             ^
14 8 8
15 */

```

○ c++



```

1  int main(){
2      const int a = 9;
3      int* p = &a;
4      *p = 8;
5      printf("%d %d\n", a, *p);
6      return 0;
7  }
8
9  /*
10 t.cpp:20:14: error: invalid conversion from 'const int*' to 'int*'
    [-fpermissive]
11     int* p = &a;
12 */

```

### 111. 把数组当作指针使用

```

1  int main(){
2      int arr[3][4][5];
3      int cnt = 0;
4      for(int i = 0; i < 3; i++){
5          for(int j = 0; j < 4; j++){
6              for(int k = 0; k < 5; k++){
7                  arr[i][j][k] = i+j+k;
8              }
9          }
10     }
11
12     for(int i = 0; i < 3; i++){
13         for(int j = 0; j < 4; j++){
14             for(int k = 0; k < 5; k++){
15                 if(cnt++ % 15 == 0){
16                     cout << endl;
17                 }
18                 cout << arr[i][j][k] << " ";
19             }
20         }
21     }
22     cnt = 0;
23     cout << endl;
24     // 把数组当作指针使用
25     for(int i = 0; i < 3; i++){
26         for(int j = 0; j < 4; j++){
27             for(int k = 0; k < 5; k++){
28                 if(cnt++ % 15 == 0){
29                     cout << endl;
30                 }
31                 cout << *((*(arr+i)+j)+k) << " ";
32             }
33         }
34     }
35     return 0;
36 }
37
38 /*
39
40 0 1 2 3 4 1 2 3 4 5 2 3 4 5 6

```

```

41 3 4 5 6 7 1 2 3 4 5 2 3 4 5 6
42 3 4 5 6 7 4 5 6 7 8 2 3 4 5 6
43 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9
44
45 0 1 2 3 4 1 2 3 4 5 2 3 4 5 6
46 3 4 5 6 7 1 2 3 4 5 2 3 4 5 6
47 3 4 5 6 7 4 5 6 7 8 2 3 4 5 6
48 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9
49 */

```

指针数组与多维数组存在本质的差异（存储方式不同），但二者具有相同的访问形式，可以把多维数组当作指针数组来访问。

## 11.2. 类型兼容规则

类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。有三种用法

- 派生类对象可以隐含转换为基类对象
- 派生类的对象也可以初始化基类对象的引用
- 派生类对象的地址也可以隐含转换为指向基类的指针

虽然根据类型兼容规则，可以在基类对象出现的场合使用派生类对象进行替代，但是，如果不使用虚函数的话，替代之后派生类仅仅发挥出基类的作用。

类型兼容规则的底层实现原理是类的内存布局——对象地址加减偏移量

```

1  class Base1{
2  public:
3      void show()const{
4          cout << "Base1::display()" << endl;
5      }
6  };
7
8  class Base2:public Base1{
9  public:
10     void show()const{
11         cout << "Base2::display()" << endl;
12     }
13 };
14
15 class Derived:public Base2{
16 public:
17     void show()const{
18         cout << "Derived::display()" << endl;
19     }
20 };
21
22 void fun(Base1* p){
23     p->show();
24 }
25 int main(){
26     Base1 b1;
27     Base2 b2;
28     Derived d1;
29     fun(&b1);
30     fun(&b2);
31     fun(&d1);
32
33     return 0;

```

```

34 }
35
36 /*
37 Base1::display()
38 Base1::display()
39 Base1::display()
40 */

```

### 113. 派生类的复制构造函数

```

1  class A{
2  public:
3      A(int val){
4          cout << "A构造函数" << endl;
5      }
6      A(const A& a){
7          cout << "A复制构造函数" << endl;
8      }
9  };
10
11 class B: public A{
12 public:
13     B(int val): A(val){
14         cout << "B构造函数" << endl;
15     }
16     B(const B& b): A(b){
17         cout << "B复制构造函数" << endl;
18     }
19 };
20 int main(){
21     B b1(2);
22     B b2 = b1;
23     return 0;
24 }
25
26 /*
27 A构造函数
28 B构造函数
29 A复制构造函数
30 B复制构造函数
31 */

```

上面派生类B的复制构造函数在给A的复制构造函数传参时，传的是b的引用，这是因为类型兼容规则在这里起了作用：可以用派生类的对象去初始化基类的引用。因此当函数的形参是基类的引用时，实参可以是派生类的对象。

### 114. 虚基类

在类的派生过程中使用了 virtual 关键字，他的应用是在一个类有多个直接基类，而这多个基类有些都由同一个类派生而来。

- 在不使用虚基类的情况下，必须使用作用域标识符。

```

1  class Base{
2  public:
3      int var;
4      void show(){
5          cout << "Base: var=" << var << endl;

```

```

6     }
7 };
8
9 class A: public Base{
10
11 };
12
13 class B: public Base{
14
15 };
16
17 class C: public A, public B{
18
19 };
20 int main(){
21     C c;
22     cout << sizeof(c) << endl; // 为8, 说明确实有两份var的副本
23     c.A::var = 9;
24     c.B::var = 8;
25     c.A::show();
26     c.B::show();
27     return 0;
28 }
29
30 /*
31 8
32 Base: var=9
33 Base: var=8
34 */

```

在这种情况下，派生类对象在内存中就同时拥有成员 var 的两份同名副本。

- 使用虚基类

```

1 class Base{
2 public:
3     int var;
4     void show(){
5         cout << "Base: var=" << var << endl;
6     }
7 };
8
9 class A: virtual public Base{
10
11 };
12
13 class B: virtual public Base{
14
15 };
16
17 class C: public A, public B{
18
19 };
20 int main(){
21     C c;
22

```

```

23 // 为24，两个虚基类的指针，一个var的副本，8+8+4 = 20，内存对齐 20+4 =
24 24
25
26 cout << sizeof(c) << endl;
27
28 c.A::var = 9;
29 c.B::var = 8;
30 c.A::show();
31 c.B::show();
32 c.var = 7; // 使用了虚基类就可以直接这样访问
33 c.show();
34 return 0;
35 }
36
37 /*
38 24
39 Base: var=8
40 Base: var=8
41 Base: var=7
42 */

```

相比之下，前者可以容纳更多的数据，而后者使用更为简洁，内存空间更为节省

## 115. 类的组合与继承

类的组合与继承都使得已有对象成为新对象的一部分，从而达到代码复用的目的。但是它们两个是有所区别的。

### ◦ 类的组合

表示的是有一个，即整体与部分的关系

```

1 class Base{
2 public:
3     int var;
4     void show(){
5         cout << "Base: var=" << var << endl;
6     }
7 };
8
9 class A{
10 public:
11     Base b;
12 };

```

A类有一个Base类

### ◦ 类的继承

使用最为普遍的公有继承，反映的是一个的关系。

```

1  class Base{
2  public:
3      int var;
4      void show(){
5          cout << "Base: var=" << var << endl;
6      }
7  };
8
9  class B: public Base{
10
11 };

```

类B是一个Base类。

#### 116. 虚基类和虚函数的区别

虚基类：在类的派生过程中使用了 virtual 关键字，他的应用是在一个类有多个直接基类，而这多个基类有些都由同一个类Base派生而来。使用虚基类的情况主要是想使用类Base的成员，在这种情况下如果不适用虚基类，一方面会浪费内存空间，另一方面在实际使用的时候还需要用到作用域标识符，不太方便。

虚函数：是动态绑定的基础。如果需要通过基类的指针指向派生类的对象，并访问某个与基类同名的成员，那么首先在基类中将这个同名函数说明为虚函数。

#### 117. 派生类对象的内存布局

基类指针可以指向基类对象，也可以指向派生类对象，这时基类指针可以访问基类成员，也可以访问派生类成员，主要原因就是派生类对象的内存布局。**派生类对象的内存布局需满足的要求是，一个基类指针，无论其指向基类对象，还是派生类对象，通过它来访问一个基类中定义的数据成员，都可以用相同的步骤。**

- 单继承

```

1  class Base{
2  public:
3      int var;
4      void show(){
5          cout << "Base: var=" << var << endl;
6      }
7  };
8
9  class A: virtual public Base{ // 这里加不加virtual其实效果一样，只要基类
    不是虚函数就可以
10 public:
11     void show(){
12         cout << "A: var=" << var << endl;
13     }
14 };
15
16 int main(){
17     Base* b1 = new Base;
18     b1->show();
19     A* pA = new A;
20     b1 = pA;
21     b1->show();
22     return 0;
23 }
24
25 /*

```

```

26 Base: var=6496560
27 Base: var=7602512
28 */

```

■ 内存布局如下:

```

■ b1-----> |Base 成员      b2-----> |Base 成员
                                   |A成员

```

- 这种情况比较简单, 可以看出虽然指向的对象具有不同的类型, 但任何一个Base数据成员到该对象首地址都具有相同的偏移量, 因此使用 Base 指针 pba pbb 访问 Base 类中定义的数据成员时, 可以采用相同的方式, 而无须考虑具体的对象类型。

○ 多继承

```

1  class Base1{
2  public:
3      int var;
4      void show(){
5          cout << "Base1: var=" << var << endl;
6      }
7  };
8
9  class Base2{
10 public:
11     int var;
12     void show(){
13         cout << "Base2: var=" << var << endl;
14     }
15 };
16
17 class A: public Base1, public Base2{
18 public:
19     using Base1::var; // 使用using可以消除二义性
20     void show(){
21         cout << "A: var=" << var << endl;
22     }
23 };
24
25
26 int main(){
27     Base1* b1 = new Base1;
28     b1->show();
29     Base2* b2 = new Base2;
30     b2->show();
31
32     A* pA = new A;
33     b1 = pA;
34     b1->show();
35     b2 = pA;
36     b2->show();
37     return 0;
38 }
39
40 /*
41 Base1: var=39592240
42 Base2: var=1794864
43 Base1: var=1794864

```

```
44 Base2: var=0
45 */
```

■ 内存布局如下:

- b1-----> |Base1 成员                      b2-----> |Base2 成员
- pA-----> |Base1 成员  
                  |Base2 成员  
                  |A 成员

- pA 赋给 b1指针时, 与单继承时的情形相似, 只需要把地址复制一遍即可。但将 pA 赋给 b2指针时, 则不能简单地执行地址复制操作, 而应当在原地址的基础上加一个偏移量, 使 b2 指针指向 A 对象中 Base2 类的成员的首地址。可以看出, 指针转换并非都保持原先的地址不变, 地址的算术运算可能在指针转换时发生。但这又不是简单的地址算术运算, 因为如A指针的值为0, 则转换后的 Base2 指针值也应为0, 而非一个非0值。

○ 虚拟继承的情况 (不同的编译器有不同的处理方法)

```
1  class Base0{
2  public:
3      int var;
4      void show(){
5          cout << "Base: var=" << var << endl;
6      }
7  };
8
9  class Base1: virtual public Base0{
10
11  };
12
13  class Base2: virtual public Base0{
14
15  };
16
17  class A: public Base1, public Base2{
18
19  };
20  int main(){
21      Base0* p0 = new Base0;
22      Base1* p1 = new Base1;
23      Base2* p2 = new Base2;
24      A* pA     = new A;
25      p0->show();
26      p1->show();
27      p2->show();
28      pA->show();
29
30      p0 = p2;
31      p0->show();
32      p0 = pA;
33      p0->show();
34
35      return 0;
36  }
37
38  /*
39  Base: var=39395632
```



```

40 Base: var=2031952
41 Base: var=2031952
42 Base: var=0
43 Base: var=2031952
44 Base: var=0
45 */

```

■ 内存布局如下:

- p0-----> |Base0 成员
- p1-----> |Base0 指针
- | Base1新增成员
- | Base0 成员
- p2-----> |Base0 指针
- |Base2新增成员
- |Base0 成员
- pA-----> |Base0 指针
- |Base1新增成员
- |Base0 指针
- |Base2新增成员
- | A 新增成员
- | Base0 成员

#### 118. 基类向派生类的转换及其安全性问题

c++ 在转换种采取的处理方式:从特殊到一般是安全的, 因此允许隐含转换;从一般到特殊是不安全的, 因此只能显式地转换。

如:

- void\* 和 int\* 中, void\* 是一般的, int\* 是特殊的。
- 基类是一般的, 派生类是特殊的。

```

1  int main(){
2      int val = 9;
3      void* p1 = NULL;
4      int* p2 = &val;
5      // p2 = p1; // error
6      p1 = p2;
7      cout << *(int*)p1 << endl;
8      return 0;
9  }
10
11 /*
12 9
13 */

```

这里有一个问题需要做下说明。

- 基类对象一般无法被显式转换为派生类对象。而从派生类对象到基类对象的转换之所以能够执行, 是因为基类对象的复制构造函数 数接收一个基类引用的参数, 而用派生类对象是可以给基类引用初始化的, 因此基类的复制构造函数可以被调用, 转换就能够发生。

#### 119. 基类与派生类的对象、指针或引用之间, 哪些情况下可以隐含转换, 哪些情况下可以显示转换?在涉及多重继承或虚继承的情况下, 在转换时会面临哪些新问题?

- 子转父: 隐式, 反过来: 显示
- 执行基类指针到派生类指针的显示转换时, 有时需要将指针所存储的地址值进行调整后才能得到新指针的值, 但是如果A类是B类的虚基类, 虽然B类指针可以隐式转换为A类指针, 但A类

指针却无法通过 static\_cast 隐式转换为B类型的指针。

120. 不能被重载的操作符,

- 类属关系运算符 "."
- 成员指针运算符 "->"
- 作用域分辨符 "::"
- 和三目运算符 "? : "

121. 运算符重载示例

```
1  class Array{
2  private:
3      int* arr;
4      int size;
5  public:
6      Array(int size): size(size){
7          arr = new int[this->size];
8      }
9      ~Array(){
10         delete [] arr;
11     }
12
13     void construct(int begin){
14         for(int i = 0; i < size; i++){
15             arr[i] = begin++;
16         }
17     }
18     void show(){
19         for(int i = 0; i < size; i++){
20             cout << arr[i] << " ";
21         }
22         cout << endl;
23     }
24     int& operator[](int ind){
25         if(ind >= size){
26             cout << "越界" << endl;
27             // 这里不能定义一个局部变量返回——不能返回局部变量的引用
28             return arr[0];
29         }
30         return arr[ind];
31     }
32 };
33
34 int main(){
35     Array arr(5);
36     arr.construct(1);
37     arr.show();
38     cout << arr[2] << endl;
39     arr[2] = 10;
40     cout << arr[2] << endl;
41     return 0;
42 }
43
44 /*
45 1 2 3 4 5
46 3
47 10
48 */
```

## 122. 抽象类的实现与使用

- Derived未实现 show()。

```
1  class Base1{
2  public:
3      virtual void show(int i = 3)const=0;
4  };
5
6  class Base2:public Base1{
7  public:
8      void show(int i = 6)const{
9          cout << "Base2::display()" << i << endl;
10     }
11 };
12
13 class Derived:public Base2{
14
15 };
16
17 void fun(Base1* p){
18     p->show();
19 }
20 int main(){
21     Base2 b2;
22     Derived d1;
23     fun(&b2);
24     fun(&d1);
25
26     return 0;
27 }
28
29 /*
30 Base2::display()3
31 Base2::display()3
32 */
```

- Derived实现 show()。

```
1  class Base1{
2  public:
3      virtual void show(int i = 3)const=0;
4  };
5
6  class Base2:public Base1{
7  public:
8      void show(int i = 6)const{
9          cout << "Base2::display()" << i << endl;
10     }
11 };
12
13 class Derived:public Base2{
14 public:
15     void show(int i = 9)const{
16         cout << "Derived::display()" << i << endl;
17     }
18 };
```

```

19
20 void fun(Base1* p){
21     p->show();
22 }
23 int main(){
24     Base2 b2;
25     Derived d1;
26     fun(&b2);
27     fun(&d1);
28
29     return 0;
30 }
31
32 /*
33 Base2::display()3
34 Derived::display()3
35 */

```

123. 什么叫做流？流的提取和插入是指什么？

流是一种抽象，他管理生产者所产生的数据向消费者传递数据的过程，管理数据的流动。

提取：读操作，如cin

插入：写操作，如cout

这里所说的操作是程序对于键盘，屏幕等外设（操作系统将他们看为文件）而言的，比如写操作，是程序向powershell写入数据。

124. 文件的写入示例

```

1  int main(){
2      ofstream oFile;
3      ifstream iFile;
4      char cha;
5
6      for(int i = 0; i < 5; i++){
7          static int cnt = 1;
8          oFile.open("t1.txt", ios::app); // 追加模式，如果不存在该文件，会先
创建该文件
9          oFile << "123 " << cnt << endl;
10         oFile.close();
11         cnt++;
12     }
13
14
15     iFile.open("t1.txt", ios::app);
16     while (iFile.get(cha))
17     {
18         cout << cha;
19     }
20     iFile.close();
21     return 0;
22 }
23
24 /*
25 123 1
26 123 2
27 123 3

```

```
28 123 4
29 123 5
30 */
```

## 125. 异常处理程序, try, catch, throw示例

```
1  int test(int& val){
2      if(val == 2 || val == 3){
3          throw val;
4      }
5      else{
6          return 5;
7      }
8  }
9  int main(){
10     int arr[5];
11
12     for(int i = 0; i < 5; i++){
13         arr[i] = i;
14     }
15
16     try
17     {
18         for(int i = 0; i < 5; i++){
19             cout << test(arr[i]) << endl;
20         }
21     }
22     catch(int e)
23     {
24         cout << "捕捉到异常 为" << e << endl;
25     }
26     cout << "测试结束" << endl;
27     return 0;
28 }
29
30 /*
31 5
32 5
33 捕捉到异常 为2
34 测试结束
35 */
```

## 126. 变量的存储类型以及生存期

有四种存储类型: auto、register、extern、static

- auto

具有局部作用域的变量都默认是auto型的, 这些变量因函数的调用而存在, 因函数调用结束而消失

- register

告诉编译器把这个变量放到cpu的寄存器中存储, 这样可以加快访问速度

- extern

其他文件想要共享一个全局变量, 可以声明为extern型

- static

使变量成为静态变量，该变量从定义知道程序运行结束期间一直存在，在这期间仅被初始化一次，同时被声明为static的变量在其他文件都不可以被访问。