

1. 对单目运算符重载为友元函数时，可以说明一个形参。而重载为成员函数时，不能显式说明形参

1. 答案：T

2. 解析：

1. 不能重载的运算符：., ::, *, ?:, sizeof
2. 只能作为成员函数：=, (), [], ->
3. 只能重载为友元函数：<<（输出），>>（输入）
4. 对于单目运算符，建议选择成员函数；双目运算符，最好重载为友元函数
5. 对于运算符“+ =, -=, /=, *=, &=, !=, ~=, %=, <=, >=”建议重载为成员函数
6. 对于其他运算符，建议重载为友元函数
7. 双目运算符一般可以重载为友元运算符函数或成员运算符函数，但当两个操作数的类型不一致时，必须使用友元函数

2. 使用提取符(<<)可以输出各种基本数据类型的变量的值，也可以输出指针值。

1. 答案：T

3. 重载operator+时，返回值的类型应当与形参类型一致。

比如以下程序中，operator+的返回值类型有错：

```
class A {
```

```
    int x;
```

```
public:
```

```
    A(int t=0):x(t){    }
```

```
    int operator+(const A& a1){ return x+a1.x;  }
```

```
};
```

1. 答案：F

2. 解析：A有参数为int的构造函数，在operator+返回的时候，会自动调用A(int t)将int转化成A类型

4.

关于纯虚函数和抽象类的描述中，（ ）是错误的。

- ☐ A. 纯虚函数是一种特殊的虚函数，它没有具体的实现
- ☒ B. 抽象类是指具有纯虚函数的类
- ☐ C. 一个基类中说明有纯虚函数，该基类的派生类一定不再是抽象类
- ☐ D. 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出

答案错误: 0 分

 创建提问

1. 答案: A

2. 解析: 纯虚函数的实现在派生类之中

5. 重载函数允许使用默认参数，但是要注意二义性

6. 子类调用父类的构造函数

```
1  class A {
2  public:
3      int a;
4      A(int a) {
5          cout << "A:" << a << endl;
6          this->a = a;
7      }
8  };
9  class B : A{
10 public:
11     B(int a) :A(a) {
12         cout << "B:" << a << endl;
13         cout << this->a;
14     }
15 };
```

7. 要想使用基类指针引用子类对象，子类的继承类型必须是**public**

8. 写**class**的声明时，最后记着加;

9. **strcpy**只有两个参数**strcpy(char* target, char* source)**

10.

```

char operator [] (int i)  const  1 分
{
    if (i<m_len && i>=0) return m_pStr[i];
    throw  ERROR()  1 分 ;
}

char& operator [] (int i)  const  1 分
{
    if (i<m_len && i>=0) return m_pStr[i];
    ERROR e;
    throw e  1 分 ;
}

```

1. 1 `char& operator [] (int i);` //这里由于返回的是对成员函数的引用，故这个函数不能是 `const` 类型的，应该啥也不填

11. 模板类编程，在类外实现函数体的时候，记着要写返回值 `T& ARRAY<T>::`

```

1  template <typename T>
2  T& ARRAY<T>::at(int index){
3      if (index<0 || index>m_size){
4          throw IndexError();
5      }
6      return m_ptr[index];
7  }

```

12. 如果不关心异常对象，则 `catch block` 中的参数可以被省略

1. 答案: T

2. 解析: 可以 `catch(...)`

13. 构造函数不能是 `virtual` 的，但析构函数可以是 `virtual` 的

14. 一旦函数被定义成是 `virtual` 的，在其派生类中的该函数一定是 `virtual` 的，不管是否有 `virtual` 关键字

15. 构造函数的调用顺序: **基类** → **成员变量** → **派生类**

析构函数的调用顺序: **派生类** → **成员变量** → **基类**

16. 下列代码的输出是

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  class A {
5      int i;
6  public:
7      A() :i(0) {}
8      ~A() { cout << get(); }
9      void set(int i) { this->i = i; }

```

```

10     int get() { return i; }
11 };
12 int main() {
13     A* p = new A[2];
14     delete p;
15     return 0;
16 }

```

1. 答案: 0

2. 解析: **delete**的时候, 使用的是**delete p**, 仅仅会删除**p**指向的第一个A类型的元素, 因此只会输出一个0

17. 类型转换:

```

1 static_cast<type>(data); //不允许const -> 非const
2 dynamic_cast<type>(data); //会检查类指针的downcast是否安全
3 const_cast<type>(data); //const -> 非const
4 reinterpret_cast<type>(data); //指针 <-> 地址值

```

18. 下列代码的输出是

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 class C {
5 public:
6     explicit C(int) {
7         cout << "i" << endl;
8     }
9     C(double) {
10         cout << "d" << endl;
11     }
12 };
13 int main() {
14     C c1(7);
15     C c2 = 7;
16     return 0;
17 }

```

1. 答案: i d

2. 解析: c1(7)是显式调用, 会调用C(int),

c2=7是隐式调用, 不会调用C(int), 而是将7类型转换为double, 然后调用C(double)

19. 下列代码的输出是

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 class A {
5     int s[10];
6 public:
7     int operator[](int i) const {
8         cout << "operator[](int) const" << endl;
9         return s[i];
10    }
11    int& operator[](int i) {

```

```

12         cout << "operator[](int)" << endl;
13         return s[i];
14     }
15 };
16 int main() {
17     A a1;
18     const A& a2 = a1;
19     a1[0] = a2[1];
20     return 0;
21 }

```

1. 答案:

```

operator[] (int) const
operator[] (int)

```

2. 解析:

1. **a1[0] = a2[1]**, 会先执行等号右边, 再执行等号左边
2. **a2**是**const A&**类型的, 会调用**operator[] (int i) const**;
3. **a1**是**A**类型的, 会调用**operator[] (int i)**;

20. 下列代码的输出是

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  class A {
5  public:
6      virtual ~A() {};
7  };
8  class B : public A{};
9  int main() {
10     A a;
11     B b;
12
13     A* ap = &a;
14     if (dynamic_cast<B*>(ap)) cout << "OK1" << endl;
15     else cout << "FAIL" << endl;
16
17     if (static_cast<B*>(ap)) cout << "OK2" << endl;
18     else cout << "FAIL" << endl;
19
20     ap = &b;
21
22     if (dynamic_cast<B*>(ap)) cout << "OK3" << endl;
23     else cout << "FAIL" << endl;
24
25     if (static_cast<B*>(ap)) cout << "OK4" << endl;
26     else cout << "FAIL" << endl;
27     return 0;
28 }

```

1. 答案:

FAIL
OK2
OK3
OK4

2. 解析:

1. 将指向父类A的父类指针转化为指向子类B的指针, **dynamic_cast**会认为是不安全的, 返回**nullptr**
2. **static_cast**不会判断类的继承关系
3. 将指向子类B的父类指针转化为指向子类B的指针, **dynamic_cast**可以转换
4. **static_cast**不会判断类的继承关系

21. 下列代码的输出是

```
1  #include <iostream>
2  using namespace std;
3  struct Base{
4      virtual ~Base() {
5          cout << "Destructing Base" << endl;
6      }
7      virtual void f() {
8          cout << "I'm in Base\n";
9      }
10 };
11 struct Derived : public Base{
12     ~Derived() {
13         cout << "Destructing Derived\n";
14     }
15     void f() {
16         cout << "I'm in Derived\n";
17     }
18 };
19 int main() {
20     Base* p = new Derived();
21     (*p).f();
22     p->f();
23     delete p;
24 }
```

1. 答案:

I'm in Derived
I'm in Derived
Destructing Derived
Destructing Base

2. 解析:

1. $(*p).f()$ 和 $p \rightarrow f()$ 是等价的
2. 析构p的时候, 先调用子类的析构函数, 再调用父类的析构函数

22.

