

1、下列代码的结果是

1	main()
2	{
3	Int a[5]={1, 2, 3, 4, 5};
4	Int *ptr=(int*) (&a+1);
5	printf("%d,%d",*(a+1),*(ptr-1));
6	}

- 3, 5
- 2, 4
- **2, 5**
- 3, 4

**解释：**&a 的类型是 int (\*p) [5], p is a pointer which points to array. &a+1 指向 a[5]的地址(越界)，之后强制转换为 int\*

2、下列关于联合的描述中，错误的是？

- **联合变量定义时不可初始化**
- 联合的成员是共址的
- 联合的成员在某一个时刻只有当前的是有效的
- 联合变量占有的内存空间是该联合成员占有最大内存空间的成员所需的存储空间

**解释：**

1. 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）。

2. 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。

3. union 类型的变量在定义时是可以被初始化的，定义如下 union 类型

1	union Test
2	{
3	Int a;
4	Float b;
5	};
6	Test test = {1};

test 变量的定义可以初始化，**初始值的类型必须是 union 中第一个成员的类型。**

3、下列代码的输出为：

```
1  classCParent
2  {
3      public: virtual  void Intro()
4      {
5          printf( "I'm a Parent, "); Hobby();
6      }
7      virtual  void Hobby()
8      {
9          printf( "I like football!");
10     }
11 };
12 Class CChild :  public CParent {
13     public: virtual  void Intro()
14     {
15         printf( "I'm a Child, "); Hobby();
16     }
17     virtual  void Hobby()
18     {
19         printf( "I like basketball!\n");
20     }
21 };
22 Int main(void)
23 {
24     CChild *pChild =  new CChild();
25     CParent *pParent = (CParent *) pChild;
26     pParent->Intro();
27     return(0);
```

28	}
----	---

- I'm a Parent, I like football!
- I'm a Parent, I like basketball!
- **I'm a Child, I like basketball!**
- I'm a Child, I like football!

**解释：**一个类如果定义了虚函数，则不管是类中还是类外，对这个函数的调用都是通过对对象的虚函数表，所以 Intro() {hobby()}; 相当于 vptr[0] (this\*) {this->vptr[1] (this)}, 虚函数表的第一二项分别是 Intro() 和 hobby() 的地址，而且子类已经把这两个都重写了，所以调用的自然都是子类的函数。

4、

1	classA
2	{
3	public:
4	A() { cout<<"A"<<endl; }
5	~A() { cout<<"~A"<<endl; }
6	};
7	classB:publicA
8	{
9	public: B(A &a): _a(a)
10	{
11	cout<<"B"<<endl;
12	}
13	~B() { cout<<"~B"<<endl; }
14	private: A _a;
15	};
16	voidmain(void)
17	{ A a;
18	B b(a);
19	}

结果为

1	A
2	A
3	B
4	~B
5	~A
6	~A
7	~A

1	加上拷贝构造函数-----
2	-----
3	classA
4	{
5	public: A() { cout<<"A"<<endl; }
6	A(constA& other) { cout<<"copy A"<<endl;}
7	~A() { cout<<"~A"<<endl; } };
8	classB:publicA
9	{
10	public:
11	B(A &a): _a(a)
12	{
13	cout<<"B"<<endl;
14	}
15	~B() { cout<<"~B"<<endl; }
16	private: A _a;
17	};
18	voidmain(void)
19	{
20	A a;
21	B b(a);

	}
--	---

1	结果显示
2	A
3	A
4	copy A
5	B
6	~B
7	~A
8	~A
9	~A
1	修改后----- -----classA { public: A() { cout<<"A"<<endl; } A(constA& other){ cout<<"copy A"<<endl;} ~A() { cout<<"~A"<<endl; } }; classB:publicA { public: B(A &a) { cout<<"B"<<endl; } ~B() { cout<<"~B"<<endl; } private: A _a; }; void main(void) { A a; B b(a); }

1	结果为
2	A
3	A
4	A
5	B
6	~B
7	~A
8	~A
9	~A

**解释：**派生类的构造函数都会先调用基类的构造函数

5、`int x[6][4], (*p)[4]: p=x;` 则`*(p+2)`指向哪里？

- `x[0][1]`
- `x[0][2]`
- `x[1][0]`
- **`x[2][0]`**

**解释：**`x` 为二维数组，`p` 是一个数组指针，将 `p` 指向长度为 4 的 `int` 数组，那么 `p` 指向的元素是 `x` 的第一行元素的首个，`p+2` 指的就是第三行的首个元素，所以 `p[2]` 所指即为 `x[2][0]`

6、在重载运算符函数时，下面（）运算符必须重载为类成员函数形式（）

- `+`
- `-`
- `++`
- `->`

**解释：**

大多数操作符可以定义为成员函数或非成员函数。当操作数为成员函数时，他的第一个操作数隐式绑定到 `this` 指针。有些操作符（包括赋值操作符）必须是定义自己的类的成员。因为赋值必须是类的成员，所以 `this` 绑定到指向左操作数的指针。因此，赋值操作符接受单个形参，且该形参是同一类类型的对象。右对象一般作为 `const` 引用传递。

`[]`、`()`、`->`、`=`这几个运算符如果要重载, 必须重载为成员函数

7、其中

`strlen(a)`是 3，`sizeof(a)`是 4，

`strlen(b)`是未知，

**sizeof(a)**是 4,

**sizeof(b)**是 3

```
chara[] = "xyz", b[] = {'x', 'y', 'z'};
int p = strlen(a); //3
int q = strlen(b); //未知
int m = sizeof(a); //4
int n = sizeof(b);//3
if(strlen(a) > strlen(b))
    printf("a > b\n");
else
    printf("a <= b\n");
```

## 8、运算符++重载

```
1. class AA
2. {
3. private:
4.     int a;
5. public:
6.     AA():a(0) {};
7.     AA& operator ++()// prefix
8.     {
9.         cout <<"++AA"<<endl;
10.        ++a;
11.        return *this;
12.    }
13.
14.    AA operator ++(int)// suffix
15.    {
16.        cout <<"AA++"<<endl;
17.        AA temp = *this;
18.        ++a;
19.        return temp;
20.    }
21.
22.    void show()
23.    {
24.        cout<<"int: " <<a<<endl;
25.    }
26. };
```

## 9、可以用顺序存储的下标表示指针，存储树状结构。

10、一个有序数列，序列中的每一个值都能够被 2 或者 3 或者 5 所整除，这个序列的初始值从 1 开始，但是 1 并不在这个数列中。求第 1500 个值是多少？

- 2040
- 2042
- 2045
- 2050

**解释：**2、3、5 的最小公倍数是 30。[ 1, 30]内符合条件的数有 22 个。如果能看出[ 31, 60]内也有 22 个符合条件的数，那问题就容易解决了。也就是说，这些数具有周期性，且周期为 30。

第 1500 个数是：1500/22=68 1500%22=4。也就是说：第 1500 个数相当于经过了 68 个周期，然后再取下一个周期内的第 4 个数。一个周期内的前 4 个数：2, 3, 4, 5。

故，结果为 68\*30=2040+5=2045

2040/2 = 1020 .... 2040 前有 1020 个能被 2 整除的数

2040/3 = 680 .... 2040 前有 1020 个能被 3 整除的数

2040/5 = 408 .... 2040 前有 1020 个能被 5 整除的数

2040/6 = 340 .... 2040 前有 1020 个能被 6 整除的数（2 和 3 的最小公倍数为 6）

2040/10 = 204 .... 2040 前有 1020 个能被 10 整除的数（2 和 5 的最小公倍数为 10）

2040/15 = 136 .... 2040 前有 1020 个能被 15 整除的数（3 和 5 的最小公倍数为 15）

2040/30 = 68 .... 2040 前有 1020 个能被 30 整除的数（2、3、5 的最小公倍数为 30）

那么 1020+680+408-340-204-136+68=1496（说明 2040 是该数列中的第 1497 个元素（加上第一个元素 1））

所以第 1498 是 2042，第 1499 是 2043，第 1500 是 2045

11、为什么线性表在顺序存储时，查找第 i 个元素的时间同 i 的值无关

顺序存储是先根据数据量的需要先分配好存储空间的，相当于先给数据分好了带编号的座位，所以可以直接找到。而链式是不事先定好存储空间的，就是第一个数据好了再给存第二个，且有个指针区指向下个数据的位置，所以要想找到第几个数据都要从头来

12、 函数 fun 的声明为 int fun(int \*p[4]), 以下哪个变量可以作为 fun 的合法参数（）

- int a[4][4];
- **int \*\*a;**
- int \*\*a[4]
- int (\*a)[4];

13、

```
time_t t;
```

哪个选项可以将 t 初始化为当前程序的运行时间？



```
t = clock();
```

14、c 里面的 struct 只是变量的聚合体，struct 不能有函数

15、下列选项中，正确的 C++标识符是

- 6\_group
- group~6
- age+3
- **\_group\_6**

**解释：**在 C 语言中,标识符的命名规则是:由字母(大、小写皆可)、数字及下划线组成，且第一个字符必须是字母或者下划线，长度为 8 位。在 C 语言中，大写字母和小写字母是有区别的，即作为不同的字母来看待，应引起注意。

A 以数字开头

B 中有非法符号~ C 中有非法符号+

D 正确

16、写出判断 ABCD 四个表达式的是否正确，若正确，写出经过表达式中 a 的值(3 分)

```
int a = 4;
```

(A) a += (a++); (B) a+= (++a) ; (C) (a++) += a; (D) (++a) += (a++);

a = ?

**解释：**C 错误，左侧不是一个有效变量，不能赋值，可改为(++a) += a;

改后答案依次为 9, 10, 10, 11

17、某 32 位系统下，C++程序，请计算 sizeof 的值(5 分)。

```
char str[] = "http://www.ibegroup.com/"
```

```
char *p = str ;
```

```
int n = 10;
```

请计算

sizeof (str ) = ? (1)

sizeof ( p ) = ? (2)

sizeof ( n ) = ? (3)

```
void Foo ( charstr[100]){
```

请计算

sizeof( str ) = ? (4)

```
}
```

```
void *p = malloc(100 );
```

请计算

sizeof ( p ) = ? (5)

**解释：** (1) 25 (2) 4 (3) 4 (4) 4 (5) 4

18、strcpy(str, “hello”);如果 str 指针内存不够，函数会报错，[SegmentFault](#)

19、C++中为什么用模板类。

**解释：** (1) 可用来创建动态增长和减小的数据结构

(2) 它是类型无关的，因此具有很高的可复用性。

(3) 它在编译时而不是运行时检查数据类型，保证了类型安全

(4) 它是平台无关的，可移植性

(5) 可用于基本数据类型

20、函数模板与类模板有什么区别？

**解释：** 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。

21、定义一个函数指针，指向的函数有两个 int 形参并且返回一个函数指针，返回的指针指向一个有一个 int 形参且返回 int 的函数

**int (\*(\*F)(int, int))(int)**

**解释：**

1. 函数指针定义

函数类型 (\*指针变量名)(形参列表);

“函数类型”说明函数的返回类型，由于“()”的优先级高于“\*”，所以指针变量名外的括号必不可少，后面的“形参列表”表示指针变量指向的函数所带的参数列表。

例如：

int (\*f)(int x);

double (\*ptr)(double x);

在定义函数指针时请注意：

函数指针和它指向的函数的参数个数和类型都应该是一致的；

函数指针的类型和函数的返回值类型也必须是一致的。

22、Given a string with n characters, suppose all the characters are different from each other, how many different substrings do we have?

- n+1
- $n^2$
- **$n(n+1)/2$**
- $2^{n-1}$
- n!

**解释：** 求子串，首先必须是连续的， $n + (n-1) + (n-2) \dots + 1 = n(n+1)/2$

23、假设在一个 32 位 little endian 的机器上运行下面的程序，结果是多少？

```
#include
int main(){
    long long a = 1, b = 2, c = 3;
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

- 1, 2, 3
- **1, 0, 2**
- 1, 3, 2
- 3, 2, 1

**解释：** long 占 8 个字节，假设其实地址是 0，一个地址存一个字节，按照地位编址 1, 2 会按照如下方式存储

地址 0 1 2 3 4 5 6 7 8 ...

内容 1 0 0 0 0 0 0 0 2 ...

%d 只能按照四字节连续输出

那么第一个输出的便是 0001=1 第二个输出 0000=0 第三个输出 0002

24、下面程序的执行结果：

1	class A{
2	public:
3	long a;
4	};
5	class B : public A {
6	public:
7	long b;
8	};
9	void seta(A* data, int idx) {
10	data[idx].a = 2;
11	}
12	int main(int argc, char *argv[]) {
13	B data[4];
14	for(int i=0; i<4; ++i){

15	data[i].a = 1;
16	data[i].b = 1;
17	seta(data, i);
18	}
19	for(int i=0; i<4; ++i){
20	std::cout << data[i].a << data[i].b;
21	}
22	return 0;
23	}

- 11111111
- 12121212
- 11112222
- 21212121
- **22221111**

**解释：**这道题应该注意 指针类型加减 时步长的问题。

A 大小为 4

B 大小为 8

那么：

```
void seta(A* data, int idx) {
    data[idx].a = 2;
}
```

由于传入的实参为 B 类型，大小为 8，而形参为 A 类型，大小为 4，data[idx] 取 data + idx 处的元素，这时指针 data 加 1 的长度不是一个 B 长度，而是一个 A 长度，或者说是 1/2 个 B 长度。这时该函数中 data[0~3] 指向的是原 data[0].a, data[0].b, data[1].a, data[1].b，由于隐式类型转换的缘故，data[0].a, data[0].b, data[1].a, data[1].b 处的值全部由于 data[idx].a = 2; 操作变为 2。

这道题如果改为 void seta(B\* data, int idx)，那么形参中 data 指针加 1 步长为 8，结果就是 21212121。但是由于步长为 4，所以结果就是 22221111。

25、下列选项中，会导致用户进程从用户态切换到内核的操作是？

I. 整数除以零    II. sin( ) 函数调用    III. read 系统调用

- 仅 I、II
- **仅 I、III**
- 仅 II 、III
- I、II 和 III

**解释：**用户态切换到内核态的 3 种方式

- a. 系统调用
- b. 异常
- c. 外围设备的中断

26、The Orchid Pavilion(兰亭集序) is well known as the top of “行书” in history of Chinese literature. The most fascinating sentence is “Well I know it is a lie to say that life and death is the same thing, and that longevity and early death make no difference Alas!”(固知一死生为虚诞，齐彭殇为妄作). By counting the characters of the whole content (in Chinese version), the result should be 391(including punctuation). For these characters written to a text file, please select the possible file size without any data corrupt.

- 782 bytes in UTF-16 encoding
- 784 bytes in UTF-16 encoding
- 1173 bytes in UTF-8 encoding
- 1176 bytes in UTF-8 encoding
- None of above

**解释：**如果只是论一个汉字占用的字节数，那么 UTF-8 占用 3 个字节， UTF-16 占用 2 个字节。但是如果存储文本的话，需要在文本使用 EF BB BF 三个字节表示使用 UTF-8 编码，使用 FE FF 表示使用 UTF-16 编码。

UTF-16 固定表示两个字节表示一个字符，不管是字母还是汉字； UTF-8 使用 1- 3 个字节表示一个字符

- 0xxxxxxx 一个字节兼容 ASCII，能表示 127 个字符
- 110xxxxx 10xxxxxx. 如果是这样的格式，则把两个字节当一个字符
- 1110xxxx 10xxxxxx 10xxxxxx 如果是这种格式则是三个字节当一个字符

所以，UTF-8 的空间是根据保存的内容不同而不同。如果保存的汉字多，使用 UTF-16 占用字符数双倍的空间，使用 UTF-8 占用字符数三倍的空间；如果保存的英文字母多，使用 UTF-16 使用字符数双倍的空间，使用 UTF-8 使用字符数相同的空间。

所以，不同情况下有不同的选择，这也是这么多字符集和编码格式存在的原因之一。

27、 c++中，声明 `const int i`, 是在哪个阶段做到 `i` 只可读的？

- **编译**
- 链接
- 运行
- 以上都不对

**解释：** `const` 用来说明所定义的变量是只读的。 这些在编译期间完成，编译器可能使用常数直接替换掉对此变量的引用。`const int i = 10;` 编译时候 就和 变量 `i` 做了对应, 后面程序用到 `i` 的时候, 直接从编译器的符号表中取 10, 不会再查找内存...

29、指出下面程序哪里可能有问题？

1	class CBuffer
2	{
3	char * m_pBuffer;
4	int m_size;
5	public:
6	CBuffer()
7	{
8	m_pBuffer=NULL;
9	}
10	~CBuffer()
11	{
12	Free();
13	}
14	void Allocte(int size) (1) {
15	m_size=size;
16	m_pBuffer= new char[size];
17	}
18	private:
19	void Free()
20	{
21	if(m_pBuffer!=NULL) (2)
22	{
23	delete m_pBuffer;
24	m_pBuffer=NULL;
25	}
26	}
27	public:
28	void SaveString(const char* pText) const (3)

29	{
30	strcpy(m_pBuffer, pText); (4)
31	}
32	char* GetBuffer() const
33	{
34	return m_pBuffer;
35	}
36	
37	};
38	
39	void main (int argc, char* argv[])
40	{
41	CBuffer buffer1;
42	buffer1.SaveString("Microsoft");
43	printf(buffer1.GetBuffer());
44	}

- 1
- 2
- 3
- 4

### 解释：

- (1) const 成员函数表示不会修改数据成员，而 SaveString 做不到，去掉 const 声明
- (2) m\_pBuffer 指向 NULL，必须用 Allocte 分配空间才能赋值。
- (3) 另外需要将 Allocte 成员函数声明为私有成员函数更符合实际
- (4) delete m\_pBuffer, 如果是内置类型不会造成内存泄露，不够不建议使用，如果操作对象是类对象会引起内存泄露；还是这种 delete[] m\_pBuffer 正规

30、代码执行后，a 和 b 的值分别为？

```
class Test{
public:
    int a;
    int b;
    virtual void fun() {}
```

```

Test(int temp1 = 0, int temp2 = 0)
{
    a=temp1 ;
    b=temp2 ;
}
int getA()
{
    return a;
}
int getB()
{
    return b;
}
};

int main()
{
    Test obj(5, 10);
    // Changing a and b
    int* pInt = (int*)&obj;
    *(pInt+0) = 100;
    *(pInt+1) = 200;
    cout << "a = " << obj.getA() << endl;
    cout << "b = " << obj.getB() << endl;
    return 0;
}

```

- 200 10
- 5 10
- 100 200
- 100 10

**解释：**头4个字节是虚函数指针

31、有以下程序

1	#include <stdio.h>
2	#include <stdio.h>
3	void fun( char *s )
4	{
5	char a[10];
6	strcpy ( a, "STRING" );

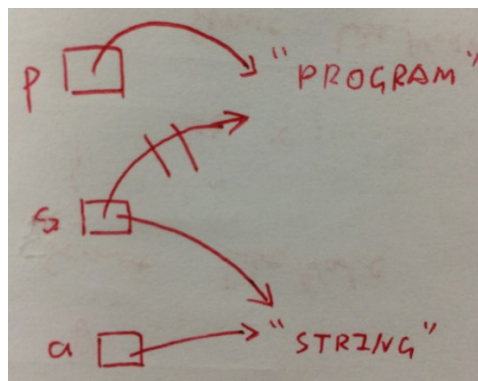


7	s = a ;
8	}
9	main( )
10	{
11	char *p= "PROGRAM" ;
12	fun( p );
13	printf ( "%s\n ", p) ;
14	}

程序运行后的输出结果是(此处□代表空格)?

- STRING
- STRING□□□□
- STRING□□□
- **PROGRAM**

解释:



32、 在链队列中, 即使不设置尾指针也能进行入队操作()

- **对**
- 错

**解释:** 遍历整个队列, 当一个元素的 next 为 null 时, 则此元素为最后一个, 在其后面添加新的元素即可。

33、下列数据结构具有记忆功能的是?

- 队列
- 循环队列
- **栈**

- 顺序表

**解释：**栈先进后出。最先进去的数肯定是最后出来的 所以说有记忆功能。比如我们往一个很窄的瓶子里放 1, 2, 3, 4 号球。最先拿出来的是 4 号，最后是 1 号。栈的模型就是这样的。队列这种数据结构 是先进先去，就像我们平时排队 你先去排队，就站在队列的最前面 如果进行出队操作，最先的元素就出队 没有了，无法有记忆作用。栈是相反 在前面的后出来，就具有了记忆作用。

34、test.c 文件中包括如下语句：

1	#define INT_PTR int*
2	typedef int*int_ptr;
3	INT_PTR a,b;
4	int_ptr c,d;

文件中定义四个变量，哪个变量不是指针类型？

- a
- **b**
- c
- d
- 都是指针
- 都不是指针

**解释：** #define 为宏定义指令 int\*为宏体是整型指针，INT\_PTR 为宏名；经过预处理，进行宏替换，INT\_PTR a,b;替换为 int\* a,b;故 a 是整型指针而 b 是整型。

typedef 作用是给已存在的数据类型引入一个别名，语法 typedef 已有类型名 类型别名，所以 int\_ptr 是 int\*整型指针类型的别名。

35、关于 do 循环体 while(条件表达式)，以下叙述中正确的是？

- 条件表达式的执行次数总是比循环体的执行次数多一次
- 循环体的执行次数总是比条件表达式执行次数多一次
- **条件表达式的执行次数与循环体的执行次数一样**
- 条件表达式的执行次数与循环体的执行次数无关

36、请找出下面代码中的所有错误。说明：以下代码是把一个字符串倒序，如“abcd”倒序后变为“dcba”。

1	1#include "string.h"
2	2intmain()

3	3{
4	4       char*src = "hello,world";
5	5       char*dest = NULL;
6	6       intlen = strlen(src);
7	7       dest = (char*)malloc(len);
8	8       char*d = dest;
9	9       char*s = src[len];
10	10      while(len-- != 0)
11	11          d++ = s--;
12	12      printf("%s", dest);
13	13      return0;
14	14
15	15}

- 第 7 行要为'\0' 分配一个空间
- 第 9 行改成 `char * s = &src[len-1]`
- 第 12 行前要加上 `*d = 0;`
- 第 13 行前要加上 `free(dest)` 释放空间

**解释：** `*d = 0` 与 `*d = '\0'` 效果一样， `char q[] = "hello";sizeof(q) == 6`

37、关于内联函数正确的是（）

- 类的私有成员函数不能作为内联函数
- 在所有类说明中内部定义的成员函数都是内联函数
- 类的保护成员函数不能作为内联函数
- 使用内联函数的地方会在运行阶段用内联函数体替换掉

**解释：** 感觉只有 B 对。A 是可以的，私有成员函数可以内联， C 也可以， D 应该是在编译阶段替换

38、下面有关析构函数和虚函数的说法，错误的是？

- 析构函数的作用是当对象生命期结束时释放对象所占用的资源
- 析构函数是特殊的类成员函数，它的名字和类名相同，没有返回值，没有参数不能随意调用，但是可以重载
- 使用虚函数，我们可以灵活的进行动态绑定

- 如果一个类可能做为基类使用的话，将其析构函数虚拟化， 这样当其子类的对象退出时， 也会一并调用子类的析构函数释放内存

**解释：**重载，就是函数或者方法有同样的名称，但是参数列表不相同的情形，这样的同名不同参数的函数或者方法之间，互相称之为重载函数或者方法。析构函数没有返回值，也没有参数，没有函数类型，因此不能被重载。**程序员可以自己重新定义一个析构函数，系统在调用完自己定义的析构函数后，还依然会调用默认析构函数。**

39、下列 for 循环的循环体执行次数为

1	for(int i=10, j=1; i=j=0; i++, j--)
---	-------------------------------------

- **0**
- 1
- 无限
- 以上都不对

**解释：**i=j=0 的返回值就是赋值结果，也就是 0，为假，不循环。如果改为 i=j=1 就是无限循环了

40、

- **纯虚函数的声明以 “=0;” 结束**
- **有纯虚函数的类叫抽象类，它不能用来定义对象**
- **抽象类的派生类如果不实现纯虚函数，它也是抽象类**
- **纯虚函数能有函数体**

**解释：**纯虚函数和抽象类：含有纯虚函数的类是抽象类，不能生成对象，只能派生。他派生的类的纯虚函数没有被改写，那么，它的派生类还是个抽象类。定义纯虚函数就是为了让基类不可实例化, 因为实例化这样的抽象数据结构本身并没有意义. 或者给出实现也没有意义如果一个类中至少有一个纯虚函数，那么这个类被称为抽象类（abstract class）。

**举例：**

```
class virtualClass
{
    virtual int func() = 0; //纯虚函数的
    virtual int func2(); //虚函数的
    int func3();
};
virtualClass c; //不能这样实例化。
```

- 纯虚函数能有函数体

```

class virtualClass
{
virtual int func() = 0
{
int a = 0;
}
virtual int func2();
int func3();
};

```

41、

1	struct Date
2	{
3	Char a;
4	Int b;
5	int64_t c;
6	char d;
7	};
8	Date data[2][10];

在 32 位系统上，如果 Date 的地址是 x，那么 data[1][5].c 的地址是 ( )

- X+195
- X+365
- **X+368**
- X+215

**解释：**结构体成员地址对齐

a b c d

$1 + (3) + 4 + 8 + 1 + (7) = 24$ ，( ) 内表示为了满足对齐填充的大小。

$\&\text{data}[1][5].c = x + 10 * 24 + 5 * 25 + 1 + (3) + 4 = 368$ 。

42、有如下模板定义：

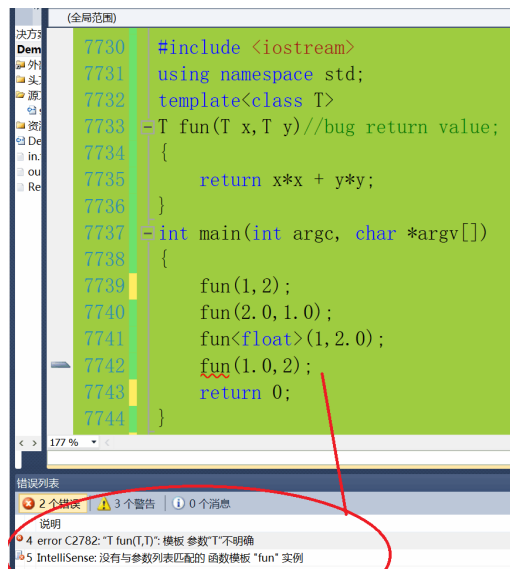
1	template <classT>
2	T fun(T x,T y) {
3	return x*x+y*y;
4	}

在下列对 fun 的调用中，错误的是 ( )

- fun(1, 2)
- **fun(1.0, 2)**
- fun(2.0, 1.0)
- fun<float>(1, 2.0)

**解释：**最下面看准了编译器的提示；

“T fun(T, T)”：模板参数“T”不明确



43、在 Windows 32 位操作系统中，假设字节对齐为 4，对于一个空的类 A，sizeof(A) 的值为 ( ) ？

- 0
- **1**
- 2
- 4

**解释：**为了确保不同的对象会拥有不同的地址，所以类的大小不会为 0 字节。空类的大小为 1 字节

44、

1	int main(void)
2	{
3	int i = 1;
4	int j = i++;
5	if((i++ > ++j) && (++i == j)) i += j;
6	printf("%d\n", i);

7	return 0;
8	}

请问最终输出的 i 值为 ( )

- 2
- **3**
- 4
- 5

**解释：**B。j = i ++ 以后 j 是 1 i 是 2。 (i++ > ++j) 这里 i 是 2 , ++j 是 2。所以 false , i ++ 执行完以后就是 3 了。 && 后面短路不执行

45、引用标准库，下面的说法哪些是正确的？

- 语句#include "stdlib. h" 是正确的，但会影响程序的执行速度
- **语句#include <stdio. h>是正确的，而且程序执行的速度比#include"stdio. h"要快**
- 语句#include "stdlib. h"和#include <stdio. h>都是正确的，程序执行速度没有区别
- 语句#include "stdlib. h"是错误的

**解释：**A 没有对比对象

46、若有以下程序

1	#include<stdio.h>
2	main()
3	{
4	int s=0,n;
5	for(n=0;n<4;n++)
6	{
7	switch(n)
8	{
9	default: s+=4;
10	case1: s+=1;break;
11	case2: s+=2;break;
12	case3: s+=3;

13	}
14	}
15	printf("%d\n",s);
16	}

则程序的输出结果是？

- 10
- 11
- 13
- 15

**解释：**break 语句的作用是终止正在执行的 switch 流程,跳出 switch 结构或者强制终止当前循环,从当前执行的循环中跳出。题干中第一次循环 n 值为 0,执行 default 语句后的 s+=4,s 的值变为 4,执行 case1 语句后的 s+=1,s 的值变为 5,遇到 break 语句跳出 switch 语句,进入第二次循环。第二次循环 n 的值为 1,执行 case1 后的 s+=1,s 的之变为 6,遇到 break 语句跳出 switch 语句,进入第三次循环。第三次循环时 n 的值为 2,执行 case2 后的 s+=2,s 的值变为 8,遇到 break 语句跳出 switch 语句,进入第四次循环。第四次循环时 n 的值为 3,执行 case3 后的 s+=3,s 的值变为 11。再判断循环条件为假,退出循环打印 s 的值 11。

47、如果 x=2014，下面函数的返回值是（）

1	intfun(unsigned  intx)
2	{
3	intn=0;
4	while((x+1))
5	{
6	n++;
7	x=x (x+1);
8	}
9	returnn;
10	}

- 20
- 21
- 23
- 25



**解释：** 返回值为：23

2014 对应的二进制为：0000 0000 000 0000 0000 0111 1101 1110

而  $x|(x+1)$  的作用是对一个数中二进制 0 的个数进行统计

例如本题：

第一次循环：0000 0000 000 0000 0000 0111 1101 1110 | 0000 0000 000 0000 0000  
0111 1101 1111 = 0000 0000 000 0000 0000 0111 1101 1111

第二次循环：0000 0000 000 0000 0000 0111 1101 1111 | 0000 0000 000 0000 0000  
0111 1110 0000 = 0000 0000 000 0000 0000 0111 1111 1111

.

每循环一次 0 的数量减一，直到溢出

所以 2014 二进制中一共有 23 个 0 则返回值为 23

## 1、输出结果是 14

```
1. #include <iostream>
2. using namespace std;
3.
4. #define max(a, b) (a) > (b) ? (a) : (b)
5.
6. int main(int argc, char *argv[])
7. {
8.     int x = 12;
9.     int y = 10;
10.    int i = max(++x, y);
11.    cout<<i<<endl;
12.    system("pause");
13. }
```

## 2、static、const 之实现原理

## 3、关于非空二叉树的性质，下面哪个结论不正确(D)

A、有两个节点的节点一定比没有子节点的节点少一个  $n_0 = n_2 + 1$

B、根节点所在的层数为第 0 层，则第 i 层最多有  $2^i$  个节点

C、若知道二叉树的前序遍历序列和中序遍历序列，则一定可以推出后序遍历序列。

D、堆一定是一个完全二叉树

### 3、各种排序算法的时间复杂度，空间复杂度

4、

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. int main(int argc, char *argv[])
6. {
7.     string name1 = "youku";
8.     const char* name2 = "youku";
9.     char name3[] = {'y','o','u','k','u'};
10.    char name4[] = "youku";
11.    size_t l1 = name1.size(); //5
12.    size_t l2 = strlen(name2); //5
13.    size_t l3 = sizeof(name2); //4
14.    size_t l4 = sizeof(name3); //5
15.    size_t l5 = strlen(name3); //未知
16.    size_t l6 = sizeof(name4); //6
17.    size_t l7 = strlen(name4); //5
18.    cout<<name1<<endl; //youku
19.    cout<<name2<<endl; //youku
20.    cout<<name3<<endl; //youku 乱码。。。
21.    cout<<name4<<endl; //youku
22.    system("pause");
23. }
```

### 5、树的深度是 k,则二叉树最多有 $2^k-1$ 个节点。

6、

```
int a[5]={1,2,3,4,5};
int *p = a;
```

**\*p++** 先取指针 p 指向的值（数组第一个元素 1），再将指针 p 自增 1；

```
cout << *p++; // 结果为 1
cout <<(*p++); // 1
```

**(\*p)++** 先去指针 p 指向的值（数组第一个元素 1），再将该值自增 1（数组第一个元素变为 2）

```
cout << (*p)++; // 1
cout <<((*p)++) //2
```

**\*++p** 先将指针 p 自增 1（此时指向数组第二个元素），\* 操作再取出该值

```
cout << *++p; // 2
cout <<(*++p) //2
```

**++\*p** 先取指针 p 指向的值（数组第一个元素 1），再将该值自增 1（数组第一个元素变为 2）

```
cout <<++*p; // 2
cout <<(++*p) //2
```

注意，上面的每条 cout 输出，要单独输出才能得到后面的结果。

7、先看一个空的类占多少空间？

```
1. class Base
2. {
3. public:
4.     Base();
5.     ~Base();
6.
7. };
```

**解释：**注意到我这里显示声明了构造跟析构，但是 `sizeof(Base)` 的结果是 1. 因为一个空类也要实例化，所谓类的实例化就是在内存中分配一块地址，每个实例在内存中都有独一无二的地址。同样空类也会被实例化，所以编译器会给空类隐含的添加一个字节，这样空类实例化之后就有了独一无二的地址了。所以空类的 `sizeof` 为 1。

8、接着看下面一段代码

```
1. class Base
2. {
3. public:
4.     Base();
5.     virtual ~Base();           //每个实例都有虚函数表
6.     void set_num(int num)      //普通成员函数，为各实例公有，不归入
    sizeof 统计
7.     {
```

```

8.         a=num;
9.     }
10.private:
11.     int a;           //占 4 字节
12.     char *p;         //4 字节指针
13.};
14.
15.class Derive:public Base
16.{
17.public:
18.     Derive():Base(){};
19.     ~Derive(){};
20.private:
21.     static int st;    //非实例独占
22.     int d;           //占 4 字节
23.     char *p;         //4 字节指针
24.
25.};
26.
27.int main()
28.{
29.     cout<<sizeof(Base)<<endl;
30.     cout<<sizeof(Derive)<<endl;
31.     return 0;
32.}

```

结果自然是

12

20

Base 类里的 `int a;char *p;` 占 8 个字节。

而虚析构函数 `virtual ~Base();` 的指针占 4 子字节。

其他成员函数不归入 `sizeof` 统计。

Derive 类首先要具有 Base 类的部分，也就是占 12 字节。

`int d;char *p;` 占 8 字节

`static int st;` 不归入 `sizeof` 统计

所以一共是 20 字节。

**无论有几个虚函数，sizeof 类都等于 sizeof(数据成员)的和+sizeof(V 表指针，为 4)**

9、对于虚继承的子类，其 `sizeof` 的值是其父类成员，加上它自己的成员，以及它自己一个指向父类的指针（大小为 4, 指针，指向一个关于偏移量的数组，且称之为虚基类偏移量表指针），对齐后的 `sizeof`

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. class a
6. {
7. private:
8.     int x;
9. };
10.
11. class b: virtual public a
12. {
13. private:
14.     int y;
15. };
16.
17. class c: virtual public a
18. {
19. private:
20.     int z;
21. };
22.
23. class d: public b, public c
24. {
25. private:
26.     int m;
27. };
28. int main(int argc, char* argv[])
29. {
30.     cout<<sizeof(a)<<endl; //4
31.     cout<<sizeof(b)<<endl; //12
32.     cout<<sizeof(c)<<endl; //12
33.     cout<<sizeof(d)<<endl; //24
34.     return 0;
35. }
```

```
14 30 42 00 //Class b 的虚基类偏移量表指针
02 00 00 00 //y
C4 2F 42 00 //Class c 的虚基类偏移量表指针
03 00 00 00 //z
04 00 00 00 //m
01 00 00 00 //x
```

## 10、不能建立数组的引用

## 11、

(1)、全局变量是不显示用 **static** 修饰的全局变量，但全局变量默认是静态的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 **extern** 全局变量名的声明，就可以使用全局变量。

(2)、全局静态变量是显示用 **static** 修饰的全局变量，作用域是所在的文件，其他的文件即使用 **extern** 声明也不能使用。一 (3) 内存基本构成：

可编程内存存在基本上分为这样的几大部分：静态存储区、堆区和栈区。

静态存储区：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都存在。它主要存放静态数据、全局数据和常量。

栈区：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。

堆区：亦称动态内存分配。程序在运行的时候用 **malloc** 或 **new** 申请任意大小的内存，程序员自己负责在适当的时候用 **free** 或 **delete** 释放内存。动态内存的生存期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉动态内存。但是，良好的编程习惯是：如果某动态内存不再使用，需要将其释放掉，否则，我们认为发生了内存泄漏现象。

12、switch 语句后的控制表达式只能是 short、char、int、long 整数类型和枚举类型，不能是 float，double 和 boolean 类型

## 13、

### 1.重载

在同一个类中，或者在顶层函数（与 main 函数同层）中，如果

(1) 函数名字相同

(2) 但是函数签名不同

就是函数重载。所谓函数签名，就是函数的参数列表（包括参数类型、个数、出现顺序）

在编译期间生成的标识。注意，返回值不是函数签名的部分。

下面代码中，两个 function 函数就是重载函数

```
class ClassA {
public:
    void function();
    void function(int);
};
```

再如返回值不同，这样是编译不过去的。因为返回值不是函数签名的一部分，所以两个函

数签名相同，自然也不是函数重载了

```
class ClassA {
public:
    void function();
    int function();
};
```

## 2. 覆盖

函数覆盖发生在继承层次之中。覆盖必须满足下面 4 个条件

- （1）发生覆盖的函数必须分别在父类和子类中
- （2）子类的成员函数的函数名与父类相同
- （3）子类参数个数和类型与父类相同
- （4）父类函数必须是虚函数

下面就是函数覆盖的一个例子，ClassA 的指针指向 ClassB 类的对象，调用的是 ClassB 的

函数

```
class ClassA {
public:
    virtual void function();
};
class ClassB : public ClassA {
```

```

public:
    void function();
};
int main() {
    ClassA* pa = new ClassB;
    pa->function();
}

```

如果上面没有使用 virtual 关键字，那么调用的将是 ClassA 函数

可见，函数覆盖可以实现多态调用

3.隐藏（遮蔽）：指子类中具有与父类同名的函数（不管参数列表是否相同），除了函数覆盖的情况称为隐藏

（1）子类的成员函数与父类的这个非虚成员函数有不同的函数签名

（2）子类的虚函数无法覆盖父类的虚函数

就是说，继承层次中，父类与子类的同名函数要么是覆盖要么是隐藏了

第一个例子：

上面的例子中少了 virtual 关键字的情况，这时候我们说 ClassB::function 遮蔽了继承而来的

ClassA::function。

第二个例子：父类的 function 有一个 int 参数，而子类没有参数，是函数隐藏

```

class ClassA {
public:
    void function(int);
};
class ClassB : public ClassA {
public:
    void function();
};

```

第三个例子：父类是虚函数，但子类没有与其相同的函数签名

```

class ClassA {
public:
    virtual void function(int);
};

```



```
};
class ClassB : public ClassA {
public:
    void function();
};
```

总结：

- (1) 函数重载发生在同一个类或顶层函数中，同名的函数而具有不同的参数列表
- (2) 函数覆盖发生在继承层次中，该函数在父类中必须是 virtual，而子类的该函数必须与父类有相同的参数列表
- (3) 函数隐藏（遮蔽）发生在继承层次中，父类和子类同名的函数中，不属于函数覆盖的都属于函数隐藏

14、如果一个变量被频繁使用，需保存在寄存器中，因为寄存器的速度要比内存快的许多。在早期的编译器中需要手动定义为 register 型，但是后来编译器可以自动将调用次数多的变量放入寄存器中。

auto: 给变量动态分配内存，默认的分配类型。一般不需要手动声明了；

static：静态分配内存。变量在整个作用域内是全局变量。

extern：声明为外部变量；在函数的外部定义变量；

15、&代表取存储地址

\*表示取地址存储的值

a[1][1]的位置如下

a[0][0] a[0][1] a[0][2] a[0][3]

a[1][0] a[1][1] .....

.....

二维数组也是线性存储的

A D:代表取 `a[0][0]`的地址，在向后推 5 位，取那个地址的值，显然是正确的（上面排列）

B:数组名称就是代表指向数组首地址的指针，所以`*(a+1)`指 `a[0][1]`的值，+1 再取这个值所代表的地址的值，这样是取不到 `a[1][1]`的。甚至产生内存越位。

C:`&a[1]`代表存储 `a[1][0]`的地址，+1 所以下一位地址就是存储 `a[1][1]`的地址，\*取值所以可以取到了 `a[1][1]`的值。

## 16、字符串常量

（1）C 语言中，为什么字符串可以赋值给字符指针变量

```
char *p,a='5';
```

```
p=&a;           //显然是正确的，
```

```
p="abcd";       //但为什么也可以这样赋值？？
```

双引号做了 3 件事：

- 1.申请了空间(在常量区)，存放了字符串
- 2.在字符串尾加上了`'\0'`
- 3.返回地址

你这里就是 返回的地址，赋值给了 p

（2）"abcd"是常量吗？答案是有时是，有时不是。

不是常量的情况："abc"作为字符数组初始值的时候就不是，如 `char str[] = "abc"`; 因为定义的是一个字符数组，所以就相当于定义了一些空间来存放"abc"，而又因为字符数组就是把字符一个一个地存放的，所以编译器 把这个语句解析为 `char str[3] = {'a','b','c'}`;又根据上面的总结 1，所以 `char str[] = "abc"`;的最终结果是 `char str[4] = {'a','b','c','\0'}`;

做一下扩展，如果 `char str[] = "abc";`是在函数内部写的话，那么这里的`"abc\0"`因为不是常量，所以应该被放在栈上。

是常量的情况:把`"abc"`赋给一个字符指针变量时，如 `char* ptr = "abc";`因为定义的是一个普通字符指针，并没有定义空间来存放`"abc"`，所以编译器得帮我们 找地方来放`"abc"`，显然，把这里的`"abc"`当成常量并把它放到程序的常量区是编译器最合适的选择。所以尽管 `ptr` 的类型不是 `const char*`，并且 `ptr[0] = 'x';`也能编译 通过，但是执行 `ptr[0] = 'x';`就会发生运行时异常，因为这个语句试图去修改程序常量区中的东西。记得哪本书中曾经说过 `char* ptr = "abc";`这种写法原来在 `c++`标准中是不允许的，但是因为这种写法在 `c` 中实在是太多了，为了兼容 `c`，不允许也得允许。虽然允许，但是建议的写法应该是 `const char* ptr = "abc";`这样如果后面写 `ptr[0] = 'x';`的话编译器就不会让它编译通过，也就避免了上面说的运行时异常。又扩展一下，如果 `char* ptr = "abc";`写在函数体内，那么虽然这里的`"abc\0"`被放在常量区中，但是 `ptr` 本身只是一个普通的指针变量，所以 `ptr` 是被放在栈上的，只不过是它所指向的东西被放在常量区罢了。

(3) 对于 `char str[] = "abcdef";`就有 `sizeof(str) == 7`,因为 `str` 的类型是 `char[7]`，也有 `sizeof("abcdef") == 7`，因为`"abcdef"`的类型是 `const char[7]`。

对于 `char *ptr = "abcdef";`就有 `sizeof(ptr) == 4`，因为 `ptr` 的类型是 `char*`。

对于 `char str2[10] = "abcdef";`就有 `sizeof(str2) == 10`，因为 `str2` 的类型是 `char[10]`。

对于 `void func(char sa[100],int ia[20],char *p);`

就有 `sizeof(sa) == sizeof(ia) == sizeof(p) == 4`，

因为 `sa` 的类型是 `char*`，`ia` 的类型是 `int*`，`p` 的类型是 `char*`。

## 17、C++函数中那些不可以被声明为虚函数的函数

常见的不能声明为虚函数的有：普通函数（非成员函数）；静态成员函数；内联成员函数；构造函数；友元函数。

### 1、为什么 C++不支持普通函数为虚函数？

普通函数（非成员函数）只能被 overload，不能被 override，声明为虚函数也没有什么意思，因此编译器会在编译时绑定函数。

### 2、为什么 C++不支持构造函数为虚函数？

这个原因很简单，主要是从语义上考虑，所以不支持。因为构造函数本来就是为了明确初始化对象成员才产生的，然而 virtual function 主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual 函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用 virtual 函数来完成你想完成的动作。（这不就是典型的悖论）

### 3、为什么 C++不支持内联成员函数为虚函数？

其实很简单，那内联函数就是为了在代码中直接展开，减少函数调用花费的代价，虚函数是为了在继承后对象能够准确的执行自己的动作，这是不可能统一的。（再说了，inline 函数在编译时被展开，虚函数在运行时才能动态的绑定函数）

### 4、为什么 C++不支持静态成员函数为虚函数？

这也很简单，静态成员函数对于每个类来说只有一份代码，所有的对象都共享这一份代码，他也没有要动态绑定的必要性。

### 5、为什么 C++不支持友元函数为虚函数？

因为 C++不支持友元函数的继承，对于没有继承特性的函数没有虚函数的说法。

\*\*\*\*\*

1、**顶层函数**：多态的运行期行为体现在虚函数上，虚函数通过继承方式来体现出多态作用，顶层函数不属于成员函数，是不能被继承的。

2、**构造函数**：（1）构造函数不能被继承，因而不能声明为 virtual 函数。

（2）构造函数一般是用来初始化对象，只有在一个对象生成之后，才能发挥多态的作用，如果将构造函数声明为 virtual 函数，则表现为在对象还没有生成的情况下就使用了多态机制，因而是行不通的，如下例：

[cpp] [view plaincopy](#)

```
1. #include <iostream>
2. using namespace std;
3.
4. class B
5. {
6. public:
7.     B() {}
8.     virtual void show()
9.     {
10.         cout<<"***"<<endl;
11.     }
12.};
13.class D:public B
14.{
15.public:
16.    D() {}
17.    void show()
18.    {
19.        cout<<"==="<<endl;
20.    }
21.};
22.
23.int main(void)
24.{
25.    B *pb;
26.    D d;        //先生成对象
27.    pb=&d;
28.    pb->show(); //再体现多态
```

29.

```
30.    pb=new D(); //先调用构造函数
```

```
31.    pb->show(); //再多态
```

```
32.    delete pb;
```

```
33.    return 0;
```

```
34.}
```

3、static 函数：不能被继承，只属于该类。

4、友元函数：友元函数不属于类的成员函数，不能被继承。

5、inline 函数：inline 函数和 virtual 函数有着本质的区别，inline 函数是在程序被编译时就展开，在函数调用处用整个函数体去替换，而 virtual 函数是在运行期才能够确定如何去调用的，因而 inline 函数体现的是一种编译期机制，virtual 函数体现的是一种运行期机制。

此外，一切 virtual 函数都不可能是 inline 函数。

## 18、C++抽象类

### 一、纯虚函数定义.

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”

### 二、引入原因：

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual Return Type Function()= 0;），则编译器要求在派生类中必须予以重载以实现多态

性。**同时含有纯虚拟函数的类称为抽象类，它不能生成对象。**这样就很好地解决了上述两个问题。

### 3、抽象类的规定

(1) 抽象类只能用作其他类的基类，不能建立抽象类对象。

(2) 抽象类不能用作参数类型、函数返回类型或显式转换的类型。

(3) 可以定义指向抽象类的指针和引用，此指针可以指向它的派生类，进而实现多态性。

## 三、相似概念：

### 1、多态性

指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。

a.编译时多态性：通过函数重载和运算符重载来实现的。

b 运行时多态性：通过继承和虚函数来实现的。

### 2、虚函数

虚函数是在基类中被声明为 virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载

**纯虚函数的声明有着特殊的语法格式：virtual 返回值类型成员函数名（参数表）**

**=0；**

**请注意，纯虚函数应该只有声明，没有具体的定义，即使给出了纯虚函数的定义也会被编译器忽略。**

### 3、抽象类

包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。在 C++ 中，我们可以把只能用于被继承而不能直接创建对象的类设置为抽象类 ( Abstract Class )。之所以要存在抽象类，最主要是因为它具有不确定因素。我们把那些类中的确存在，但是在父类中无法确定具体实现的成员函数称为纯虚函数。纯虚函数是一种特殊的虚函数，它只有声明，没有具体的定义。抽象类中至少存在一个纯虚函数；存在纯虚函数的类一定是抽象类。存在纯虚函数是成为抽象类的充要条件。

程序举例：

```
1//基类: 2class A
3{
4public:
5    A();
6void f1();
7virtualvoid f2();
8virtualvoid f3()=0;
9virtual ~A();
10};
112//子类:13class B : public A
14{
15public:
16    B();
17void f1();
18void f2();
19void f3();
20virtual ~B();
21};
2223//主函数:24int main(int argc, char* argv[])
25{
26    A *m_j=new B();
27    m_j->f1();
28    m_j->f2();
29    m_j->f3();
30    delete m_j;
31return0;
32}
33/*34f1()是一个普通的重载.
```



35 调用 `m_j->f1()`; 会去调用 A 类中的 `f1()`, 它是在我们写好代码的时候就会定好的. 因为 `f1()` 不是虚函数, 不会动态绑定

36 也就是根据它是由 A 类定义的, 这样就调用这个类的函数.

37 `f2()` 是虚函数.

38 调用 `m_j->f2()`; 会调用 `m_j` 中保存的对象中, 对应的这个函数. 这是由于 `new` 的 B 对象.

39 `f3()` 与 `f2()` 一样, 只是在基类中不需要写函数实现.

40\*/

## 19、输出 16 进制和 10 进制数字

```
1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. int main()
5. {
6.     int a = 1;
7.     int b = 2;
8.     cout<< "a = " << hex << setfill('0') << setw(16) << a << dec
    << " b = " << b << endl;
9.     system("pause");
10. }
```

## 20、默认参数设置

如果某个参数是默认参数, 那么它后面的参数必须都是默认参数

下面两种情况都可以

```
void Func(int i, float f = 2.0f, double d = 3.0) ;
void Func(int i, float f, double d = 3.0) ;
```

但是这样就不可以

```
void Func(int i, float f = 2.0f, double d) ;
```

默认参数的连续性能保证编译器正确的匹配参数。所以可以下这样一个结论, 如果一个函数含有默认参数, 那么它的最后一个参数一定是默认参数。

默认参数可以放在函数声明或者定义中, 但只能放在二者之一

通常我们都将默认参数放在函数声明中, 因为如果放在函数定义中, 那么将只能在函数定义所在地文件中调用该函数。(为什么呢?)

.h 文件

```
class TestClass
```

```
{
public:
    void Func(int i, float f, double d) ;
};
```


.cpp 文件

```
#include "TestClass.h" void TestClass::Func(int i = 1, float f = 2.0f, double d
= 3.0)
{
    cout << i << ", " << f << ", " << d << endl ;
}
```

像上面这样，只能够在 TestClass.cpp 中调用 Func 函数。岂不是很痛苦？


### 函数重载时谨慎使用默认参数值

比如下面两个重载函数 func，一个只接受一个参数，而另一个接受两个参数，包括一个默认参数。

```

class Test
{
public:
    int func(int a)
    {
        return a;
    }

    int func(int a, int b = 1)
    {
        return a + b;
    }
};
```

如果像下面这样调用函数

```

int main(void)
{
    Test test;
    int result = test.func(1);

    getchar() ;
    return 0 ;
}
```

则编译器就不知道选择哪个函数，这就造成了混淆。

## 21、字符串指针不允许更改字符串常量

```
1. int main()
2. {
3.     char p[] = "Hello";
4.     char *q = "Hello";
5.     cout<<p<<endl;
6.     p[0] = 'A';           //可以
7.     cout<<p<<endl;       //输出"Aello"
8.     cout<<q<<endl;
9.     q[0] = 'A';           //失败
10.    cout<<q<<endl;
11.    system("pause");
12.}
```