【C++】1. 请答出 const 与#define 相比，有何优点？

答：**const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。**（有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。）

【C++】2. C++语言内存的分配方式有几种？

答：（1）**从静态存储区域分配。**内存在程序编译的时候已经分配，这部分内存在程序的整个运行期间都存在。例如全局变量、静态变量。

（2）**在栈上创建。**在执行函数时，函数内局部变量的存储单元可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率高，但分配的内存容量有限。

（3）**从堆上分配，亦称动态内存分配。**程序在运行的时候用 new 或 malloc 申请任意多少的内存，程序员自己负责在何时用 delete 或 free 释放内存。动态内存的生存期由程序员决定，使用非常灵活，但问题也最多。

【static】3. static 全局变量与普通全局变量有什么区别？static 局部变量和普通局部变量有什么区别？

答：全局变量（外部变量）的说明之前再冠以 static 就构成静态的全局变量。**全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。**这两者在存储方式上并无不同。这两者的区别存在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

因此，**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。把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。**

static 全局变量与普通的全局变量的区别：static 全局变量只初始化一次，防止在其他文件单元中被引用。

static 局部变量与普通局部变量的区别：static 局部变量只被初始化一次，下一次依据上一次结果值。

【变量】4. 变量的存储类包括四类：

（1）**自动变量**：只有在函数内或复合语句内定义的局部变量，才能被定义为自动变量。程序中大多数变量属于自动变量，其主要特点是**临时性**。未给定初值时，它的**初值是不确定的**。

（2）**静态变量**：在程序**编译**时预分配，并在程序**执行之前就被确定存储单元**。定义静态的局部变量时，未指定初值，则系统自动给静态的局部变量赋一个**二进位信息全为 0 的初值**。

（3）**外部变量**：一个在函数之外某处定义的**全局变量**，它或在源程序文件的后面定义，或在别的源程序文件中定义。

（4）**寄存器变量**：一般情况下，变量的值是存放在内存中的。当程序中用到哪一个变量的值时，需要将内存中该变量的值送到 CPU 中的运算器。经过运算器运算之后，如果需要存数，再从运算器中将数据送到内存存放。如果程序中有一些变量使用频繁，则存取变量的值要花不少时间。为了提高执行效率，C++允许将局部变量的值放在 CPU 中的寄存器中，需要用时直接从寄存器取出，不必再到内存中去存取。这种**放在 CPU 的寄存器中的变量叫做寄存器变量。只有 int 型、char 型及指针类型的局部变量和形参才可以是寄存器变量。全局**

变量或其他复杂数据类型的变量都不可以是寄存器变量。（由于离开函数或分程序后，值就消失）将一个局部变量指定为寄存器变量时，是提醒编译程序，这个变量在程序中使用得十分**频繁**，在为该变量分配存储空间时，有可能的话，尽量为它分配寄存器，因为**访问寄存器要比访问存储单元来得快**。将一个形参的存储类指定为寄存器时，可能是因为要访问某些特殊设备的驱动程序，这些设备的驱动程序要求以寄存器为参数与系统进行信息传递。

【内联函数】5. 内联函数：在**编译**时将函数体的**代码**直接插入到函数调用处，将调用函数的方式改为顺序执行直接插入的程序代码，这样就减少了程序执行时间，这个过程称为函数的内联。这种嵌入到主调函数中的函数称为内联函数。

在程序执行过程中，当调用一个函数时，需要保存现场数据和返回地址（以便在函数调用之后继续执行），然后转到被调用函数代码的起始地址去执行。被调用函数执行完以后，又要取出先前保存的现场数据和返回地址，转回到主调函数继续执行。这些操作都要花费一定的时间，使用**内联函数可以节省参数传递、控制转移等开销**，但目标文件的存储空间却增大了。**内联函数实质上是以空间来换取时间的**，因此一般只将规模很小（一般为 5 个语句以下）而使用频繁的函数声明为内联函数。

【宏】6. 设宏定义#define P(x) x/x，则执行语句 cout<<P(4+6)<<endl;后的输出结果是（C）

A. 1　　B.8.5　　C.11　　D.11.5

评：易错选 D，注意不能在程序中坚持惯性的数学思维，6/4 的结果是由数据类型决定的。另外：最易犯的编程错误就是 int/int 的结果算成浮点型的（即不能整除时，把小数点以后的结果也算进去了。）

【函数重载】7. 函数**重载**是指一个函数可以和同一作用域中的其他函数具有相同的名字，但这些**同名函数的形参表必须互不相同**（形参表不同是指参数的个数不同或参数的个数相同但其类型不同）。使用函数重载时需要注意：**一个函数不能既作为重载函数，又作为有默认参数的函数。**

```
int myMax(int x,int y);
int myMax(int x,int y,int z=100);//myMax 是重载函数，又有默认参数
```

如调用“myMax(1,2)”,编译系统无法判定如何调用，出现二义性，系统无法执行。

【指针】8. （1）定义指针数组 p,它有 10 个指向整型数据的指针元素：int * p[10];

（2）定义指向含 10 个元素的一维数组的指针变量 p：int (*p)[10];

（3）p 为返回一个指针的函数，该指针指向整形数据：int * p();

（4）p 为指向函数的指针，该函数返回一个整数值：int (*p)();

【数组】9. 若有如下定义：int a[10];则 **a** 是数组名，它是**数组首地址常量**，a++不合法。

【指针，数组，C 语言】10. 设 char ch[]={“abc\0def”},*p=ch;则执行下列语句的输出为：

```
cout<<*(p+4)<<endl;//  d
cout<<p<<endl;//  abc
cout<<p+3<<endl;//      什么都不输出，cout<<'\0';会输出一个空格。
cout<<p+4<<endl;//      def
```

评：cout 输出时遇到'\0'就会停止输出，而且并不将'\0'显式输出，若显式输出'\0'会得到一个空格（空值但占一个字符位置）。当用 cout 输出一个地址时，cout 会将从该地址开始一直

到'\0'的地方之间的内容全部输出。
补充：在 C 语言中 printf("%s",串首地址)用来输出字符串，printf("%c",字符)用来输出字符，如果用 printf("%c",字符地址)则会将字符地址对应的 ASCII 码变成字符输出。

【指针，数组】***11.** 若有如下定义：char *aa[2]={"abcd","ABCD"};则可以这样理解：*aa 存放的是指针数组的首地址，aa 中存放的是地址的地址，故输出 aa 会显示一段地址。具体如下：

```
cout<<aa<<endl;//  地址
cout<<*aa<<endl;//      abcd
cout<<aa[1]<<endl;//    ABCD
cout<<*aa+2<<endl;//    cd
cout<<aa[1]+2<<endl;// CD
cout<<*(aa[1]+2)<<endl;//   C
cout<<*aa[0]+2<<endl;//     99
cout<<*aa[0]<<endl;//   a
cout<<*aa[1]<<endl;//   A
```

补充：**常用 ASCII 码值：0---48，A---65，a---97**

【const，指针】12. 分为以下三类：(谁是常量谁就不能被修改)
（1）**指向常量的指针**：const char *name = "liu";
（2）**常指针**：char s[]="abcd"; char *const p=s;
（3）**指向常量的常指针**：const char * const name="liu";

【引用，指针】13. C++语言中引用与指针有哪些区别？
答：（1）指针是指向对象或变量的地址，指针通过某个指针变量指向一个对象后，对所指向的变量间接操作。引用实际上是所引用的对象或变量的别名，对引用的操作就是对目标变量的操作。
（2）引用在定义时必须初始化，而指针在定义时不必初始化。
（3）不存在指向空值（void）的引用，但是存在指向空值（NULL）的指针。
（4）引用在初始化后不能改变引用关系，而指针可以随时改变所指的对象（非 const 指针）
（5）引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。

【静态成员函数】14. 当成员函数不访问非 static 类数据成员时，才声明为 static。静态成员函数没有 this 指针，因此无法对一个对象中的非静态数据成员进行直接访问。

【构造函数】15. **成员对象的构造函数的执行顺序仅与成员对象在类中声明的顺序有关**，而与成员初始化列表中给出的成员对象的顺序无关。

【拷贝构造函数】16. 拷贝构造函数的作用是，用一个已存在的对象去初始化一个**新**的同类对象，也可以说是用一个已知的对象去**创建**另一个同类对象。调用时机如下：
（1）当定义对象时，用一个对象去初始化该类的另一个对象。
（2）如果函数的形参是类的对象，调用函数时使用的是值传递方式进行形参和实参的结合。
（3）如果函数的返回值是类的对象，函数调用完成时，可调用拷贝构造函数。
注意：**将类的一个对象赋予该类的另一个对象时，不会调用拷贝构造函数。**

【析构函数】17. **析构函数**和构造函数一样，也是类的一个**公有成员函数，没有返回值**类型说明，也不能被指定为 void 类型。和构造函数不同的是，析构函数**不接收任何参数**，但**可以是虚函数**。由于析构函数没有函数参数，因此它**不能被重载**。一个类只有一个析构函数。

【构造函数】18. 假定 MyClass 为一个类，则执行 MyClass a,b(2),*p;语句时，自动调用该类的构造函数的次数为（A）。

A.2　　B.3　　C.4　　D.5

评：定义一个指向类对象的指针变量时，并不创建一个对象，因此，定义指向类对象的指针变量时，并不会调用构造函数（包括默认构造函数）进行初始化。本题如果改为 MyClass a,b(2),*p=new MyClass;则结果为调用构造函数 3 次。

【静态数据成员】19. 静态数据成员的初始化与该类的构造函数无关，不能在构造函数中对它进行初始化。静态数据成员初始化：int Class::static_data_member = 0;(已有如下定义：static int static_data_member;)，另外，初始化只能在类体外进行。

补充：**在类体中不允许对数据成员进行初始化。**

【常数据成员】20. 只能通过成员初始化列表的方式来生成构造函数对该数据成员初始化。如下：Class(int i):x(i) {}（Class 为类名，已有如下定义：const int x;）。

【常成员函数】21. 只有常成员函数才有资格操作常对象。常对象只能调用常成员函数。

【构造函数】22. 当程序执行到对象定义时，调用自动局部对象的构造函数。该对象的析构函数在对象离开范围时调用（即离开定义对象的块时，这一点很重要，**假如一个对象 obj 在外部函数中被定义，而该函数又在 main 函数中被调用，那么当函数被调用时，创建 obj 对象，当函数调用结束，返回 main 函数时，则该对象已离开其作用域，此时即被删除**）。自动对象的构造函数与析构函数在每次对象进入和离开范围时调用。

假设生成派生类对象，基类和派生类都包含其他类的对象，则在建立派生类的对象时，首先执行基类成员对象的构造函数，接着执行基类的构造函数，然后执行派生类的成员对象的构造函数，最后才执行派生类的构造函数。析构函数的调用次序与调用构造函数的次序相反。

【构造函数，拷贝构造函数】***23.*** 分析程序写结果：

```
#include<iostream>
using namespace std;
class B{
public:
    B(){cout<<"default constructor is called"<<endl;}
    B(int i):data(i){cout<<"constructor by parameter "<<data<<" is called"<<endl;}
    B(B &b){cout<<"copy constructor is called"<<endl;}
    ~B(){cout<<"destructor is called"<<endl;}
private:
    int data;
};
```

```
B Play(B b)
{
	cout<<"function Play() now is running"<<endl;
	return b;
}
void local_fuc()
{
	B b;
}
int main()
{
	local_fuc();
	B temp = Play(5);
	return 0;
}
```

运行结果如下：

```
default constructor is called
destructor is called
constructor by parameter 5 is called
function Play() now is running
copy constructor is called
destructor is called
destructor is called
```

分析：调用函数 local_fuc();时，创建局部对象，当调用结束返回 main 函数时，该对象立即被删除。B temp = Play(5);该句传参数进入 Play()函数。看函数原型，参数为对象。此时最大的问题就是会否调用拷贝构造函数。应该明确：拷贝构造函数调用的原因是有一个新对象，而该新对象与已经存在的一个对象有了关系。此时，一个对象都没有，牵扯不到拷贝构造函数，所以，首先调用构造函数，而当函数返回一个对象并将其赋值给一个新对象时，才会调用拷贝构造函数。另外，题目中涉及到了拷贝构造函数调用的三个时机，但应该只调用一次，究其原因，应该再次考虑拷贝构造函数存在的目的，它是为了复制一个临时对象，如果只看到拷贝构造函数调用的时机来了，就去调用，岂不是复制出很多临时对象？但只要一个就够了，编译器何必浪费内存空间呢？

【this 指针，内存，对象】24. **每个对象所占用的存储空间只是该对象的数据部分所占用的存储空间，而不包括成员函数代码所占用的存储空间。**类定义中声明的数据成员（静态数据成员除外）对该类的每个对象都有一个备份，而类中的成员函数对该类的所有对象只有一个备份。

【protected】25. 保护成员允许该类的成员函数存取保护成员数据或调用保护成员函数，也允许该类的派生类的成员函数存取保护成员数据或调用保护成员函数，但其他函数不能存取该类的保护成员数据，也不能调用该类的保护成员函数。这意味着在类的**访问权限上它与 private 非常相似，即类的对象是不能直接访问保护成员的**。但保护成员在不同条件下，分

别具有公有成员或私有成员的特性。（基类自身的成员可以访问基类中任何一个成员，但是通过**基类的对象就只能访问其公有成员**。）

对于单个类来说，私有成员和保护成员没有什么区别。但对于继承来说，保护成员与私有成员则不同，保护成员可以被派生类的成员（注意，不是对象）访问，而私有成员不可以被派生类的成员访问。保护成员和公有成员又有所不同，保护成员即使在公有继承的情况下也不能被派生类的对象访问，而公有成员可以在公有继承的情况下，被派生类的对象访问。

【多继承，二义性】26. 基类和派生类中同时出现的同名函数，不存在二义性。因为如果不使用限定法，则派生类默认访问派生类函数。

【虚基类】27. **虚基类**是这样一个类：它虽然被一个派生类间接地多次继承，但派生类却只继承一份该基类的成员，这样就**避免了在派生类中访问这些成员时产生二义性**。

【虚基类，构造函数】***28.** 一般派生类不需要为间接基类提供构造函数的初始化值，它们只与直接基类打交道。对于非虚基类，在派生类的构造函数中初始化间接基类是不允许的，而对于虚基类，则必须在派生类中对虚基类初始化。如果在派生类的构造函数的成员初始化列表中没有列出对虚基类构造函数的调用，则表示使用该虚基类的默认构造函数。

我们将建立对象时所指定的类称为最派生类。建立一个对象时，如果这个对象中含有从虚基类继承来的成员，则虚基类的成员是由最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，而且，只有最派生类的构造函数会调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用都被自动忽略，从而保证了虚基类的构造函数只被调用一次，从虚基类中继承来的数据成员只被初始化一次。

当在一个成员初始化列表中同时出现对虚基类和非虚基类构造函数的调用时，**虚基类的构造函数先于非虚基类的构造函数执行。**

例子程序：

```
#include<iostream>
using namespace std;
class Base{
public:
    Base(char i){cout<<"Base constructor. --"<<i<<endl;}
};
class Derived1:virtual public Base{
public:
    Derived1(char i,char j):Base(i)
    {
        cout<<"Derived1 constructor. --"<<j<<endl;
    }
};
class Derived2:virtual public Base{
public:
    Derived2(char i,char j):Base(i)
    {
        cout<<"Derived2 constructor. --"<<j<<endl;
    }
```

```
};
class MyDerived:public Derived1,public Derived2{
public:
    MyDerived(char i,char j,char k,char l,char m,char n,char x)
        :Derived2(i,j),Derived1(k,l),Base(m),d(n)
    {
        cout<<"MyDerived constructor. --"<<x<<endl;

    }
private:
    Base d;
};
int main()
{
    MyDerived obj('A','B','C','D','E','F','G');
    return 0;
}
```

程序的输出结果如下：

```
Base constructor. --E
Derived1 constructor. --D
Derived2 constructor. --B
Base constructor. --F
MyDerived constructor. --G
```

【派生类】29. **派生类将继承它的所有基类中除构造函数和析构函数之外的所有成员**。这也就是创建一个派生类的对象时，首先要考虑到调用基类构造函数的原因。析构函数同理，（析构函数调用顺序与构造函数相反）。另外，**每个派生类只需负责它的直接基类的构造**，依次上溯。

【继承，虚基类，二义性】30. 阅读如下程序，回答问题。

```
#include<iostream>
using namespace std;
class A{
public:
    int n;
};
class B:public A{};//class B:virtual public A{};
class C:public A{};//class C:virtual public A{};
class D:public B,public C{
public:
    int getn(){return B::n;}
};
int main()
```

```
{
    D d;
    d.B::n = 10;
    d.C::n = 20;
    cout<<d.getn()<<endl;
    return 0;
}
```

（1）执行程序后，输出结果为____10____;

（2）将有注释的部分修改为注释中的内容，重新编译运行后输出结果为____20_____.

评：有虚基类时，派生类只继承一份该基类的成员，避免了在派生类中访问虚基类成员时产生二义性。

【多态性，联编】***31.** 所谓多态性，就是不同对象收到相同的消息时，产生不同的动作。实现“一个接口，多种方法”。（通过继承相关的不同的类，他们的对象能够对同一个函数调用作出不同的响应。）

在 C++中，多态性的实现和联编这一概念有关。一个源程序经过编译、连接，成为可执行文件的过程是把可执行代码联编在一起的过程。其中在运行之前就完成的联编称为静态联编；而在程序运行时才完成的联编称为动态联编。

静态联编是指系统在编译时就决定如何实现某一动作。静态联编要求在程序编译时就知道调用函数的全部信息。因此，这种联编类型的函数调用速度很快。效率高是静态联编的主要优点。

动态联编是指系统在运行时动态实现某一动作。采用这种联编方式，一定要到程序运行时才能确定调用哪个函数。动态联编的主要优点是：提供了更好的灵活性、问题抽象性和程序易维护性。

静态联编所支持的多态性称为编译时多态性。在 C++中，编译时多态性是通过函数重载和模板体现的。

动态联编所支持的多态性称为运行时多态性。在 C++中，运行时多态性是通过继承和虚函数来实现的。

【虚函数】32. 虚函数在使用时，要注意以下几点：

（1）只有类的成员函数才能声明为虚函数。因为，虚函数仅适用于有继承关系的类对象，所以**普通函数不能声明为虚函数**。

（2）**静态成员函数不能是虚函数**。因为，静态成员函数不受限于某个对象，而虚函数调用要靠特定的对象来决定应该激活哪个函数。

（3）**内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时仍将其看作是非内联的。**

（4）**构造函数不能是虚函数**，因为构造时对象还是一片未定型的空间。只有在构造完成后，对象才能成为一个类的名副其实的实例。

（5）**析构函数可以是虚函数**，而且通常声明为虚函数。

【运算符重载】33. **不能重载的运算符只有 5 个，它们是成员访问运算符“.”、成员指针访问运算符“*”、域运算符“::”、长度运算符 sizeof 和条件运算符“?:”。**前面两个运算符不能重载保证了 C++中访问成员功能的含义不被改变。域运算符和 sizeof 运算符的操作数是类型，而不是普通的变量或表达式，也不具备重载的特征。

【纯虚函数，抽象类】34. 如果将带有虚函数的类中的一个或多个虚函数声明为纯虚函数，则该类就成为抽象类。纯虚函数是在声明时“初始化值”为 0 的函数。
（1）抽象类只能作为其他类的基类，不能声明抽象类的对象，但可以声明指向抽象类的指针变量和引用变量；也就是说，抽象类只能用作基类来派生新类，而不能用来创建对象。
（2）抽象类中可以有多个纯虚函数，也可以定义其他非纯虚函数。
（3）如果在派生类中没有重新定义基类中的纯虚函数，则必须将该虚函数声明为纯虚函数，这时，这个派生类仍然是一个抽象类；如果在派生类中给出了所有纯虚函数的具体实现，则该派生类就不再是抽象类。
（4）从抽象类可以派生出具体类或抽象类，但不能从具体类派生出抽象类。

【虚函数，sizeof】35. 在用 sizeof 求派生类的大小时，一定不能忘了要包括继承来的基类数据成员的大小。如果该派生类的基类中含有虚函数（包含一个虚函数和多个虚函数的类的长度没有区别，因为一个类中的所有虚函数的入口地址都包含在该类的 VTABLE 表中，在对象中多存储的是一个指向 VTABLE 表的 vptr 指针），那么大小将多 4 个字节，这 4 个字节正好是一个地址值，这个地址指向的就是虚表 VTABLE。

【继承】36.有如下程序：

```
#include<iostream>
using namespace std;
class Base{
public:
    void print(){cout<<'B';}
};
class Derived:public Base{
public:
    void print(){cout<<'D';}
};
int main()
{
    Derived *pd = new Derived;
    Base *pb = pd;
    pb->print();
    pd->print();
    delete pd;
    return 0;
}
```

程序执行后，输出结果为（B）。
A.BB　　B.BD　　C.DB　　D.DD
分析：由于基类中与派生类中同名的函数并不是虚函数，不存在多态机制。另外，以我之见，指针 pb 只是指向了派生类对象的基类部分，由于派生类包含基类。

【虚函数，指针，引用】37.阅读程序，写出运行结果：

```
#include<iostream>
```

```
using namespace std;
class Base{
public:
    virtual void Who(){cout<<"B";}
};
class FD:public Base{
public:
    void Who(){cout<<"F";}
};
class SD:public Base{
public:
    void Who(){cout<<"S";}
};
int main()
{
    Base base_obj;
    FD f_obj;
    SD s_obj;
    base_obj.Who();
    base_obj = f_obj;
    base_obj.Who();
    base_obj = s_obj;
    base_obj.Who();
    Base &bref = f_obj;
    bref.Who();
    bref = s_obj; //引用一旦确定就不会再是别的对象的别名，但可以修改引用所指对象的值
    bref.Who();
    Base *bp;
    bp = &f_obj;
    bp->Who();
    bp = &s_obj;
    bp->Who();
    cout<<endl;
    return 0;
}
```

程序运行结果为：BBBFFFS

分析：将派生类的对象赋值给基类对象，是将派生类对象的基类部分的成员数据赋值给基类的成员数据，该基类对象仍然是一个基类对象。引用一旦确定，将不能改变所指对象，但是可以通过引用修改它所指向的对象的值。

【性能】38. 按值调用传递对象的安全性较好，因为被调用函数无法访问原始对象，但如果要复制大对象，则按值调用可能使性能下降。对象按引用调用传递时可以按指针或对象引用传递。按引用调用有性能优势，但安全性较差，因为被调用函数可以访问原始对象。**按常量引用调用则既安全，又有性能优势。**

【继承】39. 派生类的对象可作为其 public 基类的对象处理，但是反过来不行。

程序员可以用显示类型转换把基类指针转换为派生类指针。但是，如果要复引用该指针，那么在转换前首先应该把它指向某个派生类的对象。