

### 答案：

多重继承在语言上并没有什么很严重的问题，但是标准本身只对语义做了规定，而对编译器的细节没有做规定。所以在使用时（即使是继承），最好不要对内存布局等有什么假设。此类的问题还有虚析构函数等。为了避免由此带来的复杂性，通常推荐使用复合。但是，在《C++设计新思维》（Andrei Alexandrescu）一书中对多重继承和模板有极为精彩的运用。

(1) 多重继承本身并没有问题，如果运用得当可以收到事半功倍的效果。不过大多数系统的类层次往往有一个公共的基类，就像 MFC 中的 Cobject，Java 中的 Object。而这样的结构如果使用多重继承，稍有不慎，将会出现一个严重现象——菱形继承，这样的继承方式会使得类的访问结构非常复杂。但并非不可处理，可以用 virtual 继承（并非唯一的方法）及 Loki 库中的多继承框架来掩盖这些复杂性。

(2) 从哲学上来说，C++多重继承必须要存在，这个世界本来就不是单根的。从实际用途上来说，多重继承不是必需的，但这个世界上有多少东西是必需的呢？对象不过是一组有意义的数据集合及其上的一组有意义的操作，虚函数（晚期绑定）也不过是一堆函数入口表，重载也不过是函数名扩展，这些东西都不是必需的，而且对它们的不当使用都会带来问题。但是没有这些东西行吗？很显然，不行。

(3) 多重继承在面向对象理论中并非是必要的——因为它不提供新的语义，可以通过单继承与复合结构来取代。而 Java 则放弃了多重继承，使用简单的 interface 取代。多重继承是把双刃剑，应该正确地对待。况且，它不像 goto，不破坏面向对象语义。跟其他任何威力强大的东西一样，用好了会带来代码的极大精简，用坏了那就不用说了。

C++是为实用而设计的，在语言里有很多东西存在着各种各样的“缺陷”。所以，对于这种有“缺陷”的东西，它的优劣就要看使用它的人。C++不回避问题，它只是把问题留给使用者，从而给大家更多的自由。像 Ada、Pascal 这类定义严格的语言，从语法上回避了问题，但并不是真正解决了问题，而使人做很多事时束手束脚（当然，习惯了就好了）。

(4) 多重继承本身并不复杂，对象布局也不混乱，语言中都有明确的定义。真正复杂的是使用了运行时多态（virtual）的多重继承（因为语言对于多态的实现没有明确的定义）。为什么非要说多重继承不好呢？如果这样的话，指针不是更容易出错，运行时多态不是更不好理解吗？

因为 C++中没有 interface 这个关键字，所以不存在所谓的“接口”技术。但是 C++可以很轻松地做到这样的模拟，因为 C++中的不定义属性的抽象类就是接口。

(5) 要了解 C++，就要明白有很多概念是 C++试图考虑但是最终放弃的设计。你会发现很多 Java、C#中的东西都是 C++考虑后放弃的。不是说这些东西不好，而是在 C++中它将破

坏 C++作为一个整体的和谐性，或者 C++并不需要这样的东西。举一个例子来说明，C#中有一个关键字 base 用来表示该类的父类，C++却没有对应的关键字。为什么没有？其实 C++中曾经有人提议用一个类似的关键字 inherited，来表示被继承的类，即父类。这样一个好的建议为什么没有被采纳呢？这本书中说得很明确，因为这样的关键字既不必须又不充分。不必须是因为 C++有一个 typedef \* inherited，不充分是因为有多个基类，你不可能知道 inherited 指的是哪个基类。

很多其他语言中存在的时髦的东西在 C++中都没有，这之中有的是待改进的地方，有的是不需要，我们不能一概而论，需要具体问题具体分析。

**面试例题 2：** 声明一个类 Jetplane，它是从 Rocket 和 Airplane 继承而来的。

**解析：**多重继承问题。

**答案：**

```
class JetPlane: public Rocket, public Airplane
```

**面试例题 3：** 在多继承的时候，如果一个类继承同时继承自 class A 和 class B，而 class A 和 B 中都有一个函数叫 foo()，如何明确地在子类中指出 override 是哪个父类的 foo()？

**解析：**多重继承问题。

**答案：**比如，C 继承自 A 和 B，如果出现了相同的函数 foo()，那么 C.A::foo(), C.B::foo() 就分别代表从 A 类中继承的 foo 函数和从 B 类中继承的 foo 函数。

```
#include <iostream>
#include <memory.h>
#include<assert.h>

using namespace std;
class A
{
public :
    void foo()
    {} ;
};

class A2
{
public :
    void foo()
    {} ;
};

class D : public A,public A2
{
};

int main()
{
    D d;
    d.A::foo() ;

    return 0;
}
```

**面试例题 4：**下面程序的输出结果是多少？[英国某图形软件公司 2009 年 10 月面试题]

```
#include <iostream>
using namespace std;
class A{
    int m_nA;
};

class B{
    int m_nB;
};

class C : public A, public B{
    int m_nC;
};
```

```

};

int main() {
    C* pC = new C;
    B* pB = dynamic_cast<B*>(pC);
    A* pA = dynamic_cast<A*>(pC);
    //cout << pC << endl;
    //cout << pB << endl;
    //cout << (C *)pB << endl;
    //cout << pC << endl;
    if(pC==pB)
        {cout << "equal" << endl;}
    else

```

```

    {cout << "not equal" << endl;}

    //cout << pC << endl;
    //cout << pB << endl;
    //cout << int(pC) << endl;
    //cout << int(pB) << endl;
    if(int(pC)==int(pB))
        {cout << "equal" << endl;}
    else
        {cout << "not equal" << endl;}
    return 0;
}

```

**解析：**本题涉及基类和派生类的地址和布局的问题。

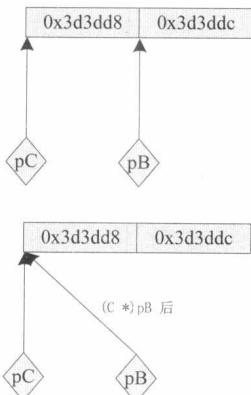
```
if (pC==pB)
```

这里两端的数据类型不同，比较时需要进行隐式类型转换。`(pC==pB)`相当于：

```
pC==(C *)pB // equal
```

`pB`实际上指向的地址是对象`C`中的父类`B`部分，从地址上跟`pC`不一样，所以直接比较地址数值的时候是不相等的。

但是，当进行`pC==pB`比较时，实际上是比较`pC`指向的对象和`(C *)`隐式转换`pB`后`pB`指向的对象（`pC`指向的对象）的部分，这个是同一部分，也就显示相等了。假设`pC`指向的地址是`0x3d3dd8`，`pB`指向的地址是`0x3d3ddc`。`(C *)`隐式转换`pB`后`pB`指向的地址也就变成了`0x3d3dd8`，如下图所示。



第二处两端转换为`int`，指针`pC`和`pB`的值不同，转换后的`int`值不同：

```
int (pC)==int (pB) // not equal
```

如果转化成下面的形式，就可以了：

```
int (pC)==int ((C *)pB) //equal
```

答案：输出如下：

```
equal
not equal
```

## 11.5 检测并修改不适合的继承

**面试例题 1：**如果鸟是可以飞的，那么鸵鸟是鸟么？鸵鸟如何继承鸟类？[美国某著名分析软件公司 2005 年面试题]

**解析：**如果所有鸟都能飞，那鸵鸟就不是鸟。回答这种问题时，不要相信自己的直觉。将直觉和合适的继承联系起来还需要一段时间。

根据题干可以得知：鸟是可以飞的。也就是说，当鸟飞行时，它的高度是大于 0 的。鸵鸟是鸟类（生物学上）的一种，但它的飞行高度为 0（鸵鸟不能飞）。

不要把可替代性和子集相混淆。即使鸵鸟集是鸟集的一个子集（每个鸵鸟集都在鸟集中），但并不意味着鸵鸟的行为能够代替鸟的行为。可替代性与行为有关，与子集没有关系。当评价一个潜在的继承关系时，重要的因素是可替代的行为，而不是子集。

**答案：**如果一定要让鸵鸟来继承鸟类，可以采取组合的办法，把鸟类中的可以被鸵鸟继承的函数挑选出来，这样鸵鸟就不是“a kind of”鸟了，而是“has some kind of”鸟的属性而已。代码如下：

```
#include<iostream>
#include<string>
using namespace std;

class bird
{
public:
    void eat();
    void sleep();
    void fly();
};

class ostrich
{
```

```
public:
    bird eat() {cout<<"ostrich eat";};
    bird sleep() {cout<<"ostrich sleep";};

};

int main()
{
    ostrich xiaoq;
    xiaoq.eat();
    xiaoq.sleep();
    return 0;
}
```

**面试例题 2：**Find the defects in each of the following programs, and explain why it is incorrect. (找出下面程序的错误，并解释它为什么是错的。) [中国台湾某著名杀毒软件公司 2005 年面试题]

```
#include <iostream>
using namespace std;
```

```
class Base {
public:
    int val;
```

```

        Base() { val=1; };
    };

    class Derive: Base {
        public:
            int val;
            Derive(int i) { val=Base::val+i; };
    };
}

```

```

int main(int, char**, char**) {
    Derive d(10);
    cout<<d.Base::val<<endl
        <<d.val<<endl;
    return 0;
}

```

**解析：**这是个类继承问题。如果不指定 `public`，C++默认的是私有继承。私有继承是无法继承并使用父类函数中的公有变量的。

**答案：**把 `class Derive: Base` 改成 `class Derive:public Base`。

## 扩展知识（组合）

若在逻辑上 A 是 B 的“一部分（a part of）”，则不允许 B 从 A 派生，而是要用 A 和其他东西组合出 B。

例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，而不是派生而成。程序如下：

```

class Eye
{
    public:
        void Look(void);
};

class Nose
{
    public:
        void Smell(void);
};

class Mouth
{
    public:
        void Eat(void);
};

class Ear
{
    public:
        void Listen(void);
};

```

```

};

class Head
{
    public:
        void Look(void)
        { m_eye.Look(); }
        void Smell(void)
        { m_nose.Smell(); }
        void Eat(void)
        { m_mouth.Eat(); }
        void Listen(void)
        { m_ear.Listen(); }

    private:
        Eye m_eye;
        Nose m_nose;
        Mouth m_mouth;
        Ear m_ear;
};

```

Head 由 Eye、Nose、Mouth、Ear 组合而成。如果允许 Head 从 Eye、Nose、Mouth、Ear 派生而成，那么 Head 将自动具有 Look、Smell、Eat、Listen 这些功能。程序十分简短并且运行正确，但是下面这种设计方法却是不对的。

```

class Head : public Eye, public Nose, public Mouth, public Ear
{
};

```

**面试例题 3:** Find the defects in each of the following programs, and explain why it is incorrect. (找出下面程序的错误，并解释它为什么是错的。) [德国某著名软件咨询企业 2005 年面试题]

```
class base{
private: int i;
public: base(int x){i=x;}
};

class derived: public base{
```

```
private: int i;
public: derived(int x, int y) {i=x;}
void printTotal() {int total =
i+base::i;}
};
```

**解析:** 要在子类中设定初始成员变量，把 derived(int x, int y) 改成 derived(int x, int y) : base(x)。

**答案:**

代码如下：

```
class base
{
protected: //这里的访问属性需要改变
    int i;
public:
    base(int x){i=x;}
};

class derived: public base
{
private:
    int i;
```

```
public:
derived(int x, int y) : base(x)
//以前没有初始化基类的成员变量
{
    i=y;
}
void printTotal()
{
    int total = i+base::i;
}
```

## 11.6 纯虚函数

**面试例题 1:** 下面的程序有何错误？[德国某著名软件咨询企业 2004 年面试题]

```
#include <iostream>
using namespace std;
class Shape
{
public:
    Shape(){}
    ~Shape(){}
};
```

```
virtual void Draw()=0;
};

int main()
{
    Shape s1;
```

**解析:** 因为 Shape 类中的 Draw 函数是一个纯虚函数，所以 Shape 类是不能实例化一个对象的。Shape s1; 是不可以的，解决方法是把 Draw 函数修改成一般的虚函数。

**答案:**

修改后的代码如下：

```
#include <iostream>
using namespace std;
class Shape
```

```
{ public:
    Shape(){}
};
```

```

~Shape(){}
virtual void Draw(){}
};

```

```

int main()
{
    Shape s1;
}

```

### 面试例题2：什么是虚指针？[美国某著名移动通信企业面试题]

**答案：**虚指针或虚函数指针是一个虚函数的实现细节。带有虚函数的类中的每一个对象都有一个虚指针指向该类的虚函数表。

### 面试例题3：声明一个类 Vehicle，使其成为抽象数据类型。写出类 Car 和 Bus 的声明，其中每个类都从类 Vehicle 里派生。使 Vehicle 成为一个带有两个纯虚函数的 ADT，使 Car 和 Bus 不是 ADT。[美国某著名移动通信企业面试题]

**答案：**

```

class Vehicle
{
public:
    virtual void Move()=0;
    virtual void Haul()=0;
};

class Car : public Vehicle
{
public:

```

```

    virtual void Move();
    virtual void Haul();
};

class Bus : public Vehicle
{
public:
    virtual void Move();
    virtual void Haul();
};

```

### 面试例题4：虚函数的入口地址和普通函数有什么不同？[英国某著名计算机图形图像公司面试题]

**答案：**每个虚函数都在 vtable 中占了一个表项，保存着一条跳转到它的入口地址的指令（实际上就是保存了它的入口地址）。当一个包含虚函数的对象（注意，不是对象的指针）被创建的时候，它在头部附加一个指针，指向 vtable 中相应的位置。调用虚函数的时候，不管是用什么指针调用的，它先根据 vtable 找到入口地址再执行，从而实现了“动态联编”。而不像普通函数那样简单地跳转到一个固定地址。

### 面试例题5：

1. C++中如何阻止一个类被实例化？
2. 构造函数被声明成 private 会阻止编译器生成默认的 copy constructor 吗？
3. 什么时候编译器会生成默认的 copy constructor 呢？
4. 如果你已经写了一个构造函数，编译器还会生成 copy constructor 吗？[英国某著名计算机图形图像公司面试题]

**答案：**

1. 使用抽象类，或者构造函数被声明成 private。

2. 构造函数设为 private，并不能阻止编译器生成默认的 copy constructor，这是毫不相干的两件事情。如果我们没有定义复制构造函数，编译器就会为我们合成一个。与合成默认构造函数不同，即使我们定义了其他构造函数，也会合成复制构造函数。

3. 只要自己没写，而程序中需要，都会生成。
4. 会生成。

## 扩展知识

---

在 C# 中，有一个专门的 seal class（封闭类）来防止继承。

---

## 11.7 运算符重载与 RTTI

**面试例题 1：** Which of the following statements provide a valid reason NOT to use RTTI for distributed (i.e. networked between different platforms) applications in C++? （在分布式系统中，不使用 RTTI 的一个合理解释是？） [中国某互联网公司 2010 年 6 月面试题]

- A. RTTI is too slow and use too much memory (RTTI 太慢了)
- B. RTTI does not have standardized run-time behavior (RTTI 不是一个标准行为)
- C. TTI's performance is unpredictable/non-deterministic and is lack of expansibility (RTTI 行为不可预期及缺乏扩展性)
- D. RTTI must fail to function correctly at run-time (RTTI 函数在运行时会失败)

**解析：** C++ 中的每个特性，都是从程序员平时生活中逐渐精化而来的。在不正确的场合使用它们必然会引起逻辑、行为和性能上的问题。对于上述特性，应该只在必要、合理的前提下才使用。

C++ 引入的额外开销体现在以下两方面。

### 1. 编译时开销

模板、类层次结构、强类型检查等新特性，以及大量使用了这些新特性的 C++ 模板、算法库都明显地增加了 C++ 编译器的负担。但是应当看到，这些新机能再不增加程序执行效率的前提下，明显降低了广大 C++ 程序员的工作量。

### 2. 运行时开销

运行时开销恐怕是程序员最关心的问题之一了。相对于传统 C 程序而言，C++ 中有可能引入额外运行时开销特性包括：

- 虚基类。
- 虚函数。
- RTTI (dynamic\_cast 和 typeid)。
- 异常。
- 对象的构造和析构。

虚基类，从直接虚继承的子类中访问虚基类的数据成员或其虚函数时，将增加两次指针引用（大部分情况下可以优化为一次）和一次整型加法的时间开销。定义一个虚基类表，定义若干虚基类表指针的空间开销。

虚函数的运行开销有进行整型加法和指针引用的时间开销。定义一个虚表，定义若干个（大部分情况下是一个）虚表指针的空间开销。

RTTI 的运行开销主要有进行整型比较和取址操作（可能还会有一两次整形加法）所增加的时间开销。定义一个 type\_info 对象（包括类型 ID 和类名称）的空间开销。“dynamic\_cast”用于在类层次结构中漫游，对指针或引用进行自由的向上、向下或交叉转化。“typeid”则用于获取一个对象或引用的确切类型。一般地讲，能用虚函数解决的问题就不要用“dynamic\_cast”，能够用“dynamic\_cast”解决的就不要用“typeid”。

关于异常，对于几乎所有编译器来说，在正常情况（未抛出异常）下，try 块中的代码执行效率和普通代码一样高，而且由于不再需要使用传统上通过返回值或函数调用来判断错误的方式，代码的实际执行效率还会进一步提高。抛出和捕捉异常的开销也只是在某些情况下会高于函数返回和函数调用的开销。

关于构造和析构，开销也不总是存在的。对于不需要初始化/销毁的类型，并没有构造和析构的开销，相反对于那些需要初始化/销毁的类型来说，即使使用传统的 C 方式实现，也至少需要与之相当的开销。

实事求是地讲，RTTI 是有用的。但因为一些理论上及方法论上的原因，它破坏了面向对象的纯洁性。

首先，它破坏了抽象，使一些本来不应该被使用的方法和属性被不正确地使用。其次，因为运行时类型的不确定性，它把程序变得更脆弱。第三点，也是最重要的一点，它使程序缺乏扩展性。当加入了一个新的类型时，你也许需要仔细阅读你的 dynamic\_cast 或 instanceof 的代码，必要时改动它们，以保证这个新的类型的加入不会导致问题。而在这个过程中，编译器将不会给你任何帮助。

很多人一提到 RTTI，总是侧重于它的运行时的开销。但是，相比于方法论上的缺点，这点运行时的开销真是无足轻重的。

总的来说，RTTI 因为它的方法论上的一些缺点，它必须被非常谨慎地使用。今天面向对象语言的类型系统中的很多东西就是产生于避免 RTTI 的各种努力。

答案：C

## 扩展知识

RTTI 是 Runtime Type Information 的缩写，从字面上来理解就是执行时期的类型信息，其重要作用就是动态判别执行时期的类型。有的读者会认为设计类时使用虚函数就已经足够了，可是虚函数有本身的局限性，当涉及类别阶层时，需要判断某个对象所属的类别，而因为类别设计中大量使用了虚函数，所以使得这一工作难以实现，但又极其重要，于是使用 RTTI 的 typeid 运算符能使程序员确定对象的动态类型。

借用一个例子（也是一道面试题）：写了一个 base 类，派生 derived 类，在 base 类中实现一些常见的方法。下面两个函数要求能够输出形参的真实类型，funcC 是用 dynamic\_cast 类型转换是否成功来识别类型的，dynamic\_cast 操作符将基类类型对象的引用或指针转化为同一继承层次中的其他类型的引用或指针。如果绑定到引用或指针的对象不是目标类型的对象，则 dynamic\_cast 失败。如果转换到指针类型的 dynamic\_cast 失败，则 dynamic\_cast 的结果是 0 值；如果转换到引用类型的 dynamic\_cast 失败，则抛出一个 bad\_cast 类型的异常；funcD 是用 typeid 判断基类地址是否一致的办法来识别类型的。

```
#include <iostream>
#include <typeinfo>
using namespace std;
class base
{
public:
    virtual void funcA() {cout << "base"
<< endl;}
};
class derived : public base
{
public:
    virtual void funcB() {cout << "derived"
<< endl;}
};
void funcC(base *p)
{
    derived*dp=dynamic_cast<derived*>(p);
    if(dp != NULL)
        dp->funcB();
    else
        p->funcA();
}
//funcD 用 typeid 操作符
```

```
void funcD(base *p)
{
    derived *dp = NULL;
    if (typeid(*p) == typeid(derived))
    {
        dp = static_cast<derived*>(p);
        dp->funcB();
    }
    else
        p->funcA();
}
int main()
{
    base *cp = new derived;
    cout << typeid(cp).name() << endl;
    cout << typeid(*cp).name() << endl;
    funcD(cp);
    funcC(cp);
    base *dp = new base;
    funcC(dp);
    funcD(dp);
    return 0;
}
```

运行时类型识别 RTTI 使用时要注意以下几点：

- 用 typeid() 返回一个 typeinfo 对象，也可以用于内部类型，当用于非多态类型时，没有虚函数，用 typeid 返回的将是基类地址。
- 动态映射 dynamic\_cast<类型>（变量）可以映射到中间级，将派生类映射到任何一个基类，然后在基类之间可以相互映射。
- 不能对 void 指针进行映射。
- 如果 p 是基类指针，并且指向一个派生类型的对象，并且基类中有虚函数，那么 typeid(\*p) 返回 p 所指向的派生类类型，typeid(p) 返回基类类型。
- 典型的 RTTI 是通过在 VTABLE 中放一个额外的指针实现的。每个新类只产生一个 typeinfo 实例，额外指针指向 typeinfo，typeid 返回对它的一个引用。
- 动态映射 dynamic\_cast<目标\*><源指针>，先恢复源指针的 RTTI 信息，再取目标的 RTTI 信息，比较两者是否相同，或者是目标类型的基类；由于它需要检查一长串基类列表，故动态映射的开销比 typeid 大。

typeid 是 C++ 的关键字之一，等同于 sizeof 这类的操作符。typeid 操作符的返回结果是名为 type\_info 的标准库类型的对象的引用（在头文件 typeinfo 中定义，稍后我们看一下 vs 和 gcc 库里面的源码），它的表达式有下图两种形式。

typeid	类型 ID	typeid(type)
typeid	运行时刻类型 ID	typeid(expr)

如果表达式的类型是类类型且至少包含有一个虚函数，则 typeid 操作符返回表达式的动态类型，需要在运行时计算；否则，typeid 操作符返回表达式的静态类型，在编译时就可以计算。

C++ 标准规定了其实现必需提供如下四种操作。

t1 == t2	如果两个对象 t1 和 t2 类型相同，则返回 true；否则返回 false
t1 != t2	如果两个对象 t1 和 t2 类型不同，则返回 true；否则返回 false
t.name()	返回类型的 C-style 字符串，类型名字用系统相关的方法产生
t1.before(t2)	返回指出 t1 是否出现在 t2 之前的 bool 值

type\_info 类提供了 public 虚析构函数，以使用户能够用其作为基类。它的默认构造函数和复制构造函数及赋值操作符都定义为 private，所以不能定义或复制 type\_info 类型的对象。程序中创建 type\_info 对象的唯一方法是使用 typeid 操作符（由此可见，

如果把 typeid 看作函数的话，其应该是 type\_info 的友元）。这具体由编译器的实现所决定，标准只要求实现为每个类型返回唯一的字符串。

**面试例题 2：**A C++ developer wants to handle a static\_cast<char\*>() operation for the class String shown below. Which of the following options are valid declarations that will accomplish this task? (一个 C++程序员想要运行一个 static\_cast<char\*>(), 为了能够确保合法化，下面这段代码横线处应填写什么？)

```
class String {
public:
    //...
    //declaration goes here
};
```

- A. char\* operator char\*();
- B. operator char\*();
- C. char\* operator();
- D. char\* operator String();

**解析：**运算符重载问题。

运算符重载就是赋予已有的运算符多重含义。C++中通过重新定义运算符，使它能够用于特定类的对象执行特定的功能，这增强了 C++语言的扩充能力。运算符重载的作用是允许程序员为类的用户提供一个直觉的接口。通过重载类上的标准运算符，可以发掘类的用户的直觉。使得用户程序所用的语言是面向问题的，而不是面向机器的。最终目标是降低理解难度并减少错误率。

几乎所有的运算符都可用作重载。具体包含：

```
算术运算符: +, -, *, /, %, ++, --;
位操作运算符: &, |, ~, ^, <<, >>;
逻辑运算符: !, &&, ||;
比较运算符: <, >, >=, <=, ==, !=;
赋值运算符: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=;
其他运算符: [], (), ->, (逗号运算符), new, delete, new[], delete[], ->*.
```

下列运算符不允许重载：

```
., *, ::, ?:
```

用户重载新定义运算符，不改变原运算符的优先级和结合性。这就是说，对运算符重载不改变运算符的优先级和结合性，并且运算符重载后，也不改变运算符的语法结构，即单目运算符只能重载为单目运算符，双目运算符只能重载双目运算符。运算符重载实际是一个函数，所以运算符的重载实际上是函数的重载。编译程序对运算符重载的选择，遵循着函数重载的选择原则。当遇到不很明显的运算时，编译程序将去寻找参数相匹配的运算符函数。

运算符重载可以使程序更加简洁，使表达式更加直观，增加可读性。但是，运算符重载使用不宜过多，否则会带来一定的麻烦。运算符重载的函数一般采用如下两种形式：成员函

数形式和友元函数形式。这两种形式都可访问类中的私有成员。

这里有一个复数运算重载的四则运算符的例子。复数由实部和虚部构造，可以定义一个复数类，然后再在类中重载复数四则运算的运算符：

```
#include <iostream>
#include <typeinfo>
using namespace std;

class classFushu
{
public:
    classFushu() { shibu=xubu=0; }
    classFushu(double r, double i)
    {
        shibu = r, xubu = i;
    }
    classFushu operator +(const classFushu &c);
    classFushu operator -(const classFushu &c);
    classFushu operator *(const classFushu &c);
    classFushu operator /(const classFushu &c);
    friend void print(const classFushu &c);
private:
    double shibu, xubu;
};

inline classFushu classFushu::operator +(const classFushu &c)
{
    return classFushu(shibu + c.shibu, xubu + c.xubu);
}

inline classFushu classFushu::operator -(const classFushu &c)
{
    return classFushu(shibu - c.shibu, xubu - c.xubu);
}

inline classFushu classFushu::operator *(const classFushu &c)
{
    return classFushu(shibu * c.shibu - xubu * c.xubu, shibu * c.xubu + xubu * c.shibu);
}

inline classFushu classFushu::operator /(const classFushu &c)
{
    return
    classFushu((shibu * c.shibu + xubu * c.xubu) / (c.shibu * c.shibu + c.xubu * c.xubu),
               (xubu * c.shibu - shibu * c.xubu) / (c.shibu * c.shibu + c.xubu * c.xubu));
}

void print(const classFushu &c)
{
    if(c.xubu<0)
        cout<<c.shibu<<c.xubu<<'i';
    else
        cout<<c.shibu<<'+<<c.xubu<<'i';
}

int main()
```

```

{
    classFushu c1(1.0, 2.0), c2(3.0, 4.0), c3;
    c3 = c1 + c2;
    cout<<"\nc1+c2=";
    print(c3);
    c3 = c1 - c2;
    cout<<"\nc1-c2=";
    print(c3);
    c3 = c1 * c2;
    cout<<"\nc1*c2=";
    print(c3);
    c3 = c1 / c2;
    cout<<"\nc1/c2=";
    print(c3);
    c3 = (c1+c2) * (c1-c2) * c2/c1;
    cout<<"\n(c1+c2)*(c1-c2)*c1/c2=";
    print(c3);
    cout<<endl;
    return 0;
}

```

在本题中, static\_cast <char\*>实现强制类型转换, char\*是目标类型; operation char\*() 是转换函数, 无参数, 无返回类型, 以目标类型为其函数名, 其函数定义中要返回转换结果。

答案: B

**面试例题 3:** Which of the following options are returned by the typeid operator in C++? (C++里面的 typeid 运算符返回值是什么?)

- A. A reference to a std::type\_info object (type\_info 对象的引用)
- B. A const reference to a const std::type\_info object (type\_info 常量对象的常量引用)
- C. A const reference to a std::type\_info object (type\_info 对象的常量引用)
- D. A reference to a const std::type\_info object (type\_info 常量对象的引用)

**解析:** 本题用于查看 typeid 的输出结果, typeid 是用来获取一个对象或引用的确切类型。先看一段程序:

```

#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
    int vInt=10;
    int arr[2]={10,20};
    int *p=&vInt;
    int **p2p=&p;
    int *parr[2]={&vInt,&vInt};
    int (*p2arr)[2]=&arr;
    cout<<"Declaration [int vInt=10] type=="<<typeid(vInt).name()<<endl;
}

```

```

cout<<"Declaration[arr[2]={10,20}]type=="<<typeid(arr).name()<<endl;
cout<<"Declaration[int*p=&vInt]type=="<<typeid(p).name()<<endl;
cout<<"Declaration[int**p2p=&p]type=="<<typeid(p2p).name()<<endl;
cout<<"Declaration[int *parr[2]={&vInt,&vInt}]type=="<<typeid(parr).name()<<endl;
cout<<"Declaration[int (*p2arr)[2]=&arr]type=="<<typeid(p2arr).name()<<endl;
return 0;
}

```

本段程序在 VC++2008 的输出结果如下（VC6.0 和 dev 输出可能会有所不同）：

```

Declaration [int vInt=10] type==int
Declaration [arr[2]={10,20}] type==int [2]
Declaration [int *p=&vInt] type==int *
Declaration [int **p2p=&p] type==int * *
Declaration [int *parr[2]={&vInt,&vInt}] type==int * [2]
Declaration [int (*p2arr)[2]=&arr] type==int (*)[2]

```

请读者对照上面代码详细研读 typeid 的输出规律。

本题选择 D, type\_info 常量对象的引用。

答案：D

**面试例题 4:** Which of the following statements accurately describe unary operator overloading in C++? (一元运算符重载，下列说法正确的是哪项？)

- A. A unary operator can be overloaded with no parameters when the operator function is a class member (当运算符函数是一个类成员时，一元运算符能被无参形式重载)
- B. A unary operator can be overloaded with one parameter when the operator function is a class member (当运算符函数是一个类成员时，一元运算符能被带一个参数形式重载)
- C. A unary operator can be overloaded with 2 parameters when the operator function is free standing function (not a class member) (当运算符函数是一个独立函数时，一元运算符能被带两个参数形式重载)
- D. A unary operator can only be overloaded if the operator function is a class member (当且仅当运算符函数是一个类成员时，一元运算符才会被重载)

**解析：**本题考的知识点是运算符重载问题。

定义一个重载运算符就像定义一个函数，只是该函数的名字是 operator@，这里@代表运算符。函数参数表中参数的个数取决于两个因素：

- 运算符是一元的（一个参数）还是二元的（两个参数）。
- 运算符被定义为全局函数（对于一元运算符是一个参数，对于二元运算符是两个参数），如果运算符是成员函数（对于一元运算符没有参数，对于二元运算符是一个参数）。

对于二元运算符，单个参数是出现在运算符右侧的那个。当一元运算符被定义为成员函数时，没有参数。成员函数被运算符左侧的对象调用。在 C++ 中，后缀`++`如果是成员函数，那么它就是二元的操作符。这里 C++ 标准中有详细的阐述：

A binary operator shall be implemented either by a non-static member function with one parameter or by a non-member function with two parameters. (非静态成员函数操作符（带 1 个参数）是二元运算符；非成员函数操作符（带两个参数）是二元运算符。)

一个常见的例子如下：

```
#include <iostream>
using namespace std;
class A
{
private:
    int a;
public :
    A(){a=0;}
    void operator++() //这是一元操作
    {
        a += 1;
    }

    void operator++(int) //这是二元操作
    {
        a += 2;
    }
    friend void print(const A &c);
};
```

```
void print(const A &c)
{
    cout <<c.a;
}

int main( )
{
    A classa;
    print(classa);
    ++classa; //这是一元操作

    print(classa);
    classa++; //这是二元操作
    print(classa);

    return 0;
}
```

对于非条件运算符（条件运算符通常返回一个布尔值），如果两个参数是相同的类型，希望返回和运算相同类型的对象或引用。如果它们不是相同类型，它作什么样的解释就取决于程序设计者。

答案：A

# 第 12 章

## 位运算与嵌入式编程

C 语言测试是招聘嵌入式系统程序员必须且有效的方法。我参加了许多这种测试，在此过程中我意识到这些测试能为面试者和被面试者提供许多有用的信息。此外，撇开面试的压力不谈，这种测试也相当有趣。

从被面试者的角度来讲，你能了解许多关于出题者或监考者的情况。这个测试只是出题者为显示其对 ANSI 标准细节的知识而不是技术技巧而设计的吗？如要你答出某个字符的 ASCII 值。这些面试例题着重考查你的系统调用和内存分配策略方面的能力吗？这标志着出题者也许花时间在微机上而不是在嵌入式系统上。如果上述任何问题的答案是“是”的话，那么就得认真考虑是否应该去做这份工作。

从面试者的角度来讲，一个测试也许能从多方面揭示应试者的素质。最基本的，你能了解应试者 C 语言的水平。应试者是以好的直觉做出明智的选择，还是只是瞎蒙呢？当应试者在某个问题上卡住时是找借口，还是表现出对问题的真正的好奇心，把这看成学习的机会呢？我发现这些信息与面试者们的测试成绩一样有用。

有了这些想法，我们再结合一些真正针对嵌入式系统的考题，希望这些令人头痛的考题能给正在找工作的人一点儿帮助。其中有些题很难，但它们应该都能给你一点儿启迪。

### 12.1 位制转换

**面试例题 1：**求下列程序的输出结果。[美国某著名计算机硬件公司面试题]

```
#include<stdio.h>
int main()
{
    printf("%f",5);
    printf("%d",5.01);
}
```

**解析：**首先参数 5 为 int 型，32 位平台中为 4 字节，因此在 stack 中分配 4 字节的内存，用于存放参数 5。

然后 printf 根据说明符 “%f”，认为参数应该是个 double 型（在 printf 函数中，float 会自动转换成 double），因此从 stack 中读了 8 个字节。

很显然，内存访问越界，会发生什么情况不可预料。如果在 printf 或者 scanf 中指定了 “%f”，那么在后面的参数列表中也应该指定一个浮点数，或者一个指向浮点变量的指针，否则不应加载支持浮点数的函数。

于是("%f",5)有问题，而("%f",5.0)则可行。

**答案：**

第一个答案是 0.000000。

第二个答案是一个大数。

**面试例题 2：**下列程序是否有错？如果有，错在哪里？[美国著名计算机公司 I 2007 年 5 月面试题]

```
#include <iostream>
using namespace std;
struct a{
int x:1;
int y:2;
int z:33;
};
int main()
{
```

```
a d;
cout << &d;
d.z=d.x+d.y;
printf(" %d %d %d %d \n ", d.x,
d.y,d.z,sizeof(d));
return 0;
}
```

**解析：**结构体位制概念。

**答案：**

“int z:33;” 定义整型变量 z 为 33 位，也就是超过了 4 字节。这是不合法的，会造成越界，所以程序会报错。

**面试例题 3：**Find the defects in each of the following programs, and explain why it is incorrect. (找出下面程序的错误，并解释它为什么是错的。) [中国台湾某著名杀毒软件公司 2005 年面试题]

```
//The function need set corresponding bit int 0
#define BIT_MASK(bit_pos) (0x01<<(bit_pos))
int Bit_Reset(unsigned int* val, unsigned char pos) {
    if(pos >= sizeof(unsigned int) * 8) {
        return 0;
    }
    *val=(*val && ~BIT_MASK(pos));
    return 1;
}
```

**解析：**这道程序体存在着位运算问题。

**答案：**\*val=(\*val && ~BIT\_MASK(pos))这一语句中的“**&&**”应为“**&**”。

正确的程序如下所示：

```
#include <iostream>
#define BIT_MASK(bit_pos) (0x01<<(bit_pos))
int Bit_Reset(unsigned int* val,
              unsigned char pos) {
    if(pos >= sizeof(unsigned int) * 8) {
        return 0;
    }
    *val=(*val & ~BIT_MASK(pos));
}
```

```
return 1;
}
int main() {
    unsigned int x=0xffffffff;
    unsigned char y=4;

    Bit_Reset(&x,y);
    std::cout<<std::hex<<x<<'\n';
    return 0;
}
```

**面试例题 4：**In which system(进制) expression  $13*16=244$  is true? (下面哪个进制能表述  $13*16=244$  是正确的?) [中国台湾某计算机硬件公司 V2010 年 5 月面试题]

A. 5

B. 7

C. 9

D. 11

**解析：**13 如果是一个十进制的话，它可以用  $13=1*10^1+3*10^0$  来表示。现在我们不知道 13 是几进制，那我们姑且称其 X 进制。X 进制下的 13 转化为十进制可以用  $13=1*X^1+3*X^0$  表示；X 进制下的 16 转化为十进制可以用  $16=1*X^1+6*X^0$  表示；X 进制下的 244 转化为十进制可以用  $244=2*X^2+4*X^1+4*X^0$  表示；因此 X 进制下的  $13*16=244$  可以转化为十进制下的等式： $(1*X^1+3*X^0)*(1*X^1+6*X^0)=2*X^2+4*X^1+4*X^0$ 。

整理得  $X^2+5X-14=0$ 。最后得出一元二次方程  $X^2+5X-14=0$ 。  
答案  $X=-2$  或者  $X=7$ 。 $X=-2$  不合题意舍弃，所以  $X=7$ 。

**答案：**B

**面试例题 5：**以下代码哪个等同于 `int i = (int)p;`(`p` 的定义为 `char *p`) [中国台湾某计算机硬件公司 2010 年 5 月面试题]

A. `int i = dynamic_cast<int>(p)`

B. `int i = static_cast<int>(p)`

C. `int i = const_cast<int>(p)`

D. `int i = reinterpret_cast<int>(p)`

**解析：**先看这样一段代码：

```
#include <iostream>
int main()
{
    char *p = "a";
    int i = (int)"a";
    int i2 = (int)p;
    //int i3 = static_cast<int>(&p); //fail
    int i5 = reinterpret_cast<int>(&p);
```

```
cout << p << endl;
cout << &p << endl;
cout << i << endl;
cout << i2 << endl;
cout << i5 << endl;
}
```

输出结果如下：

```
a
0x22ff7c
```

```
4199056
4199056
2293628
```

解释如下：

```
cout<<p<<endl;
结果是 "a"，
```

p 是一个 char\* 指针，之所以不输出 p 这个指针的地址，而是输出 p 指针指向的字符串 “a”，那是因为 C++ 输出操作符的实现细节是这样实现的。

```
cout<<&p<<endl;
结果是 0x22ff7c。
```

这是存储 p 指针本身的内存地址。

```
所以才会有*(&p)==p;
```

```
int i = (int)"a";
```

```
cout<<i<<endl;
```

结果是 4199056 此处就是把这个字符串首字母在常量区的内存地址转化为 int 型再赋值给 i。

```
int i2 = (int)p;
```

因为 p 是一个指向常量区字符串 "a" 的 char\* 指针。

所以，这里把 p 的值（即 "a" 的常量区内存地址）转化为 int 型再赋值给 i2，所以 i2 也是 4199056

```
int i5=reinterpret_cast<int>(&p);
```

这里是把指针 p 的地址的位模式用 int 型去重新解释。而指针 p 的地址，即 &p=0x22ff7c。这是前面输出的结果，相当于把这个 0x22ff7c 内存地址用 int 型去解释。因此输出的就是十进制的指针 p 的地址。

所以结果是 2293628。

因为 2293628 转化为 16 进制就是 0x22ff7c。

此题 A, C 选项可以迅速排除。

A 选项，dynamic\_cast 顾名思义是支持动态的类型转换，即支持运行时识别指针或引用所指向的对象。原题明显是静态的类型转换。

C 选项，const\_cast 是转换掉表达式的 const 性质。因此 C 也不符合。

B 选项错在不存在 char\* 到 int 型的隐式转换，编译器会报错：

```
error: invalid
```

```
static_cast from type 'char*' to type 'int'
```

D 选项符合题意，reinterpret\_cast<type>(expression) 就是从位模式的角度用 type 型在较低的层次重新解释这个 expression。

在合法使用 static\_cast 和 const\_cast 的地方，旧式强制转换提供了与各自对应的命名强制转换一样的功能。

如果这两种强制转换均不合法，则旧式强制转换执行 reinterpret\_cast 功能。

因此 D 选项就是旧式强制转换执行 reinterpret\_cast 功能的例子。

答案：D

**面试例题 6：**Given the following program snippet, what can we conclude about the use of dynamic\_cast in C++? (下面这段程序，我们能总结 C++ 中 dynamic\_cast 的用法是什么？) [中]

国台湾某计算机硬件公司 2010 年 5 月面试题]

```
#include <iostream>
#include <memory>
//Someone else's code, e.g. library
class IGlyph
{
public:
    virtual ~IGlyph(){}
    virtual std::string Text()=0;
    virtual IIIcon* Icon()=0;
    //...
};

class IWidgetSelector
{
public:
    virtual ~IWidgetSelector(){}
    virtual void AddItem(IGlyph*)=0;
    virtual IIIcon * Selection()=0;
};

//Your code
class MyItem : public IGlyph
{
public:
    virtual std::string Text()
    {
        return this->text;
    }
}
```

```
virtual IIIcon* Icon()
{
    return this->icon.get();
}

void Activate()
{
    std::cout << "My Item Activated" << std::endl;
}

std::string text;
std::auto_ptr<IIIcon> icon;
};

void SpiffyForm::OnDoubleClick(IWidgetSelector* ws)
{
    IGlyph* gylph = ws->Selection();

    MyItem* item=dynamic_cast<MyItem*>(gylph);
    if(item)
        item->Activate();
}
```

A. The `dynamic_cast` is necessary since we cannot know for certain what concrete type is returned by `IWidgetSelector::Selection()`. (`dynamic_cast` 非常必要，因为我们不知道确定的 `IWidgetSelector::Selection()` 返回的具体类型)

B. The `dynamic_cast` is unnecessary since we know that the concrete type returned by `IWidgetSelector::Selection()` must be a `MyItem` object. (`dynamic_cast` 不是必要的，因为我们知道 `IWidgetSelector::Selection()` 返回的具体类型一定是一个 `MyItem` 对象)

C. The `dynamic_cast` ought to be a `reinterpret_cast` since the concrete type is unknown. (`dynamic_cast` 应该是 `reinterpret_cast`，因为具体类型不可知)

D. The `dynamic_cast` is redundant, the programmer can invoke `Activate` directly, e.g. `ws->Selection()->Activate();` (`dynamic_cast` 是多余的，程序员可以直接激活代码，比如 `ws->Selection()->Activate()`)

**解析：** C++有4个类型转换操作符，这4个操作符是 `static_cast`、`const_cast`、`dynamic_cast` 和 `reinterpret_cast`。

例如，假设你想把一个 `int` 转换成 `double`，以便让包含 `int` 类型变量的表达式产生出浮点

数值的结果。你应该这样写：

```
int firstNumber, secondNumber;
double result = static_cast<double>(firstNumber) / secondNumber;
```

这样的类型转换不论是对人工还是对程序都很容易识别。

在 C++ 中，`static_cast` 在功能上相对 C 语言来说有所限制。如不能用 `static_cast` 像用 C 风格的类型转换一样把 `struct` 转换成 `int` 类型，或者把 `double` 类型转换成指针类型；另外，`static_cast` 不能从表达式中去除 `const` 属性，因为另一个新的类型转换操作符 `const_cast` 有这样的功能。

其他 C++ 类型转换操作符被用在需要更多限制的地方。`const_cast` 最普通的用途就是转换掉对象的 `const` 属性。通过使用 `const_cast`，让编译器知道通过类型转换想做的只是改变一些东西的 `constness` 或者 `volatility` 属性。这个含义被编译器所约束。如果你试图使用 `const_cast` 来完成修改 `constness` 或者 `volatility` 属性之外的事情，你的类型转换将被拒绝。

下面是一个 `const_cast` 的例子：

```
#include <iostream>
#define MAX 255
typedef short Int16;
using namespace std;
class B{
public:
    int m_iNum;
};
int main(){
    B b0;
    b0.m_iNum = 100;
    const B b1 = b0;
    // 这样写会编译失败 因为 b1.m_iNum 是常量对象，不能对它进行改变
    // b1.m_iNum = 100;
    // 输出 100 100
    cout << b0.m_iNum << " " << b1.m_iNum << " " << endl;
    const_cast<B&>(b1).m_iNum = 200;
    // 去除 b1 的 const 属性后重新赋值 200
    // 输出 100 200
    cout << b0.m_iNum << " " << b1.m_iNum << " " << endl;
    return 0;
}
```

`static_cast` 和 `reinterpret_cast` 操作符修改了操作数类型。它们不是互逆的；`static_cast` 在编译时使用类型信息执行转换，在转换执行必要的检测（诸如指针越界计算，类型检查），其操作数相对是安全的。另一方面，`reinterpret_cast` 仅仅是重新解释了给出的对象的比特模型而没有进行二进制转换，编译器隐式执行任何类型转换都可由 `static_cast` 显示完成，`reinterpret_cast` 通常为操作数的位模式提供较低层的重新解释。例子如下：

```
int n=9; double d=static_cast<double>(n);
cout << n << " " << d; //输出 9 9
```

上面的例子中，我们将一个变量从 int 转换到 double。这些类型的二进制表达式是不同的。要将整数 9 转换到双精度整数 9，static\_cast 需要正确地为双精度整数 d 补足比特位。其结果为 9.0。而 reinterpret\_cast 的行为却不同：

```
int n=9;
double d=reinterpret_cast<double &>(n);
cout << n << " " << d; //输出 9 5.28421e-308
```

在进行计算以后，d 包含无用值。这是因为 reinterpret\_cast 仅仅是复制 n 的比特位到 d，没有进行必要的分析。reinterpret\_cast 这个操作符被用于的类型转换的转换结果几乎都是实现时定义（implementation-defined）。因此，使用 reinterpret\_casts 的代码很难移植。转换函数指针的代码是不可移植的，（C++不保证所有的函数指针都被用一样的方法表示），在一些情况下这样的转换会产生不正确的结果。所以应该避免转换函数指针类型，按照 C++新思维的话来说，reinterpret\_cast 是为了映射到一个完全不同类型的意思，这个关键词在我们需要把类型映射回原有类型时用到它。我们映射到的类型仅仅是为了故弄玄虚和其他目的，这是所有映射中最危险的。reinterpret\_cast 就是一把锐利无比的双刃剑，除非你处于背水一战和火烧眉毛的危急时刻，否则绝不能使用。

对于 dynamic\_cast 要注意以下 4 点：

- dynamic\_cast 是在运行时检查的，dynamic\_cast 用于在继承体系中进行安全的向下转换 downcast（当然也可以向上转换，但是没必要，因为完全可以用虚函数实现），即基类指针/引用到派生类指针/引用的转换。如果源和目标类型没有继承/被继承关系，编译器会报错；否则必须在代码里判断返回值是否为 NULL 来确认转换是否成功。
- dynamic\_cast 不是扩展 C++ 中 style 转换的功能，而是提供了类型安全性。你无法用 dynamic\_cast 进行一些“无理”的转换。
- dynamic\_cast 是 4 个转换中唯一的 RTTI 操作符，提供运行时类型检查。
- dynamic\_cast 不是强制转换，而是带有某种“咨询”性质的。如果不能转换，dynamic\_cast 会返回 NULL，表示不成功。这是强制转换做不到的。

下面是一个例子：

```
using namespace std;
class B{};
class C:public B {};
class D:public C {};

int main(){
    D* pd=new D;
    C* pc=dynamic_cast<C*>(pd);
```

```
B* pb=dynamic_cast<B*>(pd);
//C* pc=pd;
//B* pb=pd;
void *p=dynamic_cast<C*>(pd);
//void*p=pd;
return 0;
}
```

在本题中，MyItem 与 IGlyph 是继承关系，可以适用 dynamic\_cast 类型转换，而因为我们不知道确定的 IWidgetSelector::Selection()返回的具体类型是什么，所以应用 dynamic\_cast “试探性”地进行类型转换是十分必要的。

答案：A

**面试例题 7：**阅读下面程序，下列选项说法正确的是哪项？[德国某计算机软件公司 2009 年 12 月面试题]

```
#include <iostream>
#include <string>
using namespace std;
class A
{
public:
//标识1
    virtual void foo(){cout <<"A foo"
<<endl;}
    void pp(){cout <<"A PP" <<endl;}
};
class B:public A
{
public:
    void foo(){cout <<"B foo" <<endl;}
    void pp(){cout <<"B PP" <<endl;}
}
```

```
void FunctionB(){cout <<"Excute
FunctionB!" <<endl;}
int main()
{
    A a;    A *pa=&a;
    pa->foo();
    pa->pp();
//标识2
    (dynamic_cast <B*>(pa))->FunctionB();
//标识3
    (dynamic_cast <B*>(pa))->foo();
    (dynamic_cast <B*>(pa))->pp(); //标识4
    (*pa).foo();
    return 0;
}
```

- A. 标识 1 处有问题，Class A 不应该调用虚函数。
- B. 标识 2 处有问题，(dynamic\_cast <B\*>(pa))->FunctionB();无法运行通过。
- C. 标识 3 处有问题，(dynamic\_cast <B\*>(pa))->foo();无法运行通过。
- D. 标识 4 处有问题，(dynamic\_cast <B\*>(pa))->pp();无法运行通过。

因为 a 是基类对象，所以 dynamic\_cast <B\*>(pa) 将返回空指针。

**解析：**在上面的代码段中，如果 pa 指向一个 B 类型的对象，对这种情况执行任何操作都是安全的；但是，实际上 pa 指向的是一个 A 类型的对象，那么(dynamic\_cast <B\*>(pa))返回值将是一个空指针，所以题目中的代码：

```
(dynamic_cast <B*>(pa))->FunctionB(); //标识2
(dynamic_cast <B*>(pa))->foo(); //标识3
(dynamic_cast <B*>(pa))->pp(); //标识4
```

与下列代码：

```
B *somenull = NULL;
somenull->FunctionB();
somenull->foo();
somenull->pp();
```

没有任何区别，之所以标识 2 和标识 4 可以运行通过，是因为 FunctionB 和 pp 函数未使

用任何成员数据，也不是虚函数，不需要 this 指针，也不需要动态绑定，可正常运行。

而`(dynamic_cast<B*>(pa))->foo();`将导致程序崩溃，因为调用了虚函数，编译器需要根据对象的虚函数指针查找虚函数表，但此时是空，为非法访问。

如果将 A a；改为 B b；就可正常运行了，正确代码如下：

```
#include <iostream>
#include <string>
using namespace std;
class A
{
public:
    virtual void foo(){cout <<"A foo" <<endl;}
    //虚函数的出现会带来动态机制 Class A 至少要有一个虚函数
    void pp(){cout <<"A PP" <<endl;}
};

class B:public A
{
public:
    void foo(){cout <<"B foo" <<endl;}
    void pp(){cout <<"B PP" <<endl;}
    void FunctionB(){cout <<"Excute FunctionB!" <<endl;}
};

int main()
{
    B b;    A *pa=&b;
    pa->foo(); //由于存在多态，因此调用 B::foo()
    pa->pp(); //调用 A::pp()
    // (pa)->FunctionB();
    // 编译错误：'FunctionB' : is not a member of 'A'
    // 这里不能直接调用 pa 的 FunctionB 功能，因为没有声明
    (dynamic_cast<B*>(pa))->FunctionB();
    // 运行时动态的把 pa 由 class A 转换成 class B 使 pa 可以执行 FunctionB()

    (dynamic_cast<B*>(pa))->foo();
    // 运行时动态的把 pa 由 class A 转换成 class B 使 pa 可以执行 Class B 的 foo()
    // 这里会调用虚函数，编译器需要根据对象的虚函数指针查找虚函数表
    (dynamic_cast<B*>(pa))->pp();
    // 运行时动态的把 pa 由 class A 转换成 class B 使 pa 可以执行 Class B 的 pp()
    (*pa).foo(); //调用 A::foo()

    return 0;
}
```

答案：C

**面试例题 8：**下面程序的运行结果是什么？[美国某著名计算机软硬件公司面试题]

```
#include <iostream>
using namespace std;
int main()
{
    unsigned short int i = 0;
    int j = 8,p;
```

```
p = j << 1;
i = i - 1;
cout <<"\n i = " << i ;
cout <<"\n p = " << p ;
return 0;
```

**解析：**此题有两个考点，一是用最有效率的方法算出 2 乘以 8 等于几，二是无符号结果问题。

在这里，8 左移一位就是  $8 \times 2$  的结果 16。移位运算是最有效率的计算乘/除算法的运算之一。在 unsigned short int 中无符号的 -1 的结果等于 65 535。

**答案：**16, 65535

**面试例题 9：**建立一个联合体，由 char 类型和 int 类型组成。下面的程序运行结果是什么？

```
#include <iostream>
using namespace std;

union {
    unsigned char a;
    unsigned int i;
} u;

int main() {
    u.i=0xf0f1f2f3;
    cout<<hex<<u.i<<endl;
    cout<<hex<<int(u.a)<<endl;
    return 0;
}
```

**解析：**内存中数据的排列问题。

**答案：**

运行上面程序后，输出为：

```
f0f1f2f3
f3
```

这说明，内存中数据低位字节存入低地址，高位字节存入高地址，而数据的地址采用它的低地址来表示。

**面试例题 10：**嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量 a，写两段代码，第一个设置 a 的 bit 3，第二个清除 a 的 bit 3。在以上两个操作中，要保持其他位不变。

**解析：**被面试者对这个问题有 3 种基本的反应。

一种是不知道如何下手。显然该被面试者从没做过任何嵌入式系统的工作。

还有一种是用 bit fields。bit fields 是被扔到 C 语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了你的代码是不可重用的。

还有一种是用#define 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。

一些人喜欢为设置和清除值而定义一个掩码，同时定义一些说明常数，这也是可以接受的。面试官希望看到的几个要点为说明常数、“|=”和“&=”操作。

**答案：**

最佳的解决方案如下：

```
#define BIT3 (0x1 << 3)
static int a;
void set_bit3(void) {
    a |= BIT3;
}
```

```
void clear_bit3(void) {
    a &= ~BIT3;
}
```

**面试例题 11：**阅读以下代码，写出程序运行结果。[中国著名杀毒软件企业 J 公司 2008 年 4 月面试题]

```
# include <iostream>
# include <string>

using namespace std;

int main()
```

```
{
    int *pa = NULL;
    int *pb = pa + 15;
    printf("%x", pb);
    return 0;
}
```

**解析：**  $15 \times 4$  (字节) =60

所以要求输出的十六进制结果是 3C。这样在大系统里面使用未初始化的指针是很危险的。

**答案：** 3C

## 12.2 嵌入式编程

**面试例题 1：** Interrupts are an important part of embedded systems. Consequently, many compiler vendors offer an extension to standard C to support interrupts. Typically, the keyword is \_interrupt. The following routine(ISR). Point out the errors in the code. (中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展——让标准 C 支持中断。其代表事实是，产生了一个新的关键字\_interrupt。请看下面的程序（一个中断服务子程序 ISR），请指出这段代码的错误。) [中国台湾某著名 CPU 生产公司 2005 年面试题]

```
interrupt double compute_area
    (double radius)
{
    double area= PI*radius*radius;
```

```
    printf("\nArea=%f", area);
    return area;
}
```

**解析：** 嵌入式编程问题。

**答案：**

- (1) ISR 不能返回一个值。如果你不懂这个，那么是不会被雇用的。
- (2) ISR 不能传递参数。如果你没有看到这一点，被雇用的机会等同第一项。
- (3) 在许多处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额

外的寄存器入栈，有些处理器/编译器就不允许在 ISR 中做浮点运算。此外，ISR 应该是短而有效率的，在 ISR 中做浮点运算是不明智的。

(4) 与第三点一脉相承，printf()经常有重入和性能上的问题，所以一般不使用 printf()。

**面试例题 2：**In embedded system, we usually use the keyword “volatile”, what does the keyword mean? (在嵌入式系统中，我们经常使用“volatile”这个关键字，它是什么意思？) [中国台湾某著名 CPU 生产公司 2005 年面试题]

**解析：**volatile 问题。

当一个对象的值可能会在编译器的控制或监测之外被改变时，例如一个被系统时钟更新的变量，那么该对象应该声明成 volatile。因此编译器执行的某些例行优化行为不能应用在已指定为 volatile 的对象上。

volatile 限定修饰符的用法与 const 非常相似——都是作为类型的附加修饰符。例如：

```
volatile int display_register;
volatile Task *curr_task;
volatile int ixa[ max_size ];
volatile Screen bitmap_buf;
```

display\_register 是一个 int 型的 volatile 对象；curr\_task 是一个指向 volatile 的 Task 类对象的指针；ixa 是一个 volatile 的整型数组，数组的每个元素都被认为是 volatile 的；bitmap\_buf 是一个 volatile 的 Screen 类对象，它的每个数据成员都被视为 volatile 的。

volatile 修饰符的主要目的是提示编译器该对象的值可能在编译器未监测到的情况下被改变，因此编译器不能武断地对引用这些对象的代码做优化处理。

**答案：**

volatile 的语法与 const 是一样的，但是 volatile 的意思是“在编译器认识的范围外，这个数据可以被改变”。不知什么原因，环境正在改变数据（可能通过多任务处理），所以，volatile 告诉编译器不要擅自做出有关数据的任何假定——在优化期间这是特别重要的。如果编译器说：“我已经把数据读进寄存器，而且再没有与寄存器接触。”在一般情况下，它不需要再读这个数据。但是，如果数据是 volatile 修饰的，编译器则不能做出这样的假定，因为数据可能被其他进程改变了，编译器必须重读这个数据而不是优化这个代码。

就像建立 const 对象一样，程序员也可以建立 volatile 对象，甚至还可以建立 const volatile 对象。这个对象不能被程序员改变，但可通过外面的工具改变。

**面试例题 3：**关键字 const 有什么含意？下面的声明都是什么意思？

```
const int a;
int const a;
```

```
const int *a;
int * const a;
int const * a const;
```

**解析：**只要一听到被面试者说“`const` 意味着常数”，面试官就知道自己正在和一个业余者打交道。因为 ESP (Embedded Systems Programming, 嵌入式系统编程) 的每一位求职者都应该非常熟悉 `const` 能做什么和不能做什么。正确的说法是能说出 `const` 意味着“只读”就可以了。尽管这个答案不是完全的答案，但面试官可以接受它为一个正确的答案。

关键字 `const` 的作用是为读你代码的人传达非常有用的信息。实际上，声明一个参数为常量是为了告诉用户这个参数的应用目的。如果你曾花很多时间清理其他人留下的垃圾，你就会很快学会感谢这点儿多余的信息。当然，懂得用 `const` 的程序员很少会留下垃圾让别人来清理。通过给优化器一些附加的信息，使用关键字 `const` 也许能产生更紧凑的代码。

合理地使用关键字 `const` 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少 bug 的出现。

**答案：**前两个的作用是一样的。`a` 是一个常整型数(不可修改值的整型数)。第三个即 `const int * a;`，是一个指向 `const` 对象的指针，即指针本身不是 `const` 的，可以被修改，但它指向的对象，即这个整形数，可以被修改，只不过不允许通过这个指针去修改这个对象的值。这里举个最简单的例子：`a` 是一个指向 `const` 对象的指针，但它指向一个非 `const` 的 `int` 型对象 `b`。这个 `b` 的整型值是可以被修改的，只不过不能通过 `a` 这个指针去修改它，否则会报错。代码如下所示：

```
const int* a;
int b=30;
a=&b;
cout<<*a<<endl; /*a=30
b=40;
cout<<*a<<endl; /*a=40
*a=50;//error: you cannot assign to a variable that is const
```

第四个的意思是 `a` 是一个指向整型数的常指针(也就是说，指针指向的整型数可以修改，但指针不可以修改)。最后一个意味着 `a` 是一个指向常整型数的常指针(也就是说，指针指向的整型数不可以修改，同时指针也不可以修改)。

**面试例题 4：**关键字 `volatile` 有什么含意？并给出 3 个不同的例子。[中国台湾某著名计算机硬件公司面试题]

**解析：**回答不出这个问题的人是不会被雇用的。我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。搞嵌入式的家伙们经常同硬件、中断、RTOS 等打交道，所有这些都要求用到 `volatile` 变量。不懂得 `volatile` 的内容将会带来灾难。

**答案：**一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子：

- 并行设备的硬件寄存器（如状态寄存器）。
- 一个中断服务子程序中会访问到的非自动变量（Non-automatic variables）。
- 多线程应用中被几个任务共享的变量。

**面试例题 5：**一个参数可以既是 const 又是 volatile 吗？一个指针可以是 volatile 吗？解释为什么。

**答案：**

第一个问题：可以。一个例子就是只读的状态寄存器。它是 volatile，因为它可能被意想不到地改变；它又是 const，因为程序不应该试图去修改它。

第二个问题：可以。尽管这并不很常见。一个例子是当一个中断服务子程序修改一个指向一个 buffer 的指针时。

**面试例题 6：**下面的函数有什么错误？

```
int square(volatile int *ptr)
{
    return *ptr * *ptr;
}
```

**解析：**这段代码的目的是用来返还指针\*ptr 指向值的平方，但是，由于\*ptr 指向一个 volatile 型参数，编译器将产生类似下面的代码：

```
int square(volatile int *ptr)
{
    int a,b;
    a = *ptr;
    b = *ptr;
    return a * b;
}
```

由于\*ptr 的值可能被意想不到地改变，因此 a 和 b 可能是不同的。结果，这段代码可能无法返回你所期望的平方值。

**答案：**

正确的代码如下：

```
long square(volatile int *ptr)
{
    int a;
    a = *ptr;
    return a * a;
```

}

**面试例题7：**嵌入式系统经常具有要求程序员去访问某特定位置的内存的特点。在某工程中，要求设置一绝对地址为0x67a9的整型变量的值为0xaa55。编译器是一个纯粹的ANSI编译器。写代码去完成这一任务。

**解析：**这一问题测试你是否知道为了访问一个绝对地址把一个整型数强制转换(typecast)为一个指针是合法的。这一问题的实现方式随着个人风格不同而不同。典型的代码如下：

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
```

一个较晦涩的方法是：

```
* (int * const) (0x67a9) = 0xaa55;
```

建议你在面试时使用第一种方案。

**答案：**

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
```

**面试例题8：**评价下面的代码片断，找出其中的错误。

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

**解析：**这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在笔者的眼中，好的嵌入式程序员应该非常准确地明白硬件的细节和它的局限，然而PC程序往往把硬件作为一个无法避免的烦恼。对于一个int型且不是16位的处理器来说，上面的代码是不正确的。

**答案：**应编写如下代码：

```
unsigned int compzero = ~0;
```

**面试例题9：**下面的代码片段的输出是什么？为什么？

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL)
    puts("Got a null pointer");
else
    puts("Got a valid pointer");
```

**解析：**这是一道动态内存分配(Dynamic memory allocation)题。

尽管不像非嵌入式计算那么常见，嵌入式系统还是有从堆(heap)中动态分配内存的过程。

面试官期望应试者能解决内存碎片、碎片收集、变量的执行时间等问题。

这是一个有趣的问题。故意把 0 值传给了函数 malloc，得到了一个合法的指针，这就是上面的代码，该代码的输出是“Got a valid pointer”。我用这个来讨论这样的一道面试例题，看看被面试者是否能想到库例程这样做是正确的。得到正确的答案固然重要，但解决问题的方法和你做决定的基本原理更重要。

将程序修改成：

```
char *ptr;
if (int pp = (strlen(ptr = (char *)
    malloc(0))) == 0)
```

```
puts("Got a null pointer");
else
    puts("Got a valid pointer");
```

或者：

```
char *ptr;
if (int pp = (sizeof(ptr = (char *)
    malloc(0))) == 4)
```

```
puts("Got a null pointer");
else
    puts("Got a valid pointer");
```

如果求 ptr 的 strlen 值和 sizeof 值，该代码的输出是“Got a null pointer”。

答案：Got a valid pointer。

**面试例题 10：**In little-endian systems, what is the result of following C program? (在小尾字节系统中，这段 C 程序的结果是多少?)

```
typedef struct bitstruct{
    int b1:5;
    int :2;
    int b2:2;
}bitstruct;
```

```
void main(){
    bitstruct b;
    memcpy(&b, "EMC EXAMINATION", sizeof(b));
    printf("%d,%d\n", b.b1, b.b2);
}
```

**解析：**“Endian”这个词出自《格列佛游记》。小人国的内战就源于吃鸡蛋时是究竟从大头 (Big-Endian) 敲开还是从小头 (Little-Endian) 敲开，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位。

我们一般将 Endian 翻译成“字节序”，将 big Endian 和 little Endian 称作“大尾”和“小尾”。Little-Endian 主要用在我们现在的 PC 的 CPU 中，Big-Endian 则应用在目前的 Mac 机器中（注意：是指 Power 系列 处理器）

嵌入式系统开发者应该对 Little-endian 和 Big-endian 模式非常了解。采用 Little-endian 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 Big-endian 模式对操作数的存放方式是从高字节到低字节。例如，16bit 宽的数 0x1234 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	存放内容
0x4000	0x34
0x4001	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则为：

内存地址	存放内容
0x4000	0x12
0x4001	0x34

32bit 宽的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式(假设从地址 0x4000 开始存放) 为：

内存地址	存放内容
0x4000	0x78
0x4001	0x56
0x4002	0x34
0x4003	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则为：

内存地址	存放内容
0x4000	0x12
0x4001	0x34
0x4002	0x56
0x4003	0x78

在本题中 E 的 ASCII 值是 0x45( 0100 0101 ), M 是 0x4D( 0100 1101 ), 它们在内存中的表现如下：

```
1010 0010 1011 0010
----E---- ----M----
```

结构 bitstruct 是 9 位的，执行 copy 以后，b 在内存中如下：

```
1010 0010 1
```

b1 占 5 位：

```
1010 0
```

中间跳过 2 位， b2 占 2 位：

```
01
```

计算的时候再把它们逆转过来, 就成了下面的形式：

```
b1: 00101
b2: 10
```

b1 最高位是 0，表示其是正数，其原码跟补码一致，所以  $b1 = 2^0 + 2^2 = 5$ 。

b2 最高位是 1，表示其是负数，其原码要进行取反操作再加 1 为 10 所以  $b2 = -(2^1) = -2$ 。

答案：5, -2

**面试例题 11：**在某些极端要求性能的场合，我们需要对程序进行优化，关于优化，以下说法正确的是：

- A. 将程序整个用汇编语言改写会大大提高程序性能。
- B. 在优化前，可以先确定哪部分代码最为耗时，然后对这部分代码使用汇编语言编写。使用的汇编语句数目越少，程序就运转越快。
- C. 使用汇编语言虽然可能提高了程序性能，但是降低了程序的可移植性和可维护性，所以应该绝对避免。
- D. 适当调整汇编指令的顺序，可以缩短程序运行的时间。

**解析：**AC 说法都过于绝对了。至于 B 也是错的，不同的架构有不同的流水线方式，arm9 中的流水线对汇编的顺序要求很高，不然会浪费指令周期，有时候甚至用 nop 填充。

**答案：**D

**面试例题 12：**使用 C 语言将一个 1GB 的字符数组从头到尾全部设置为字符“A”，在一台典型的当代 PC 上，需要花费的 CPU 时间的数量级最接近：

- A. 0.001 秒
- B. 1 秒
- C. 100 秒
- D. 2 小时

**解析：**1GB 需要 1G 条指令，如 4 核 2GB 的 cpu，如 1 周期 1 条指令，需要 0.25 秒，所以最接近 1 秒。

**答案：**B

**面试例题 13：**十进制数 -10 的三进制 4 位数补码形式是\_\_\_\_\_：

- A. 0101
- B. 1010
- C. 2121
- D. 2122

**解析：**对于负数的补码：对于二进制而言， $-10$  的补码为  $-(2^8 - |-10|) = -(256 - 10) = -246 = 11110110$ 。

同理，对于 4 位三进制的补码： $10$  的补码为  $3^4 - |-10| = 71 = 2212$ 。

**答案：**D

## 12.3 static

**面试例题 1：**关键字 static 的作用是什么？

**解析：**这个简单的问题很少有人能回答完全。大多数应试者能正确回答第一部分，一部分能正确回答第二部分，但是很少的人能懂得第三部分。这是一个应试者的严重的缺点，因为他显然不懂得本地化数据和代码范围的好处和重要性。

**答案：**在 C 语言中，static 关键字至少有下列几个作用：

- 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有函数访问，但不能被模块外其他函数访问。
- 在模块内的 static 函数只可被这一模块内的其他函数调用，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复制。
- 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。

**面试题目 2：**写出下面程序的运行结果。

```
int sum(int a)
{
    auto int c=0;
    static int b=3;
    c+=1;
    b+=2;
    return(a+b+c);
}
```

```
void main()
{
    int I;
    int a=2;
    for(I=0;I<5;I++)
    {
        printf("%d, ", sum(a));
    }
}
```

**解析：**在求和函数 sum 里面 c 是 auto 变量，根据 auto 变量特性得知，每次调用 sum 函数时变量 c 都会自动赋值为 0。b 是 static 变量，根据 static 变量特性得知，每次调用 sum 函数时变量 b 都会使用上次调用 sum 函数时 b 保存的值。

简单地分析一下函数，可以知道，若传入的参数不变，则每次调用 sum 函数返回的结果，都比上次多 2。所以答案是：8,10,12,14,16。

**答案：**8,10,12,14,16。

### 第3部分

## 数据结构和设计模式

**Data structure and design pattern**

**本**部分主要介绍求职面试过程中出现的第二个重要的板块——数据结构，包括字符串的使用、堆、栈、排序方法等。此外，随着外企研发机构大量内迁我国，在外企的面试中，软件工程的知识，包括设计模式、UML、敏捷软件开发，以及.NET技术和完全面向对象语言C#的面试题目将会有增无减。关于设计模式的题目在今后的面试中的比重会更加扩大。

# 第 13 章

## 数据结构基础

**面**试时间一般有2个小时，其中至少有20~30分钟时间是用来回答数据结构相关问题的。而由于链表是一种相对简单的数据结构，容易引起面试官多次反复发问。此外，数组的排序和逆置也是面试官必考的内容之一。事实上，单链表的复杂程度并不亚于树、图等复杂数据结构。面试官完全有可能构造出极富挑战性的试题。最后一个考点是堆栈，面试官会结合程序对你的思维能力进行考量。

### 13.1 单链表

**面试例题 1：**编程实现一个单链表的建立/测长/打印。[日本某著名家电/通信/IT 企业面试题]

**答案：**

完整代码如下：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>
using namespace std;

typedef struct student
{
    int data;
    struct student *next;
}node;
node *creat()
{
    node *head,*p,*s;
    int x,cycle=1;
    head=(node*)malloc(sizeof(node));
    p=head;
    while(cycle)
```

```
{ 
    printf("\nplease input the data:");
    scanf("%d",&x);
    if(x!=0)
    {
        s=(node *)malloc(sizeof(node));
        s->data=x;
        printf("\n%d",s->data);
        p->next=s;
        p=s;
    }
    else cycle=0;
}
head=head->next;
p->next=NULL;
printf("\n    %d",head->data);
return(head); }
```

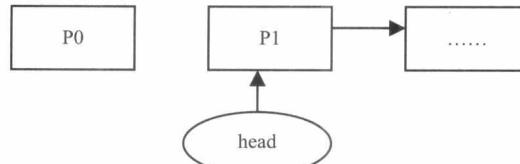
```
//单链表测长
int length(node *head)
{
    int n=0;
    node *p;
    p=head;
    while(p!=NULL)
    {
        p=p->next;
        n++;
    }
    return(n);
}
//单链表打印
```

```
void print(node *head)
{
    node *p;int n;
    n=length(head);
    printf("\nNow, These %d records are :\n",n);
    p=head;
    if(head!=NULL)

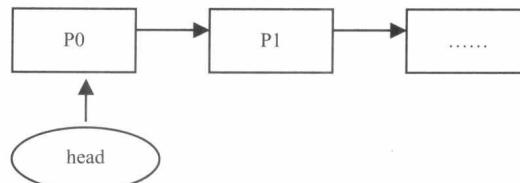
        while(p!=NULL)
        {
            printf("\n      uuu %d      ",p->data);
            p=p->next;
        }
}
```

**面试例题 2：**编程实现单链表删除节点。[美国某著名分析软件公司面试题]

**解析：**单链表的插入，如下图所示。



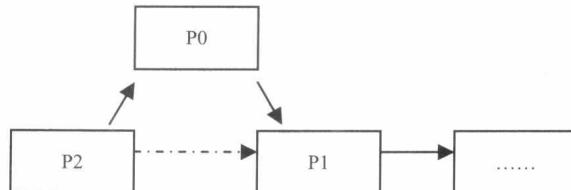
如果插入在头节点以前，则 p0 的 next 指向 p1，头节点指向 p0，如下图所示。



如果插入中间节点，如下图所示。



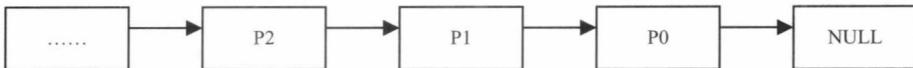
则先让 p2 的 next 指向 p0，再让 p0 指向 p1，如下图所示。



如果插入尾节点，如下图所示。



则先让 p1 的 next 指向 p0，再让 p0 指向空，如下图所示。



**答案：**

完整代码如下：

```

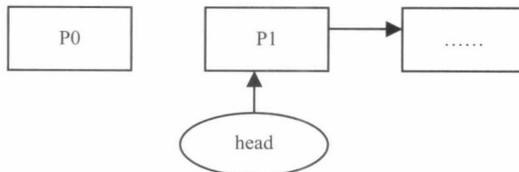
node *del(node *head, int num)
{
    node *p1,*p2;
    p1=head;
    while(num!=p1->data&&p1->next!=NULL)
        {p2=p1;p1=p1->next;}

    if(num==p1->data)
    {
        if(p1==head)
            {head=p1->next;
             free(p1);}

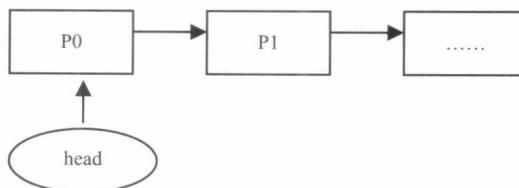
        else
            p2->next=p1->next;
    }
    else
        printf("\n%d could not been found",
               num);
    return(head);
}
  
```

**面试例题 3：**编写程序实现单链表的插入。[美国某著名计算机嵌入式公司 2005 年面试题]

**解析：**单链表的插入，如下图所示。



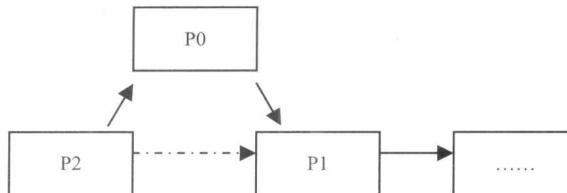
如果插入在头节点以前，则 p0 的 next 指向 p1，头节点指向 p0，如下图所示。



如果插入中间节点，如下图所示。



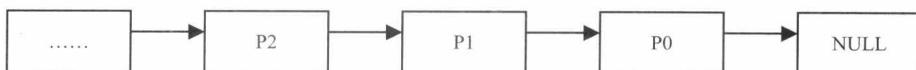
则先让 p2 的 next 指向 p0，再让 p0 指向 p1，如下图所示。



如果插入尾节点，如下图所示。



则先让 p1 的 next 指向 p0，再让 p0 指向空，如下图所示。



**答案：**完整代码如下：

```

node *insert(node *head, int num)
{
    node *p0, *p1, *p2;
    p1 = head;
    p0 = (node *)malloc(sizeof(node));
    p0->data = num;
    while (p0->data > p1->data && p1->next != NULL)
        {p2 = p1; p1 = p1->next;}
    if (p0->data <= p1->data)
    {
        if (head == p1)
            {p0->next = p1;
  
```

```

head = p0;
}
else
{
    p2->next = p0;
    p0->next = p1;
}
else
{
    p1->next = p0; p0->next = NULL;
}
return (head);
}
  
```

**面试例题 4：**编程实现单链表的排序。

**答案：**

完整代码如下：

```

node *sort(node *head)
{
    node *p, *p2, *p3;
    int n; int temp;
    n = length(head);
    if (head == NULL || head->next == NULL)
        return head;
    p = head;
    for (int j = 1; j < n; ++j)
    {
        p = head;
        for (int i = 0; i < n - j; ++i)
  
```

```

        {
            if (p->data > p->next->data)
            {
                temp = p->data;
                p->data = p->next->data;
                p->next->data = temp;
            }
            p = p->next;
        }
    }
  
```