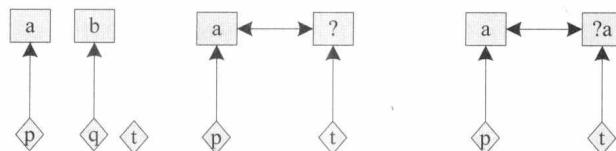


指针。

这里要注意：

```
int *temp;
*temp=*p;
```

是不符合逻辑的一段代码，`int *temp` 新建了一个指针（但没有分配内存）。`*temp=*p` 不是指向而是复制。把`*p` 所指向的内存里的值（也就是实参 `a` 的值）复制到`*temp` 所指向内存里了。但是 `int *temp` 不是不分配内存吗？的确不分配，于是系统在复制时临时给了一个随机地址，让它存值。分配的随机地址是个“意外”，且函数结束后不收回，造成内存泄漏，如下图所示。



那么 `swap2` 到底能否实现两数交换吗？这要视编译器而定，笔者在 Dev-C++ 可以通过测试，但是在更加“严格”的编译器如 vs2008，这段代码会报错。

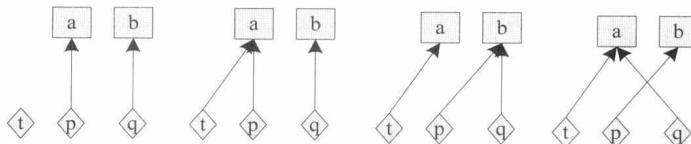
`swap3` 传的是一个地址进去，在函数体内的形参`*p`、`*q` 是指向实际参数 `a`、`b` 地址的两个指针。这里要注意：

```
int *temp;
temp=p;
```

`int *temp` 新建了一个指针（但没有分配内存）。`temp=p` 是指向而不是复制。`temp` 指向了`*p` 所指向的地址（也就是 `a`）。而代码：

```
p=q;
q=temp;
```

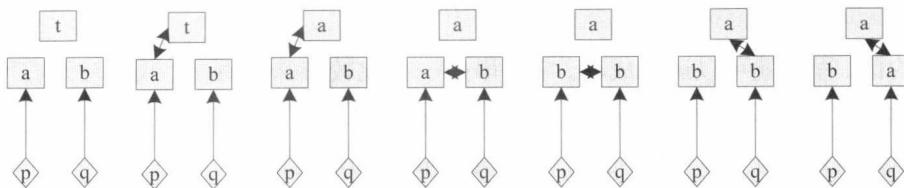
意思是 `p` 指向了`*q` 所指向的地址（也就是 `b`）。`q` 指向了`*t` 所指向的地址（也就是 `a`），如下图所示。



但是函数 `swap3` 不能实现两数的交换，这是因为函数体内只是指针的变化，而对地址中的值却没有改变。举个简单的例子，`a`、`b` 两个仓库的两把备用钥匙 `p`、`q`，`p` 钥匙用来开 `a` 仓库，`q` 钥匙用来开 `b` 仓库。现在进入函数体，`p`、`q` 钥匙功能发生了改变：`p` 钥匙用来开 `b` 仓

库，`q` 钥匙用来开 `a` 仓库；但是仓库本身的货物没有变化（`a` 仓库原来是韭菜现在还是韭菜，`b` 仓库原来是番薯现在还是番薯）。当函数结束，`p`、`q` 两把备用钥匙自动销毁。主函数里用主钥匙打开 `a`、`b` 两个仓库，发现值还是没有变化。

函数 `swap4` 可以实现两数的交换，因为它修改的是指针所指向地址中的值，如下图所示。



`swap5` 函数与 `swap4` 类似，是一个引用传递，修改的结果直接影响实参。

答案：`swap4` 函数和 `swap5` 函数。

面试例题 2：What will happen after running the “Test”？(这个程序测试后会有什么结果？) [美国某著名计算机嵌入式公司 2005 年 9 月面试题]

```
#include <iostream>

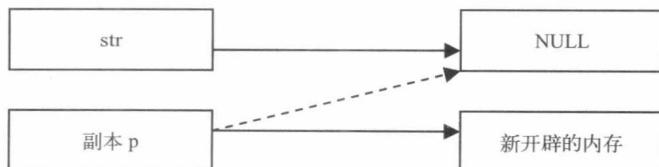
void GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}

int main()
{
```

```
    char *str = NULL;

    GetMemory(str,100);
    strcpy(str,"hello");
    return 0;
}
```

解析：毛病出在函数 `GetMemory` 中。`void GetMemory(char *p, int num)` 中的`*p` 实际上是主函数中 `str` 的一个副本，编译器总是要为函数的每个参数制作临时副本。在本例中，`p` 申请了新的内存，只是把 `p` 所指的内存地址改变了，但是 `str` 丝毫未变。因为函数 `GetMemory` 没有返回值，因此 `str` 并不指向 `p` 所申请的那段内存，所以函数 `GetMemory` 并不能输出任何东西，如下图所示。事实上，每执行一次 `GetMemory` 就会申请一块内存，但申请的内存却不能有效释放，结果是内存一直被独占，最终造成内存泄漏。



如果一定要用指针参数去申请内存，那么应该采用指向指针的指针，传 `str` 的地址给函数 `GetMemory`。代码如下：

```
#include <iostream>
void GetMemory(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}
int main()
{
    char *str = NULL;
}
```

```
GetMemory(&str,100);
strcpy(str,"hello");
cout << *str << endl;
cout << str << endl;
cout << &str << endl;
return 0;
}
```

这样的话，程序就可以运行成功。字符串是一个比较特殊的例子。我们分别打印*str、str、&str 可以发现，结果分别是 h、hello、0*22f7c。str 就是字符串的值；*str 是字符串中某一字符的值，默认的是首字符，所以是 h；&str 是字符串的地址值。

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，代码如下：

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

char *GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
    return p;
}
int main()
```

```
{
    char *str = NULL;

    str=GetMemory(str,100);
    strcpy(str,"hello");
    return 0;
}
```

我们可以对这道题推而广之，看一下整型变量是如何传值的，代码如下：

```
#include <iostream>
using namespace std;
void GetMemory2(int *z)
{
    *z=5;
}
int main()
```

```
{
    int v;
    GetMemory2(&v);
    cout << v << endl;
    return 0;
}
```

GetMemory2 把 v 的地址传了进来，*z 是地址里的值，是 v 的副本。通过直接修改地址里的值，不需要有返回值，也把 v 给修改了，因为 v 所指向地址的值发生了改变。

答案：程序崩溃。因为 GetMemory 并不能传递动态内存，Test 函数中的 str 一直都是 NULL。

面试例题 3：这个函数有什么问题？该如何修改？[美国著名硬盘公司 S 2008 年 4 月面试题]

```
char *strA()
{
    char str[] = "hello word";
```

```
    return str;
}
```

解析：这个 str 里存的地址是函数 strA 栈帧里“hello word”的首地址。函数调用完成，栈帧恢复到调用 strA 之前的状态，临时空间被重置，堆栈“回缩”，strA 栈帧不再属于应该

访问的范围。存于 strA 栈帧里的“hello word”当然也不应该访问了。这段程序可以正确输出结果，但是这种访问方法违背了函数的栈帧机制。

分配内存时有一句老话：First time you do it, then something change it（一旦使用，它即改变）。也许是一个妨碍其他函数调用的内存块，这些情况都是无法预知的。如果运行一段函数，不会改变其他函数所调用的内存，在这种情况下，你运行多少次都不是问题。

但是只要另外一个函数调用的话，你就会发现，这种方式的不合理及危险性。我们面对的是一个有操作系统覆盖的计算机，而一个不再访问的内存块，随时都有被收回或作为他用的可能。

如果想获得正确的函数，改成下面这样就可以：

```
const char* strA()
{
    char *str = "hello word";
    return str;
}
```

首先要搞清楚 char *str 和 char str[]：

```
char c[] = "hello world";
```

是分配一个局部数组：

```
char *c = "hello world";
```

是分配一个指针变量：

局部数组是局部变量，它所对应的是内存中的栈。指针变量是全局变量，它所对应的是内存中的全局区域。字符串常量保存在只读的数据段，而不是像全局变量那样保存在普通数据段（静态存储区），如：

```
char *c = "hello world";
*c = 't'; //false
```

c 占用一个存储区域，但是局部区的数据是可以修改的：

```
char c[] = "hello world";
c[0] = 't'; //ok
```

这里 c 不占存储空间。

另外要想修改，也可以这样：

```
const char* strA()
{
    static char str[] = "hello word";
    return str;
}
```

通过 static 开辟一段静态存储空间。

答案：因为这个函数返回的是局部变量的地址，当调用这个函数后，这个局部变量 str 就释放了，所以返回的结果是不确定的且不安全，随时都有被收回的可能。

面试例题4：写出下面程序的运行结果。[美国著名硬盘公司S 2008年4月面试题]

```
int a[3];
a[0]=0; a[1]=1; a[2]=2;
int *p, *q;
```

```
p=a;
q=&a[2];
cout << a[q - p] << '\n';
```

解析：本题考的是指针与地址的关系问题。

本程序结构如下：

- (1) 先声明了一个整型数组 a[3]，然后分别给数组赋值。
- (2) 又声明了两个整数指针 p、q，但是并没有定义这两个指针所指向的地址。
- (3) 使整数指针 p 的地址指向 a (注意 a 就是 a[0])，使整数指针 q 的地址指向 a[2]。

可实际验证程序如下：

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

int main()
{
    int a[3];
    a[0]=0; a[1]=1; a[2]=2;
    int *p, *q;
    p=a;
```

```
cout << p << '\n';
cout << *p << '\n';
q=&a[2];
cout << q << '\n';
cout << *q << '\n';
cout << a[q - p] << '\n';
cout << a[*q - *p] << '\n';
```

上面的输出结果分别是：

```
0x22ff68
0
0x22ff70
```

```
2
2
2
```

q 的实际地址是 0x22ff70，p 的实际地址是 0x22ff68。 $0x22ff70 - 0x22ff68 = 0x08$ （十六进制减法），相差是 8。

$q-p$ 的实际运算是 $(q \text{ 的地址值}(0x22ff70)-p \text{ 的地址值}(0x22ff68))/\text{sizeof(int)}$ ，即 $8/\text{sizeof(int)}=2$ 。

答案：运行结果是 2。

面试例题5：请问下面代码的输出结果是多少？[中国某互联网公司2009年12月笔试题]

```
#include <stdio.h>
class A
{
public:
    A() {m_a = 1; m_b = 2;}
    ~A() {};
    void fun() {printf("%d%d", m_a, m_b);}
private:
    int m_a;
    int m_b;
};

class B
{
public:
```

```
B() {m_c = 3;}
~B();
void fun() {printf("%d", m_c);}
private:
    int m_c;
};
void main()
{
    A a;
    B *pb= (B*)(&a);
    pb->fun();
}
```

解析:首先可以肯定的是上面这段代码是非常糟糕的,无论是可读性还是安全性都很差。写这种代码的人,按照Bjarne Stroustrup(C++语言化制定者)的说法,应该“斩立决”。

这道题出的目的就是考察你对内存偏移的理解:

```
B*pb = (*)(&a);
```

这是一个野蛮的转化,强制把a地址内容看成是一个B类对象,pb指向的是a类的内存空间:

```
pb->fun();
```

正常情况下,B类只有一个元素是int m_c,但是a类的内存空间中存放第一个元素的位置是m_a,pb指向的是对象的内存首地址,比如0x22ff58,当pb->func()调用B::func()来打印m_c时,编译器对m_c对它的认识就是m_c距离对象的偏移量0,于是打印了对象a首地址的编译量0x22ff58+0变量值。所以打印的是m_a的值1。以下代码来证明如下:

```
#include <string.h>
#include <stdio.h>
#include <iostream>
using namespace std;
class A
{
public:
    A() {m_a = 1; m_b = 2;}
    ~A(){}
    void fun() {printf("%d%d", m_a, m_b);}
public:
    int m_a;
    int m_b;
};
class B
{
public:
    B() {m_c = 3;}
    ~B(){}
    void fun() {printf("%d", m_c);}
public:
    int m_c;
};
int main( void )
{
A a;
B*pb = (B*)(&a);
cout << &a << endl; // 0x22ff58
cout << &(a.m_a) << endl; // print the address of the a.m_a 0x22ff58
printf("%p\n", &A::m_a); // print the offset from m_a to the beginning A object address 00000000
printf("%p\n", &A::m_b); // print the offset from m_b to the beginning A object address 00000004
printf("%p\n", &B::m_c); // print the offset from m_c to the beginning B object address 00000000
pb->fun();
return 0;
}
```

答案: 1

面试例题 6：What results after run the following code? (下列代码的运行结果是什么?) [中国台湾某著名 CPU 生产公司 2005 年面试题]

```
int *ptr;
ptr=(int*)0x8000;
*ptr=oxaabb;
```

解析：指针问题。

答案：这样做会导致运行时错误，因为这种做法会给一个指针分配一个随意的地址，这是非常危险的。不管这个指针有没有被使用过，这么做都是不允许的。

面试例题 7：下列程序的输出结果是什么? [中国著名网络企业 XL 公司 2007 年 12 月面试题]

```
#include <iostream>
using namespace std;
class A
{
public:
    int _a;
    A()
    {
        _a = 1;
    }
    void print()
    {
        printf("%d", _a);
    }
};
```

```
class B : public A
{
public:
    int _a;
    B()
    {
        _a = 2;
    }
};
int main()
{
    B b;
    b.print();
    printf("%d ", b._a);
}
```

A. 22

B. 11

C. 12

D. 21

解析：B 类中的_a 把 A 类中_a 的“隐藏”了。在构造 B 类时，先调用 A 类的构造函数。所以 A 类的_a 是 1，而 B 类的_a 是 2。

答案：C

面试例题 8：以下描述正确的是()。[中国著名网络企业 XL 公司 2010 年 7 月面试题]

- A. 函数的形参在函数未调用时预分配存储空间
- B. 若函数的定义出现在主函数之前，则可以不必再说明
- C. 若一个函数没有 return 语句，则什么值都不返回
- D. 一般来说，函数的形参和实参的类型应该一致

解析：

- A: 错误的，调用到实参才会分配空间。
- B: 函数需要在它被调用之前被声明，这个跟 main() 函数无关。
- C: 错误的，在主函数 main 中可以不写 return 语句，因为编译器会隐式返回 0；但是在

一般函数中没 return 语句是不行的。

D: 正确的。

答案: D

面试例题 9: 下列程序会在哪一行崩溃? [美国著名软件企业 M 公司 2007 年 11 月面试题]

```
struct S {
    int i;
    int * p;
};

main()
{
    S s;
```

```
int*p=&s.i;
p[0]=4;
p[1]=3;
s.p=p;
s.p[1]=1;
s.p[0]=2;
```

解析:

int *p = &s.i; 相当于 int *p; p = &s.i。当执行 p[0]=4; p[1]=3; 的时候, p 始终等于 &s.i。s.p = p 相当于建立了如下关系:

s.p 存了 p 的值,也就是 &s.i; s.p[1] 相当于 *(&s.i + 1),即 s.i 的地址加 1,也就是 s.p。s.p[1] 跟 s.p 其实是同一个地方,所以到 s.p[1]=1,那么 s.p[0] 将指向内存地址为 1 的地方。

s.p[0] = 2; 并不是给 s.i 赋值,而是相当于 *((int *)1) = 2;。

也就是要访问 0x00000001 空间——对于一个未做声明的地址直接进行访问,所以访问出错。

编写程序如下:

```
#include <iostream>
using namespace std;
struct S
{
    int i;
    int *p;
};

main()
{
    S s;
    int *p=&s.i;
    p[0] =1;
    p[1] =5;
    //s.p=p;
    //s.p[1]=1;
    cout << p[0] << " " << s.i << endl;
    cout << &p[0] << " " << &s.i << endl;
    cout << p[1] << " " << s.p << " " <<
    endl;
```

```
cout << &p[1] << " " << &s.p << " "
<< &s.p[1] << endl;
cout << endl;
s.p = p;
cout << p[0] << " " << s.i << endl;
cout << &p[0] << " " << &s.i << endl;
cout << p[1] << " " << s.p << " " <<
s.p[1] << endl;
cout << &p[1] << " " << &s.p << " "
<< &s.p[1] << endl;
s.p[1] = 1;

cout << s.p << " " << &s.p << endl;
//s.p << " " << endl;

//s.p[0] = 2; //程序崩溃
//s.p = 2; // s.p[0]相当子*s.p
```

可以看到输出结果如下:

1 1

0x22ff78 0x22ff78

```
5 0x5
0x22ff7c 0x22ff7c 0x9
1 1
```

```
0x22ff78 0x22ff78
2293624 0x22ff78 2293624
0x22ff7c 0x22ff7c 0x22ff7c
0x1 0x22ff7c
```

答案：s.p[0] = 2; 行程序会崩溃。

7.3 函数指针

面试例题 1：const char *const * keyword1; const char const * keyword2; const char *const keyword3; const char const keyword4。请问以上四种定义，所得出的变量有什么区别，各代表什么？

答案：const char *const * keyword1; 是二级指针，const 在第二个*之前表示二级指针指向的内容不可修改，但是二级指针本身可以修改。如下例所示：

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

int main(void)
{
    const char *const * keywords;
    const char const * keywords2;
```

```
const char *p = "Hello world!\n";
keywords = &p;
cout << p << *keywords;
return 0;
}
```

相当于 const char* keywords2; 它是一个指向 const char 的指针。

const char *const keywords3;

它是一个指向 const char 的常指针，即指针本身的存储属性也是 const。

const char const keywords4;

它是一个字符常量：

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 10 月面试题]

```
//在这三个程序中找出最大值的程序
#include <stdio.h>

int max(int x,int y) {
    return x>y?x:y;
}

int main() {
    int max(x,y);
    int *p=&max;
    int a,b,c,d;
```

```
printf("Please input three integer
\n");
scanf("%d%d%d",a,b,c);
d=(*p)((*p)(a,b),c);
printf("Among %d, %d, and %d, the
maximal integer is
%d\n", a,b,c,d);
return 0;
}
```

解析：这道程序体存在着函数指针的错误使用问题。

答案：

正确的程序如下：

```
//最好使用 <cstdio>
#include <stdio.h>
int max(int x,int y) {
    return x>y?x:y;
}
int main() {
    int max(int,int);           //错误1
    int (*p)(int,int)=&max;     //错误2
    int a,b,c,d;
```

```
printf("Please input three integer
\n");
scanf("%d%d%d",&a,&b,&c);      //错误3
d=(*p)((*p)(a,b),c);
printf("Among %d, %d, and %d, the
maximal integer is
%d\n", a,b,c,d);
return 0;
}
```

面试例题 3：Write in words the data type of the identifier involved in the following definitions. (下面的数据声明都代表什么？) [美国某著名计算机嵌入式公司 2005 年 9 月面试题]

- (1) float(**def)[10];
- (2) double*(*gh)[10];
- (3) double(*f[10])();
- (4) int*((*b)[10]);
- (5) Long (* fun)(int)
- (6) Int (*(*F)(int,int))(int)

解析：函数指针的问题。

就像数组名是指向数组第一个元素的常指针一样，函数名也是指向函数的常指针。可以声明一个指向函数的指针变量，并且用这个指针来调用其他函数——只要这个函数和你的函数指针在签名、返回、参数值方面一致即可。

```
Long (* fun) (int) ,
```

上面就是一个函数指针——指向函数的指针，这个指针返回值是 long，所带的参数是 int。如果去掉(* fun)的“()”它就是指针函数，是一个带有整数参量并返回一个长整型变量的指针的函数。

```
Int (*(*F) (int,int)) (int) ,
```

如上所示，F 是一个指向函数的指针，它指向一种函数（该函数参数为 int，int 返回值为一个指针），返回的这个指针指向的是另外一个函数（参数类型为 int，返回值为 int 类型的函数）。

答案：

- (1) float(**def)[10];

def 是一个二级指针，它指向的是一个一维数组的指针，数组的元素都是 float。

(2) double*(*gh)[10];

gh 是一个指针，它指向一个一维数组，数组元素都是 double*。

(3) double(*f[10])();

f 是一个数组，f 有 10 个元素，元素都是函数的指针，指向的函数类型是没有参数且返回 double 的函数。

(4) int*((*b)[10]);

就跟 “int* (*b)[10]” 是一样的，是一维数组的指针。

(5) Long (* fun)(int)

函数指针。

(6) Int (*(*F)(int,int))(int)

F 是一个函数的指针，指向的函数的类型是有两个 int 参数并且返回一个函数指针的函数，返回的函数指针指向有一个 int 参数且返回 int 的函数。

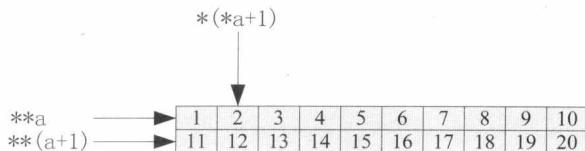
7.4 指针数组和数组指针

面试例题 1：以下程序的输出是（ ） [美国某软件公司 2009 年 12 月面试题目]

```
#include<stdio.h>
#include<iostream>
using namespace std;
int main()
{
    int v[2][10] = {{1,2,3,4,5,6,7,8,9,10},
{11,12,13,14,15,16,17,18,19,20}};
    int (*a)[10] = v; //数组指针
}
```

```
cout<< **a<<endl;
cout<< **(a+1)<<endl;
cout<< *(a+1)<<endl;
cout<< *(a[0]+1)<<endl;
cout<< *(a[1])<<endl;
return 0;
}
```

解析：本题定义一个指针指向一个 10 个 int 元素的数组。a+1 表明 a 指针向后移动 1*sizeof(数组大小)；a+1 后共向后移动 40 个字节。*a+1 仅针对这一行向后移动 4 个字节，如下图所示。



答案：输出如下：

1 11 2 2 11

面试例题 2：一个指向整型数组的指针的定义为（ ）。

- A. int (*ptr)[] B. int *ptr[] C. int *(ptr[]) D. int ptr[]

解析：

int (*ptr)[]是一个指向整型数组的指针。

int *ptr[]是指针数组，ptr[]里面存的是地址。它指向位置的值就是*ptr[0]、*ptr[1]、*ptr[2]、*ptr[3]。不要存*ptr[0]=5;、*ptr[1]=6;，因为这里面没有相应的地址。

int *(ptr[])与B相同。

int ptr[]是一个普通的数组。

答案：A

扩展知识

a是指针数组，是指一个数组里面装着指针。

b是指向数组的指针，代表它是指针，指向整个数组。

以下是指针数组a:

```
#include<iostream>
using namespace std;
int main()
{
    static int a[2]={1,2};

    int *ptr[5];
    //指针数组
    int p=5,p2=6,*page,*page2;
    page = &p;
```

下面是数组指针b:

```
#include<stdio.h>
#include<iostream>
using namespace std;
int main()
{
    static int a[2]={1,2};

    int p=5,p2=6,*page,*page2;
    //测试用二维数组
    int Test[2][3] = {{1,2,3}, {4,5,6}};
    //测试用二维数组
    int Test2[3] = {1,2,3};

    page = &p;
    page2 = &p2;
```

```
page2 = &p2;

ptr[0]=&p;
ptr[1]=page2;

cout << *ptr[0] << endl;
cout << *page << endl;
cout << *ptr[1] << endl;
return 0;
```

```
ptr[0]=&p;
ptr[1]=page2;
//int (*A)[3] = &Test[1];
//数组指针
int (*A)[3],(*B)[3];

A = &Test[1];
B = &Test2;
cout << *page << endl;
cout << (*A)[0] << (*A)[1] << (*A)[2]
<< endl;
cout << (*B)[3] << endl;
return 0;
```

面试例题3：用变量a给出下面的定义。[中国台湾某著名CPU生产公司2005年面试题]

- (1) 一个整型数 (An integer)
- (2) 一个指向整型数的指针 (A pointer to an integer)
- (3) 一个指向指针的指针，它指向的指针是指向一个整型数 (A pointer to a pointer to an integer)
- (4) 一个有10个整型数的数组 (An array of 10 integers)
- (5) 一个有10个指针的数组，该指针是指向一个整型数的 (An array of 10 pointers to integers)
- (6) 一个指向有10个整型数数组的指针 (A pointer to an array of 10 integers)
- (7) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数 (A pointer to a function that takes an integer as an argument and returns an integer)
- (8) 一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数 (An array of 10 pointers to functions that take an integer argument and return an integer)

解析：这道面试例题是嵌入式编程和指针运用中经常考到的问题。是那种要翻一下书才能回答的问题。当我写这本书时，为了确定语法的正确性，我的确查了一下书。但是当我被面试的时候，我期望被问到这个问题（或者相近的问题）。因为在被面试的这段时间里，我确定我知道这个问题的答案。应试者如果不知道所有的答案（或至少大部分答案），那么也就没有为这次面试做好准备。如果该面试者没有为这次面试做好准备，那么他又能为什么做好准备呢？

答案：

- (1) int a; // An integer
- (2) int *a; // A pointer to an integer
- (3) int **a; // A pointer to a pointer to an integer
- (4) int a[10]; // An array of 10 integers
- (5) int *a[10]; // An array of 10 pointers to integers
- (6) int (*a)[10]; // A pointer to an array of 10 integers
- (7) int (*a)(int); // A pointer to a function that takes an integer argument and returns an integer
- (8) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer

面试例题4：写出如下程序片段的输出。[美国某著名CPU生产公司面试题]

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

```
int *ptr = (int*)(&a + 1);
printf("%d %d", *(a + 1), *(ptr - 1));
```

解析：第一个结果好理解，是正常的指针运算。但是第二个却有点难以理解。

第二个的确是 5。首先 a 表示一个 1 行 5 列数组，在内存中表示为一个 5 个元素的序列。
`int *ptr = (int*)(&a + 1)` 的意思是，指向 a 数组的第 6 个元素（尽管这个元素不存在）。那么显然，`(ptr - 1)` 所指向的数据就是 a 数组的第 5 个元素——5。

如果存在这样的数组：

```
int b[2][5]={1,2,3,4,5,6,7,8,9,10}
```

那么显然：

```
int *p=(int *)(&a+1)=b[1]
```

实际上，b 的数据分布还是按照 1、2、3、4、5、6、7、8、9、10 分布的，所谓 `b[0]` 和 `b[1]` 实际上只是指向其中一个元素的指针。

时刻牢记这样的观点：数组名本身就是指针，再加个&，就变成了双指针，这里的双指针就是指二维数组，加 1，就是数组整体加一行，ptr 指向 a 的第 6 个元素。

答案：2, 5。

扩展知识（火烧赤壁的故事）

0X0000 (a) 里含有至少两个信息，第一就是地址本身，第二个是隐藏的所指向的数组的大小。1F000 地址直接和 0X0000 进行通信并为之提供服务，而不和 1、2、3、4、5 等直接通信，服务的内容为 0X0000 的需求，而和数组每个元素本身的大小没有直接关系（只是间接）。1F000 里也至少含有两个信息，即一是地址本身，二是所服务的对象的容量。就像一艘船按序排列有 1、2、3、4、5 个座位，`&a+1` 的意思是我要坐下一艘船的 1 号座位，而不是这艘船本身的座位。

话说曹操听了别人的计策，把 800 艘战船用铁链首尾相接（两船间稍有空隙）连成一条龙，准备攻打东吴。每艘船上顺序排列有 5 个位子，分别坐着船长、舵手、枪兵、弓兵、刀兵，每艘船及座位编号规律为 boat1~boat800_1~5，其中 boat1~800 代表本船在船队中的序号，1~5 代表本船上的位子。周瑜说：“把所有位子的人员按顺序逐个消灭。”诸葛亮说：“公瑾此言差矣，我用火攻，`&a+1` 的方法岂不是比逐个遍历 a[] 更快捷？即所谓倾巢之下，安有完卵？”周瑜听后道：“既生瑜，何生亮！”

7.5 迷途指针

面试例题 1：以下代码有什么错误？会造成什么问题？[美国某著名 CPU 生产公司面试题]

```
typedef unsigned short int USHORT;
#include <iostream.h>
int main()
{
    USHORT * pInt = new USHORT;
    *pInt = 10;
    cout << "*pInt: " << *pInt << endl;
    delete pInt;
    pInt = 0;
    long * pLong = new long;
```

```
*pLong = 90000;
cout << "*pLong: " << *pLong << endl;

*pInt = 20;      // uh oh, this was deleted!

cout << "*pInt: " << *pInt << endl;
cout << "*pLong: " << *pLong << endl;
delete pLong;
return 0;
}
```

解析：编程中一种很难发现的错误是迷途指针。迷途指针也叫悬浮指针、失控指针，是当对一个指针进行 `delete` 操作后——这样会释放它所指向的内存——并没有把它设置为空时产生的。而后，如果你没有重新赋值就试图再次使用该指针，引起的结果是不可预料的。如果程序崩溃都算走运了。

就如同一家水果公司搬家了，但你使用的仍然是它原来的电话号码。这可能不会导致什么严重的后果——也许这个电话号码是放在一个无人居住的房子里面。另一方面，这个号码也可能被重新分配给一个军工厂，你的电话可能引起爆炸，把整个城市炸毁。

总之，在删除指针后小心不要再使用它。虽然这个指针仍然指向原来的内存区域，但是编译器已经把这块内存区域分配给了其他的数据。再次使用这个指针会导致你的程序崩溃。更糟糕的情况是，程序可能表面上运行得很好，过不了几分钟就崩溃了。这被称为定时炸弹，可不是开玩笑。为了安全起见，在删除一个指针后，把它设置为空指针（0）。这样就可以消除它的危害。

在本题中，首先声明了一个指针 `pInt`，然后打印。打印后使用 `delete` 将其删除。那么现在 `pInt` 就是一个迷途指针，或者说是悬浮指针。

第 2 步，声明了一个新的指针 `pLong`，把 90 000 赋值给它，然后打印。

第 3 步，把值 20 赋给 `pInt` 所指向的内存区域，但此时 `pInt` 不指向任何有效的空间。因为它原来所指向的内存空间已经被释放了，所以这样做会给内存区域带来很坏的结果。

第 4 步，打印 `pInt` 的值，结果是 20。再打印 `pLong` 的值，发现它变成了 65556 而不是 90 000 了。这是因为把 90 000 赋给 `*pLong` 后，它实际存储为 5F 90 00 01。把 20（也就是十六进制的 00 14）赋给指针 `pInt`，因为指针 `pInt` 仍然指向相同的地址，因此 `pLong` 的前两个字节被覆盖了，变成了 00 14 00 01，所以打印的结果变成了 65 556。

答案：以上程序使用了迷途指针并重新赋值，会造成系统崩溃。

面试例题 2：空指针和迷途指针的区别是什么？

答案：当 delete 一个指针的时候，实际上仅是让编译器释放内存，但指针本身依然存在。这时它就是一个迷途指针。

当使用以下语句时，可以把迷途指针改为空指针：

```
MyPtr=0;
```

通常，如果在删除一个指针后又把它删除了一次，程序就会变得非常不稳定，任何情况都有可能发生。但是如果你只是删除了一个空指针，则什么事都不会发生，这样做非常安全。

使用迷途指针或空指针（如 MyPtr=0）是非法的，而且有可能造成程序崩溃。如果指针是空指针，尽管同样是崩溃，但它同迷途指针造成的崩溃相比是一种可预料的崩溃。这样调试起来会方便得多。

面试例题 3：C++中有了 malloc/free，为什么还需要 new/delete？

答案：malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，只用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。

因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。new/delete 不是库函数，而是运算符。

面试例题 4：下列程序的输出结果是什么？[中国著名网络企业 XL 公司 2007 年 10 月面试题]

```
#include<iostream>
using namespace std;
main()
{
    char* a[]{"hello","the","world"};
    char**pa=a;
    pa++;
    cout<<*pa<<endl;
}
```

- A. theworld B. the C. ello D. ellotheworld

解析：指针的指针问题。

答案：B。

7.6 指针和句柄

面试例题 1：句柄和指针的区别和联系是什么？[英国某著名计算机图形图像公司面试题]

解析：句柄是一个 32 位的整数，实际上是 Windows 在内存中维护的一个对象（窗口等）内存物理地址列表的整数索引。因为 Windows 的内存管理经常会将当前空闲对象的内存释放掉，当需要时访问再重新提交到物理内存，所以对象的物理地址是变化的，不允许程序直接通过物理地址来访问对象。程序将想访问的对象的句柄传递给系统，系统根据句柄检索自己维护的对象列表就能知道程序想访问的对象及其物理地址了。

句柄是一种指向指针的指针。我们知道，所谓指针是一种内存地址。应用程序启动后，组成这个程序的各对象是驻留在内存的。如果简单地理解，似乎我们只要获知这个内存的首地址，那么就可以随时用这个地址访问对象。但是，如果真的这样认为，那么就大错特错了。我们知道，Windows 是一个以虚拟内存为基础的操作系统。在这种系统环境下，Windows 内存管理器经常在内存中来回移动对象，以此来满足各种应用程序的内存需要。对象被移动意味着它的地址变化了。如果地址总是如此变化，我们该到哪里去找该对象呢？为了解决这个问题，Windows 操作系统为各应用程序腾出一些内存地址，用来专门登记各应用对象在内存中的地址变化，而这个地址（存储单元的位置）本身是不变的。Windows 内存管理器移动对象在内存中的位置后，把对象新的地址告知这个句柄地址来保存。这样我们只需记住这个句柄地址就可以间接地知道对象具体在内存中的哪个位置。这个地址是在对象装载（Load）时由系统分配的，当系统卸载时（Unload）又释放给系统。句柄地址（稳定）→记载着对象在内存中的地址→对象在内存中的地址（不稳定）→实际对象。但是，必须注意的是，程序每次重新启动，系统不能保证分配给这个程序的句柄还是原来的那个句柄，而且绝大多数情况下的确不一样。假如我们把进入电影院看电影看成是一个应用程序的启动运行，那么系统给应用程序分配的句柄总是不一样，这和每次电影院售给我们的门票总是不同的座位是一样的道理。

HDC 是设备描述表句柄。CDC 是设备描述表类。用 GetSafeHwnd 和 FromHandle 可以互相转换。

答案：句柄和指针其实是两个截然不同的概念。Windows 系统用句柄标记系统资源，隐藏系统的信息。你只要知道有这个东西，然后去调用就行了，它是个 32bit 的 uint。指针则标记某个物理内存地址，两者是不同的概念。

面试例题 2：In C++, which of the following are valid uses of the std::auto_ptr template considering the class definition below? （下面关于智能指针 auto_ptr 用法正确的是哪项？）[美国软件公司]

M2009年12月笔试试题]

```
class Object
{
public:
```

选项如下所示：

- A. std::auto_ptr <Object> pObj (new Object);
- B. std::vector <std::auto_ptr <Object*>> object_vector;
- C. std::auto_ptr <Object*> pObj (new

```
virtual ~Object() {}
//...
};
```

- Object);
- D. std::vector <std::auto_ptr <Object>> object_vector;
- E. std::auto_ptr <Object> source()
 { return new Object; }

解析：auto_ptr是安全指针。最初动机是使得下面的代码更安全：

```
void f() {
    T* pt( new T );
    /*...more code...*/
```

```
        delete pt;
    }
```

如果f()从没有执行delete语句（因为过早的return或者是在函数体内部抛出了异常），动态分配的对象将没有被delete，一个典型的内存泄漏。使其安全的一个简单方法是用一个“灵巧”的类指针对象包容这个指针，在其析构时自动删除此指针：

```
void f() {
```

```
    auto_ptr<T> pt( new T );
    /*...more code...*/
```

```
    }
```

```
}
```

现在，这个代码将不再泄漏T对象，无论是函数正常结束还是因为异常，因为pt的析构函数总在退栈过程中被调用。类似地，auto_ptr可以被用来安全地包容指针：

```
class C {
public:
    C();
    /*...*/
private:
```

```
    auto_ptr<CImpl> pimpl_;
};

// file c.cpp
C::C() : pimpl_( new CImpl ) { }
```

现在，析构函数不再需要删除pimpl_指针，因为auto_ptr将自动处理它。

如果了解auto_ptr格式，就知道A是对的，C是错的。B、D也是错的，因为auto_ptr放在vector之中是不合理的。因为auto_ptr的复制并不等价。当auto_ptr被复制时，原来的那一份会被删除。在《Exceptional C++》特别提到：“尽管编译器不会对此给出任何警告，把auto_ptr放入container仍是不安全的。这是因为我们无法告知这个container关于auto_ptr具有特殊的语义的情况。不错，如今我所知的大多数auto_ptr实现都会让你侥幸摆脱这个麻烦；而且在某些流行的编译器所提供的文档中，与此几乎相同的代码甚至还被作为优良的代码而给出。然而实际上它是不安全的（现在成为非法的了）。”auto_ptr并不满足对能够放入container的类别之需求，因为auto_ptr之间的复制是不等价的。创建额外的复制实在是不必要和低效的；出于商业竞争的考虑，一个厂商当然不可能会发售一个本来可以很高效的低效程序库。

E 也是正确的，从 new Object 构造出一个 auto_ptr <Object>。

答案：A, E。

7.7 this 指针

面试例题 1：Please choose the right statement of "this" pointer: (下面关于 this 指针哪个描述是正确的)

- A. "this" pointer cannot be used in static functions
- B. "this" pointer could not be stored in Register.
- C. "this" pointer is constructed before member function.
- D. "this" pointer is not counted for calculating the size of the object.
- E. "this" pointer is read only.? [英国某著名计算机图形图像公司面试题]

解析：解析:关于 This 指针，有这样一段描述：当你进入一个房子后，你可以看见桌子、椅子、地板等，但是房子你是看不到全貌了。

对于一个类的实例来说，你可以看到它的成员函数、成员变量，但是实例本身呢？ this 指针是这样一个指针，它时时刻刻指向这个实例本身。

this 指针易混的几个问题如下。

(1) This 指针本质是一个函数参数，只是编译器隐藏起形式的，语法层面上的参数。

this 只能在成员函数中使用，全局函数、静态函数都不能使用 this。

实际上，成员函数默认第一个参数为 T* const this。

如：

```
eclass A
{
    public:
        int func(int p) {}
};
```

其中，func 的原型在编译器看来应该是：

```
int func(A* const this, int p);
```

(2) this 在成员函数的开始前构造，在成员的结束后清除。这个生命周期同任何一个函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的 this 参数传递进去。如：

```
A a;
a.func(10);
```

此处，编译器将会编译成：

```
A::func(&a, 10);
```

看起来和静态函数没差别，不过，区别还是有的。编译器通常会对 this 指针做一些优化，因此，this 指针的传递效率比较高，如 VC 通常是通过 ecx 寄存器传递 this 参数的。

(3) this 指针并不占用对象的空间。

this 相当于非静态成员函数的一个隐含的参数，不占对象的空间。它跟对象之间没有包含关系，只是当前调用函数的对象被它指向而已。

所有成员函数的参数，不管是隐含的，都不会占用对象的空间，只会占用参数传递时的栈空间，或者直接占用一个寄存器。

(4) this 指针是什么时候创建的？

this 在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果 class 或者 struct 里面没有方法的话，它们是没有构造函数的，只能当作 C 的 struct 使用。采用 TYPE xx 的方式定义的话，在栈里分配内存，这时候 this 指针的值就是这块内存的地址。采用 new 方式创建对象的话，在堆里分配内存，new 操作符通过 eax 返回分配的地址，然后设置给指针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给 ecx。

(5) this 指针存放在何处？堆、栈、还是其他？

this 指针会因编译器不同而有不同的放置位置。可能是堆、栈，也可能是寄存器。

C++ 是一种静态的语言，那么对 C++ 的分析应该从语法层面和实现层面两个方面进行。

语法上，this 是个指向对象的“常指针”，因此无法改变。它是一个指向相应对象的指针。所有对象共用的成员函数利用这个指针区别不同变量，也就是说，this 是“不同对象共享相同成员函数”的保证。

而在实际应用的时候，this 应该是个寄存器参数。这个不是语言规定的，而是“调用约定”，C++ 的默认调用约定是_cdecl，也就是 C 风格的调用约定。该约定规定参数自右向左入栈，由调用方负责平衡堆栈。对于成员函数，将对象的指针（即 this 指针）存入 ecx 中（有的书将这一点单独分开，叫作 thiscall，但是这的确是 cdecl 的一部分）。因为这只是一个调用约定，不是语言的组成部分，不同编译器自然可以自由发挥。但是现在的主流编译器都是这么做的。

(6) this 指针是如何传递给类中的函数的？绑定？还是在函数参数的首参数就是 this 指针？那么，this 指针又是如何找到“类实例后函数”的？

大多数编译器通过 ecx 寄存器传递 this 指针。事实上，这也是一个潜规则。一般来说，

不同编译器都会遵从一致的传参规则，否则不同编译器产生的 obj 就无法匹配了。

(7) 我们只有获得一个对象后，才能通过对象使用 this 指针。如果我们知道一个对象 this 指针的位置，可以直接使用吗？

this 指针只有在成员函数中才有定义。因此，你获得一个对象后，也不能通过对象使用 this 指针。所以，我们无法知道一个对象的 this 指针的位置（只有在成员函数里才有 this 指针的位置）。当然，在成员函数里，你是可以知道 this 指针的位置的（可以通过`&this`获得），也可以直接使用它。

答案：E。

第 8 章

循环、递归与概率

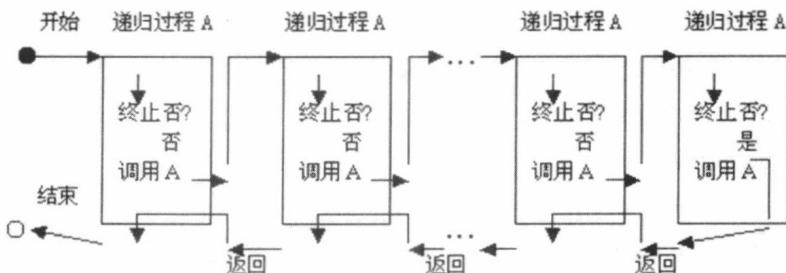
递归问题是求职笔试中最为复杂的地方，也是本书的难点之一。由递归衍生出的相关问题，诸如迭代问题、概率问题、循环问题也是企业经常重复的考点。在阅读本章之前，请读者参考数据结构经典书籍对递归基础知识做简要复习。本章将通过对各公司面试题目进行全面仔细的解析，帮助读者解决其中的难点。

8.1 递归基础知识

递归是程序设计中的一种算法。一个过程或函数直接调用自己本身或通过其他的过程或函数调用语句间接地调用自己的过程或函数，称为递归过程或函数。递归是计算机语言中的一种很有用的工具，很多数学公式用到递归定义，例如 $N!$ ：当 $n > 0$ 时， $f(n) = n \times f(n - 1)$ 。

有些数据结构（如二叉树），其结构本身就有递归的性质。有些问题本身没有明显的递归结构，但用递归求解更简单。

递归是较难理解的算法之一。简单地说，递归就是编写这样一个特殊的过程，该过程中有一个语句用于调用过程自身（称为递归调用）。递归过程由于实现了自我的嵌套执行，使这种过程的执行变得复杂起来，其执行的流程如下图所示。



递归过程的执行总是一个过程体未执行完，就带着本次执行的结果又进入另一轮过程体的执行，……，如此反复，不断深入，直到某次过程的执行时终止递归调用的条件成立，则不再深入，而执行本次的过程体余下的部分，然后又返回到上一次调用的过程体中，执行余下的部分，……，如此反复，直到回到起始位置上，才最终结束整个递归过程的执行，得到相应的执行结果。递归过程程序设计的核心就是参照这种执行流程，设计出一种适合“逐步深入，而后又逐步返回”的递归调用模型，以解决实际问题。

递归算法应该包括递归情况和基底情况两种情况。递归情况演变到最后必须达到一个基底。

在程序设计面试中，一个能够完成任务的解决方案是最重要的，解决方案的执行效率要放在第二位考虑。因此，除非试题另有要求，应该从最先想到的解决方案入手。如果它是一个递归性的方案，不妨向面试官说明一下递归算法天生的低效率问题——表示你知道这些事情。有时候同时想到两个解决方案：递归的和循环的，并且实现方式差不多，可以把两个都向考官介绍一下。比如 $N!$ 问题，利用循环语句解释就是：

```
int find(int i)
{
    int n, val=1;
    for(n=i; n>1; n--)
        val*=n;
    return val;
}
```

不过，如果用循环语句做的改进算法实现起来要比递归复杂得多的话——大幅度增加了复杂度而在执行效率上得不到满意的回报，我们建议还是优先选择递归来解决问题。

面试例题 1：递归函数 mystrlen(char *buf, int N)是用来实现统计字符串中第一个空字符前面字符长度。举例来说：

```
char buf[] = {'a', 'b', ' ', 'd', 'e', 'f', '\0', 'x', 'y', 'z'};
```

字符串 buf，当输入 $N=10$ 或者 20 ，期待输出结果是 6 ；当输入 $N=3$ 或 5 ，期待输出结果是 3 或 5 。

```
int mystrlen(char *buf, int N)
{
    return mystrlen(buf, N/2) + mystrlen(buf + N/2, N/2);
}
```

What are all the possible mistakes in the code? （代码中可能的错误是哪个/些）

- A. There are no mistakes in the code （没错）
- B. There is no termination of recursion （没有递归退出条件）
- C. Recursion cannot be used to calculate this function （递归是不能实现这个函数的功能）
- D. The use of $N/2$ in the recursion is incorrect （对 $N/2$ 的使用是错误的）

解析：递归函数关注以下几个因素：退出条件，参数有哪些？返回值是什么？局部变量有哪些？全局变量有哪些？何时输出？会不会导致堆栈溢出？本题中递归函数显然没有退出条件。

答案：B

扩展知识：下面是该递归函数的正确实现方法

```
#include<iostream>
using namespace std;
int mystrlen(char *buf , int N)
{
    if(buf[0]==0||N==0) //如果空字符出现,返回0
        return 0;
    else if (N==1)      //如果字符长度为1,返回1
        return 1;
    int t = mystrlen(buf, N/2); //折半递归取长度
    if(t<N/2) //如果长度小于输入N值的一半,取当前长度
        return t;
    else      //反之取下一个字符并继续递归
        return (t + mystrlen(buf + N/2, (N+1)/2));
}

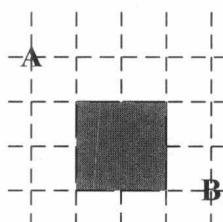
int main()
{
    char buf[] = {'a','b', 'c', 'd','e','f','\0','x','y','z'};
    int k;
    k = mystrlen (buf, 20);
    cout << k << endl;
    system("pause");
    return 0;
}
```

面试例题 2：Find the number of different shortest paths from point A to point B in a city with perfectly horizontal streets and vertical avenues as shown in the following figure. No path can cross the fenced off area shown in gray in the figure.

- A. 11 B. 15 C. 17 D. 19 E. 20

解析：此题中想要最短距离到达 B 点，所有行走路径，从起点出发后只能→或↓，任何←和↑都将增加路径长度。

在不存在阻碍的情况下，假设在任意 M,N 的此种格子上，从左上 A 出发到右下 B 的不同走法有 $f(M,N)$ 种，则根据递推可知 $f(M,N) = f(M-1,N) + f(M,N-1)$ ，等号右边两项分别对



应在当前点向下走一步和向右走一步的情况。递归终止情况为：

$$f(M,0) == f(0,N) == 0 \text{ 和 } f(1,1) == 1.$$

对于题目中有阻碍的情况，如下：

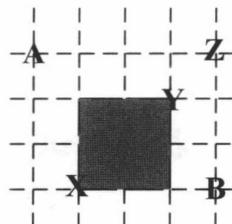
从 A 出发到 X 点的不同走法数为 $f(4,2)$ ，到达 X 点后只有 1 种走法，即一直向右到 B。

从 A 出发到 Y 点的不同走法数为 $f(2,4)$ ，到达后再到 B 点的不同走法数为 $f(3,2)$ ，因此连接的总数为 $f(2,4)*f(3,2)$ 。

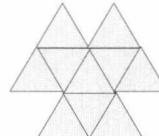
至此，唯一没有涵盖的走法为经 Z 点至 B 点，此种走法有且只有 1 种。

$$\text{因此总数目为 } f(4,2) + f(2,4)*f(3,2) + 1 == 17.$$

答案：C



面试例题 3：An algorithm starts with a single equilateral triangle and on each subsequent iteration add new triangles all around the outside. The result for the first three values of n are shown following figure. How many small triangles will be there after the 100 iterations?



n=3

- A. 19800 B. 14501 C. 14851 D. 14702 E. 15000

解析：本题规律如下，新增加的小三角形数目为 $3 * (n - 1)$ 。

$$f(1)=1;$$

$$f(2)=f(1)+3*(2-1);$$

$$f(3)=f(2)+3*(3-1);$$

.....

$$f(n)=f(n-1)+3*(n-1)$$

答案：C

8.2 典型递归问题

面试例题 1：If we define $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ ($n \geq 2$), what is the result of

$F(1025) \bmod 5$?[美国著名操作系统软件企业 M 公司 2013 年面试题]

- A. 0 B. 1 C. 2 D. 3 E. 4

解析：这里是对递归函数取余，所以我们最好先做一下拆项：

$$\begin{aligned} F(5n) &= F(5n-1) + F(5n-2) \\ &= 2 * F(5n-2) + F(5n-3) \\ &= 3 * F(5n-3) + 2 * F(5n-4) \\ &= 5 * F(5n-4) + 3 * F(5n-5) \end{aligned}$$

所以 $F(1025) \bmod 5 = 3 * F(1020) \bmod 5$ ；

依此类推：

$$F(1020) \bmod 5 = 3 * F(1015) \bmod 5;$$

.....

$$F(10) \bmod 5 = 3 * F(5) \bmod 5;$$

$F(5)$ 为 5, \bmod 后值为 0。所以 $F(1025) \bmod 5 = 0$ ；

答案：A

面试例题 2：请给出此题的非递归算法：

$$f(m, n) = \begin{cases} n & (m = 1) \\ m & (n = 1) \\ f(m-1, n) + f(m, n-1) & (m > 1, n > 1) \end{cases}$$

[中国著名门户网站企业 S 公司 2008 年 6 月面试题]

解析：本题类似于杨辉三角形，其实就是计算一个对角线值，用 list 保存一个对角线元素即可。除了横边和纵边按顺序递增外，其余每一个数是它左边和上边数字之和。

```
1, 2, 3, 4, 5
2, 4, 7, 11, 16
3, 7, 14, 25, 41
4, 11, 25, 50, 91
5, 16, 41, 91, 182
```

如上所示，假如想求 m 为 3、 n 为 4 的 $f(m, n)$ 值，就是 25。

答案：代码如下：

```
#include <iostream>
using namespace std;
#define RECURSION 0
#define NO_RECURSION 1

//=====递归版本=====
#if RECURSION
int f(int m, int n)
{
    if (m == 1)
        return n;
    if (n == 1)
        return m;
    return f(m-1, n) + f(m, n-1);
}
#endif
```

```
if (l == m)
{
    return n;
}
if (l == n)
{
    return m;
}
return f(m-1, n) + f(m, n-1);
```

```
#endif

#if NO_RECURSION
//=====非递归版本=====
int f(int m, int n)
{
    int a[100][100];
    for (int i=0;i!=m;++i)
        a[i][0]=i+1;
    for (int i=0;i!=n;++i)
        a[0][i]=i+1;
    for (int i=1;i!=m;++i)
```

```
for (int j=1;j!=n;++j)
    a[i][j]=a[i-1][j]+a[i][j-1];
return a[m-1][n-1];
}
#endif

int main()
{
    cout << f(5, 5) << endl;
    return 0;
}
```

面试例题 3：设计递归算法 $x(x(8))$ 需要调用几次函数 $x(\text{int } n)$ 。[美国著名数据分析软件企业 SA 公司 2009 年 11 月面试题]

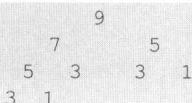
```
class Program
{
    static void Main(string[] args)
    {
        int i;
        i = x(x(8));
    }
}
```

```
static int x(int n)
{
    if (n <= 3)
        return 1;
    else
        return x(n - 2) + x(n - 4) + 1;
}
```

解析：单计算 $x(x(8))$ 的值自然是 9，但是本题要考查的是调用了几次 x 函数。可以把 $x(8)$ 理解为一个二叉树。树的节点个数就是调用次数：



$x(8)$ 的结果为 9； $x(x(8))$ 也就是 $x(9)$ 的二叉树形如：



节点数为 9，也就是又调用了 9 次函数。所以一共调用的次数是 18 次。

答案：18

8.3 循环与数组问题

面试例题 1：以下代码的输出结果是什么？[中国著名金融企业 J 银行 2008 年面试题]

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

```
using namespace std;
int main()
```

```
{
    int x = 10; int y=10,i;
    for(i=0;x>8;y=i++)
}
```

```
{
    printf("%d,%d,",x--,y);
    return 0;
}
```

- A. 10,0,9,1 B. 10,10,9,0 C. 10,1,9,2 D. 9,10,8,0

解析：for 循环括号内被两个分号分为 3 部分：i=0 是初始化变量；x>8 是循环条件，也就是只要 x>8 就执行循环；那 y=i++ 是什么？在第一次循环时执行了么？答案是不执行，y=i++ 实际上是个递增条件，仅在第二次循环开始时才执行。所以结果是 10,10,9,0。

面试者务必要搞清楚下面程序和题目的不同点：

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

int main()
{
```

```
int x = 10; int y=10,i;
for(i=0;x>8;)

    {y=i++;
    printf("%d,%d,",x--,y);
    return 0;
}
```

与题目不同，y=i++ 在循环体内，而不作为递增条件，所以在第一次循环就执行了，所以输出结果是 10,0,9,1。

答案：B

面试例题 2：输入 n，求一个 $n \times n$ 矩阵，规定矩阵沿 45 度线递增，形成一个 zigzag 数组 (JPEG 编码里取像素数据的排列顺序)，请问如何用 C++ 实现？[中国台湾著名硬件公司 2007 年 11 月面试题]

解析：在 JPEG 图形算法中首先对图像进行分块处理，一般分成互不重叠且大小一致的块，量化的结果保留了低频部分的系数，去掉了高频部分的系数。量化后的系数按 zigzag 扫描重新组织，然后进行哈夫曼编码。zigzag 数组是一个“之”字形排列的数组。

答案：用 C++ 编写完整代码如下：

```
/**
 * 得到如下样式的二维数组
 * zigzag (JPEG 编码里取像素数据的排列顺序)
 *
 * 0, 1, 5, 6, 14, 15, 27, 28,
 * 2, 4, 7, 13, 16, 26, 29, 42,
 * 3, 8, 12, 17, 25, 30, 41, 43,
 * 9, 11, 18, 24, 31, 40, 44, 53,
 * 10, 19, 23, 32, 39, 45, 52, 54,
 * 20, 22, 33, 38, 46, 51, 55, 60,
 * 21, 34, 37, 47, 50, 56, 59, 61,
 * 35, 36, 48, 49, 57, 58, 62, 63
 */
```

```
#include <stdio.h>
#include <iostream>

int main()
{
    int N;
    int s, i, j;
    int squa;
    scanf("%d", &N);
    /* 为指向 int 型指针的指针分配空间，该指针指向 n 个型指针。 */
    int **a = (int **)malloc(N *
                           sizeof(int));
    if(a == NULL)
```

```

        return 0;
    for(i = 0; i < N; i++)
    {
        if((a[i] = (int *)malloc(N *
            sizeof(int))) == NULL)
            /*对于前面的指针的每个值 (int 指针) 赋
            值, 使其指向一个 int 数组, 如果分配失败, 则释放掉它之前
            申请成功的空间*/
        {
            while(--i>=0)
                free(a[i]);
            free(a);
            return 0;
        }
    }
    /* 数组赋值 */
    squa = N*N;
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++) {

```

```

        s = i + j;
        if(s < N)
            a[i][j] = s*(s+1)/2 + (((i+j)%
                2 == 0)? i : j);
        else {
            s = (N-1-i) + (N-1-j);
            a[i][j] = squa - s*(s+1)/2 -
                (N-(((i+j)%2 == 0)? i : j));
        }
    }
    /* 打印输出 */
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++)
            printf("%6d", a[i][j]);
        printf("\n");
    }
    return 0;
}

```

面试例题 3：有两等长数组 A、B，所含元素相同，但顺序不同，只能取得 A 数组某值和 B 数组某值进行比较，比较结果为大于、小于或等于，但是不能取得同一数组 A 或 B 中的两个数进行比较，也不能取得某数组中的某个值。写一个算法实现正确匹配（即 A 数组中某值与 B 中某值等值）。[英国著名图形图像公司 A 2007 年 4 月校园招聘面试题]

解析：算法：循环加判断可以很快地解决这个问题。算法分析：假设两个数组 A[10]、B[10]。将 A.0 与 B.0 进行比较，判断它们是否等值或大于、小于，如果等值则打印出来，不等则比较 B.1……依此类推。

答案：

代码如下：

```

#include <iostream>

using namespace std;

void matching(int a[], int b[], int k)
{
    int i=0;
    while(i<=k-1)
    {
        int j=0;
        while(j<=k-1)
        {
            if(a[i]==b[j])
            {
                cout << "a[" << i << "] " << "match"
                << "b[" << j << "] " << endl;
                break;
            }
        }
    }
}

```

```

        }
        j++;
    }
    i++;
}
cout<<endl;
}

int main()
{
int a[10]={1,2,3,4,5,6,7,8,9,10};
int b[10]={10,6,4,5,1,8,7,9,3,2};

int k=sizeof(a)/sizeof(int);
matching(a,b,k);
return 0;
}

```

扩展知识

我们这里用的循环加比较的算法可以很快地解决这个问题，但却并不是最优算法。如果笔试时间充足的话，我们可以对算法做某些优化。

建立一个结构数组 C ，结构为 {某数在 B 中的位置, 标记, 某数在 A 中的位置}。其中“标记”可为大于、小于、等于。“某数在 A/B 中的位置”为 $0 \sim n-1$ ，这是相应位置。

第一次比较后， C 中元素都为 {某数在 B 中的位置, 标记, $A[0]$ } 格式。然后执行如下步骤：

(1) 在 A 数组中随机选取一个数（根据题意，我们并不知道这个值的确定值是多少），比如说 $A[i]$ ，然后和 B 数组中的数进行比较。根据数据结构 C ，将 B 数组中每个数与 $A[i]$ 进行比较，若比 $A[i]$ 大，从后向前存储，比 $A[i]$ 小则从前向后存储，要是等于 $A[i]$ ，就记录下这个值在 B 的位置 j 。继续比较，直到 B 数组中的数全部比较完成，然后再把这个 $b[j]$ 插入空余的那个中间位置。

(2) 然后再从 A 数组中取出数 $A[k]\{k=0 \sim n\}$ 与 $B[j]$ （这个 $B[j]$ 就是 $A[i]$ ，因为同一数组中不能比较大小，只能采用这种方式）比较，若比 $B[j]$ 大，那么从结构数组 C 中 $A[i]$ 后面比较，若比 $B[j]$ 小，就从结构数组 C 中 $A[i]$ 前面比较，直到找到相等的为止，然后更新结构数组 C 中与这个相等的相应值。注意，在这里，只更新相等的那个数值的“标记”，其他与 $A[k]$ 不相同（或大，或小）的情况下不更新，即还保持 $A[i]$ 的比较结果，以利于下一次进行比较。

(3) 重复步骤 (2)，继续取 A 数组剩下的值，仍然与那个 $B[j]$ 比较，这样逐步更新结构数组 C ，直到 A 数组全部取出比较完，那么这个程序也就完成了相应的功能。

这里用到了快速排序和二分法的某些思想。选择合理的 $A[i]$ ，可以大大降低比较次数。

面试例题 4：The following C++ code tries to count occurrence of each ASCII character in given string and finally print out the occurrence numbers. (下面 C++ 代码用来统计每个 ASCII 字符的出现次数，最后给出出现数值。)

```
#include<iostream>
#include<cstdlib>
void histogram(char* src)
{
    int i;
    char hist[256];
    for (i=0;i <256;i++)
    {
        hist[i]=0;
    }
    while(*src != '\0')
    {
        hist[*src]++;
    }
    for(i=0;i <=256;i++)
    {
        printf("%d ", hist[i]);
    }
}
```

```

    }
    int main()
    {
        Char*src=
"aaaabccdefghijklmnopqrst1234567890";
    }

```

```

histogram(src);
return 0;
}

```

If there may be some issue in the code above, which line(s) would be? (如果上面代码有错，将在哪行出现，如何修改？)

答案：这段代码有两个错：

(1) hist[*src]++;这一行应该修改为 hist[*src++]++;
这是因为 in while(*src != '\0') 循环体是看 *src 的值是否为 '\0' 来作为结束的。所以 src 必须递加。否则 hist[*src] 其实就是 hist['a'],'a' 会隐式转换成 97 也就是 hist[97]++ 不停递加，进入死循环。

(2) for(i=0;i <=256;i++)这一行应该修改为 for(i=0;i <=255;i++)，否则会超界。

面试例题 5：设计一个定时器程序，当输入一个时间，就按照输入值停留多少。[美国著名软件公司 I2013 年笔试试题]

解析：可以用 Sleep 来控制时间，并用双精度传值作参数传递。

答案：

```

#include <windows.h>
#include <iostream>
using namespace std;
void timer(double x)
{
    double cnt=0;
    cnt = x * 1000;
    Sleep(cnt); //停留 x*1000 微秒

}
int main()
{
    timer(0.5);
    cout << "0.5 秒过去了" << endl;
    timer(2*60);
    cout << "2 分钟过去了" << endl;
    timer(1*60*60);
    cout << "1 个小时过去了" << endl;
    return 0;
}

```

8.4 螺旋队列问题

面试例题 1：看清以下数字排列的规律，设 1 点的坐标是(0,0)，x 方向向右为正，y 方向向下为正。例如，7 点的坐标为(-1,-1)，2 点的坐标为(0,1)，3 点的坐标为(1,1)。编程实现输入

任意一点坐标(x, y)，输出所对应的数字。[芬兰某著名软件公司 2005 年面试题]

```
21 22 .....
20 7 8 9 10
19 6 1 2 11
18 5 4 3 12
17 16 15 14 13
```

解析：规律能看出来，问题就在于如何利用它。很明显这个队列是按顺时针方向螺旋向外扩展的，我们可以把它看成一层一层往外延伸。第 0 层规定为中间的那个 1，第 1 层为 2 到 9，第 2 层为 10 到 25。1、9、25……不就是平方数吗？而且是连续奇数（1、3、5……）的平方数。这些数还跟层数相关。推算一下就可以知道，第 t 层之内共有 $(2t-1)^2$ 个数，因而第 t 层会从 $[(2t-1)^2] + 1$ 开始继续往外螺旋伸展。给定坐标(x, y)，如何知道该点处于第几层？层数 $t = \max(|x|, |y|)$ 。

知道了层数，接下来就好办多了，这时我们就知道所求的那点一定在第 t 层这个圈上，顺着往下数就是了。要注意的就是螺旋队列数值增长方向和坐标轴正方向并不一定相同。可以分成 4 种情况——上、下、左、右或者东、南、西、北，分别处于 4 条边上进行分析。

- 东|右： $x == t$ ，队列增长方向和 Y 轴一致，正东方向 ($y = 0$) 数值为 $(2t-1)^2+t$ ，所以 $v = (2t-1)^2+t+y$ 。
- 南|下： $y == t$ ，队列增长方向和 X 轴相反，正南方向 ($x = 0$) 数值为 $(2t-1)^2+3t$ ，所以 $v = (2t-1)^2+3t-x$ 。
- 西|左： $x == -t$ ，队列增长方向和 Y 轴相反，正西方向 ($y = 0$) 数值为 $(2t-1)^2+5t$ ，所以 $v = (2t-1)^2+5t-y$ 。
- 北|上： $y == -t$ ，队列增长方向和 X 轴一致，正北方向 ($x = 0$) 数值为 $(2t-1)^2+7t$ ，所以 $v = (2t-1)^2+7t+x$ 。

其实还有一点很重要，不然会有问题。其他 3 条边都还好，但是在东边（右边）那条线上，队列增加不完全符合公式。注意到东北角（右上角）是本层的最后一个数，再往下却是本层的第一个数，当然不满足东线公式。所以我们把东线的判断放在最后（其实只需要放在北线之后就可以了），这样一来，东北角那点始终会被认为是北线上的点。

答案：代码如下：

```
#include <stdio.h>
#define max(a,b) ((a)<(b)?(b):(a))
#define abs(a) ((a)>0?(a):-(a))
int foo(int x, int y)
{
    int t = max(abs(x), abs(y));
    int u = t + t;
    int v = u - 1;
    v = v * v + u;
```

```
if (x == -t)
    v += u + t - y;
else if (y == -t)
    v += 3 * u + x - t;
else if (y == t)
    v += t - x;
else
    v += y - t;
return v;
```

```

}
int main()
{
    int x, y;
    for (y=-4;y<=4;y++)
    {
        for (x=-4;x<=4;x++)
    }
}

```

```

printf("%5d", foo(x, y));
printf("\n");
}
while (scanf("%d%d", &x, &y)==2)
    printf("%d\n", foo(x, y));

return 0;
}

```

扩展知识（有时公司还会考类似的面试题如下）

如矩阵：

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

找出规律，并打印出一个 $N \times N$ 的矩阵；规律就是从首坐标开始顺时针依次增大，代码如下：

```

#include <iostream>
using namespace std;
int a[10][10];
void Fun(int n)
{
    int m=1,j,i;
    for(i=0;i<n/2;i++)
    {
        for(j=0;j<n-i;j++)
        {
            if(a[i][j]==0)
                a[i][j]=m++;
        }
        for(j=i+1;j<n-i;j++)
        {
            if(a[j][n-1-i]==0)
                a[j][n-1-i]=m++;
        }
        for(j=n-i-1;j>i;j--)
        {
            if(a[n-i-1][j]==0)
                a[n-i-1][j]=m++;
        }
        for(j=n-i-1;j>i;j--)
        {
            if(a[j][i]==0)
        }
    }
}

```

```

        a[j][i]=m++;
    }
}
if(n%2==1)
    a[n/2][n/2]=m;
}

main(void)
{
    int n,i,j;
    cin >> n;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
            a[i][j]=0;
    }
    Fun(n);
    for(i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            cout <<a[i][j] << " ";
        }
        cout <<endl;
    }
}

```

面试例题 2：输入一个 n，输出 2 到 n 的具体素数值 [美国某著名企业 2012 年 10 月面试题]

答案：代码如下：

```
#include <stdio.h>
```

```
#include <algorithm>
```

```
#include <cmath>

int judge(int a)
{
    int j;
    for(j=2;j<=sqrt(a);j++)
    {
        if(a%j==0)//非素数，退出
            return 1;
    }
    return 0;
}

int main()
{
    int i;
    for(i=1;i<100;i++)
    {
        if(judge(i)==0)
            printf("%d ",i);
    }
    return 0;
}
```

8.5 概率

面试例题 1：Please write out the program output. (写出下面程序的运行结果。) [德国某著名软件咨询企业 2005 年 10 月面试题]

```
#include <stdlib.h>
#define LOOP 1000
void main()
{
    int rgnC=0;
    for(int i=0;i<LOOP;i++)
    {
        int x=rand();
        int y=rand();
        if(x*x+y*y < RAND_MAX*RAND_MAX)
            rgnC++;
    }
    printf("%d\n",rgnC);
}
```

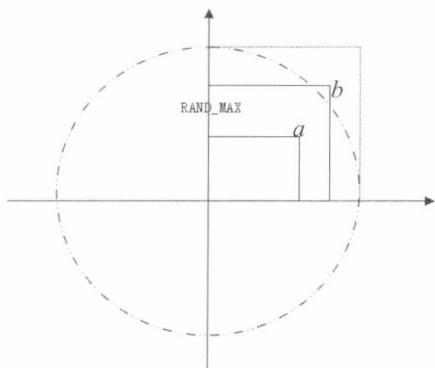
解析：这是我所见到的概率面试例题中出得非常好的一道。

从表面上看，你完全无法看出它是一个概率问题。

这里暗含的思想是一个 $1/4$ 圆和一个正方形比较大小的问题，如右图所示。

`RAND_MAX` 是随机数中的最大值，也就是相当于最大半径 R 。 x 和 y 是横、纵坐标上的两点，它们的平方和开根号就是原点到该点 (x,y) 的距离，当然这个距离有可能大于 R ，如 b 点，还有可能小于 R ，如 a 点。整个题目就蜕化成这样一个问题：随机在正方形里落 1 000 个点，落在半径里面的点有多少个。如果落在里面一个点，则累积一次。

那这个问题就很好解决了，求落点可能性之比，就是求一个 $1/4$ 圆面积和一个正方形面积之比。



1/4 圆面积 = $(1/4) \times \pi \times r \times r$; 正方形面积 = $r \times r$

两者之比 = $\pi/4$; 落点数 = $\pi/4 \times 1000 = 250 \times \pi \approx 785$

答案：出题者的意思显然就是要求你得出一个大概值，也就是 $250 \times \pi$ 就可以了。实际上呢，你只要回答 700~800 之间都是正确的。

我们算的是落点值，落点越多，越接近 $250 \times \pi$ ，落 10 个点、100 点都是很不准确的，所以该题落了 1 000 个点。读者可以上机验证该程序，将 Loop 改成 10 000、100 000 来试验，会发现结果越来越接近 $250 \times \pi$ 。

第 9 章

STL 模板与容器

标准模板库（Standard Template Library，STL）是当今每个从事 C++ 编程的人需要掌握的一项有用的技术。最近各种外企的面试题中，它的比例逐渐增大。想学习 STL 的人应该花费一段时间来熟悉它，有一些命名是不太容易凭直觉就能够记住的。然而如果一旦你掌握了 STL，就不会觉得头痛了。和 MFC 相比，STL 更加复杂和强大。

STL 有以下优点：

- 可以方便、容易地实现搜索数据或对数据排序等一系列的算法。
- 调试程序时更加安全和方便。
- 即使是人们用 STL 在 UNIX 平台下写的代码，也可以很容易地理解（因为 STL 是跨平台的）。

STL 中一些基础概念的定义如下。

- 模板（Template）：类（及结构等各种数据类型和函数）的宏（macro）。有时叫作甜饼切割机（cookie cutter），正规的名称应叫作泛型（generic）。一个类的模板叫作泛型类（generic class），而一个函数的模板也自然而然地被叫作泛型函数（generic function）。
- STL 标准模板库：一些聪明人写的一些模板，现在已成为每个人所使用的标准 C++ 语言中的一部分。
- 容器（Container）：可容纳一些数据的模板类。STL 中有 vector、set、map、multimap 和 deque 等容器。
- 向量（Vector）：基本数组模板，这是一个容器。
- 游标（Iterator）：这是一个奇特的东西，它是一个指针，用来指向 STL 容器中的元素，也可以指向其他的元素。

9.1 向量容器

面试例题1：介绍一下STL和包容器，如何实现？举例实现vector。[美国某著名移动通信企业面试题]

答案：C++的一个新特性就是采用了标准模板库(STL)。所有主要编译器销售商现在都把标准模板库作为编译器的一部分进行提供。标准模板库是一个基于模板的容器类库，包括链表、列表、队列和堆栈。标准模板库还包含许多常用的算法，包括排序和查找。

标准模板库的目的是提供对常用需求重新开发的一种替代方法。标准模板库已经经过测试和调试，具有很高的性能并且是免费的。最重要的是，标准模板库是可重用的。当你知道如何使用一个标准模板库的容器以后，就可以在所有的程序中使用它而不需要重新开发了。

容器是包容其他对象的对象。标准C++库提供了一系列的容器类，它们都是强有力地工具，可以帮助C++开发人员处理一些常见的编程任务。标准模板库容器类有两种类型，分别为顺序和关联。顺序容器可以提供对其成员的顺序访问和随机访问。关联容器则经过优化关键值访问它们的元素。标准模板库在不同操作系统间是可移植的。所有标准模板库容器类都在namespace std中定义。

举例实现vector如下：

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

void print(vector<int>);

int main()
{
    vector<int> vec;
    vec.push_back(34);
    vec.push_back(23);
    print(vec);
    vector<int>::iterator p;
    p=vec.begin();
    *p=68;
    *(p+1)=69;
    /*(p+2)=70;
    print(vec);
    vec.pop_back();
    print(vec);
    vec.push_back(101);
    vec.push_back(102);
```

```
int i=0;
while(i<vec.size())
    cout << vec[i++] << " ";
cout << endl;
vec[0] = 1000;
vec[1] = 1001;
vec[2] = 1002;
//vec[3] = 1002;
i=0;
while(i<vec.size())
    cout << vec[i++] << " ";

print(vec);
return 0;
}

void print(vector<int> v)
{
    cout << "\n vector size is: " << v.size()
    << endl;
    vector<int>::iterator p = v.begin();
}
```

或者是：

```
#include <iostream>
#include <vector>
using namespace std;
int sum(vector<int>vec)
{
```

```
int result = 0;
vector<int>::iterator p = vec.begin();
while(p!=vec.end())
    result +=*p++;
return result;
```

```

}
//void print(vector<int>);

int main()
{
    vector<int> v1(100);
    cout << v1.size() << endl;      //100
    cout << sum(v1) << endl;        //0
    v1.push_back(23);
    cout << v1.size() << endl;      //101
    cout << sum(v1) << endl;        //23
}

```

```

v1.reserve(1000);           //
v1[900]=900;
cout << v1[900] << endl;    //900
cout << v1.front() << endl; //0
cout << v1.back() << endl;  //23
v1.pop_back();
cout << v1.back() << endl;  //0
//vector<int>::iterator p2 = v1[0];
// vector<int>::iterator p2 = &(v1[0]);
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 2010 年面试题]

```

#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;

class CDemo {
public:
    CDemo():str(NULL) {};
    ~CDemo()
    {
        if(str) delete[] str;
    };
    char* str;
};

```

```

int main(int argc, char** argv) {
    CDemo d1;
    d1.str=new char[32];
    strcpy(d1.str, "trend micro");

    vector<CDemo*>*a1=new vector<CDemo*>();

    a1->push_back(&d1);
    delete a1;

    return EXIT_SUCCESS;
}

```

解析：这个程序在退出时会出问题。重复 delete 同一片内存，程序崩溃。如果把析构函数改为如下，可以更清楚地看到这一点：

```

~CDemo()
{
    if(str)
    {
        static int i=0;
        cout<<"&CDemo"<<i++<< "="<<(int*)this<<","
        str=<<(int *)str<<endl;
        delete[] str;
    }
}

```

```

cout<<"&CDemo"<<i++<< "="<<(int*)this<<","
str=<<(int *)str<<endl;
delete[] str;
}
}

```

运行时发现打印如下信息：

```

&CDemo0=0x3d3d68,      str=0x3d3d28
&CDemo1=0x22ff58,      str=0x3d3d28

```

也就是说，发生了 CDemo 类的两次析构，两次析构 str 所指向的同一内存地址空间（两次 str 值相同=0x3d3d28）。

有人认为，vector 对象指针能够自动析构，所以不需要调用 delete a1，否则会造成两次析构对象，这样理解是不准确的。任何对象如果是通过 new 操作符申请了空间，必须显示的调用 delete 来销毁这个对象。所以“delete a1;”这条语句是没有错误的。

“vector<CDemo> *a1=new vector<CDemo>();”指定一个指针，指向 vector<CDemo>，并用 new 操作符进行了初始化，所以必须在适当的时候释放 a1 所占的内存空间，因此，“delete

a1;”这句话是没有错误的。但必须明白一点，释放 vector 对象，vector 所包含的元素也同时被释放。

那到底哪里有错误？

这句 a1 的声明和初始化语句 “vector<CDemo> *a1=new vector<CDemo>();” 说明 a1 所含元素是 “CDemo” 类型的，在执行 “a1->push_back(d1);” 这条语句时，会调用 CDemo 的复制构造函数，虽然 CDemo 类中没有定义复制构造函数，但是编译器会为 CDemo 类构建一个默认的复制构造函数（浅复制），这就好像任何对象如果没有定义构造函数，编译器会构建一个默认的构造函数一样。正是这里出了问题。a1 中的所有 CDemo 元素的 str 成员变量没有初始化，只有一个四字节（32 位机）指针空间。

“a1->push_back(d1);”这句话执行完后，a1 里的 CDemo 元素与 d1 是不同的对象，但是 a1 里的 CDemo 元素的 str 与 d1.str 指向的是同一块内存。

我们知道，局部变量，如 “CDemo d1;” 在 main 函数退出时，自动释放所占内存空间，那么会自动调用 CDemo 的析构函数 “~CDemo”，问题就出在这里。

前面的 “delete a1;” 已经把 d1.str 释放了（因为 a1 里的 CDemo 元素的 str 与 d1.str 指向的是同一块内存），main 函数退出时，又要释放已经释放掉的 d1.str 内存空间，所以程序最后崩溃。

答案：本题问题归根结底就是浅复制和深复制的问题。如果 CDemo 类添加一个这样的复制构造函数就可以解决问题：

```
CDemo(const CDemo &cd)
{
    this->str = new char[strlen(cd.str)+1];
    strcpy(str, cd.str);
};
```

添加深复制后可有效解决问题。

面试例题 3：以下代码有什么问题？如何修改？[中国某著名综合软件公司 2005 年面试题]

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

void print(vector<int>);

int main()
{
    vector<int> array;
    array.push_back(1);
    array.push_back(6);
    array.push_back(6);
    array.push_back(6);
    array.push_back(3);
    //删除 array 数组中所有的 6
```

```
vector<int>::iterator itor;
vector<int>::iterator itor2;
itor=array.begin();

for(itor=array.begin();
    itor!=array.end(); )
{
    if(6==*itor)
    {
        itor2=itor;
        itor=array.erase(itor2);
    }
    itor++;
}
```

```

    }
    print(array);
    return 0;
}

void print(vector<int> v)

```

解析：

这是迭代器问题，只能删除第一个 6，以后迭代器就失效了，不能删除之后的元素。

itor2=itor; 这句说明两个迭代器是一样的。array.erase(itor2); 等于 array.erase(itor);，这时指针已经指向下一个元素 6 了。itor++; 又自增了，指向了下一个元素 3，略过了第二个 6。

答案：

修改方法 1：使用 vector 模板里面的 remove 函数进行修改，代码如下：

```

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

void print(vector<int>);

int main()
{
    vector<int> array;
    array.push_back(1);
    array.push_back(6);
    array.push_back(6);
    array.push_back(3);
    //删除 array 数组中所有的 6
    vector<int>::iterator itor;

```

```

    {
        cout << "\n vector size is: " << v.size()
        << endl;
        vector<int>::iterator p = v.begin();
    }

```

```

    vector<int>::iterator itor2;
    itor=array.begin();

    array.erase( remove( array.begin(),
                        array.end(), 6 ), array.end() );

    print(array);
    return 0;
}

void print(vector<int> v)
{
    cout << "\n vector size is:
            " << v.size() << endl;
    vector<int>::iterator p = v.begin();
}

```

修改方法 2：为了让其不略过第二个“6”，可以使“itor--；”，再回到原来的位置上。具体代码如下：

```

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

void print(vector<int>);

int main()
{
    vector<int> array;
    array.push_back(1);

    array.push_back(6);
    array.push_back(6);
    array.push_back(3);
    //删除 array 数组中所有的 6
    vector<int>::iterator itor;
    vector<int>::iterator itor2;
    itor=array.begin();
    for(itor=array.begin(); itor!=array.end(); itor++)
    {

```

```

        if(6==*itor)
        {
            itor2=itor;
            array.erase(itor2);
            itor--;
        }
        //itor--;
    }

    print(array);
    return 0;
}

void print(vector<int> v)
{
    cout << "\n vector size is:
            " << v.size() << endl;
    vector<int>::iterator p = v.begin();
}

```