第5章

指针、引用、数组

在C++语言中，指针(pointer)是一个非常重要的概念。指针是一个变量，它存储的是另一个对象的内存地址。通过使用指针，程序可以直接访问和操作内存中的数据，极大地提高了编程的灵活性和效率。在C++语言中，引用(reference)是某个已存在对象的别名(alias)。引用在创建时必须被初始化，并且一旦被初始化后，就不能再改变为引用另一个对象。这意味着引用总是指向它最初被绑定的那个对象。数组(array)是一种数据结构，它允许存储固定大小的同类型元素集合。每个元素都可以通过索引(index)来访问，索引通常是从0开始的整数。本章主要介绍指针、引用、数组的概念及其应用。

5.1 指针变量的定义与初始化

5.1.1 指针变量和间址

变量的地址是指变量在存储空间中第一个字节的地址，是一个整数。可以把这个地址存放在另一个变量中。能够存放地址的变量称为“指针类型变量”，简称“指针变量”。

指针类型变量定义形式为

```
类型 * 标识符;
```

其中，“类型”是指针所关联的数据类型，可以是任何类型，包括基本数据类型(如int、float、char等)、复合数据类型或自定义数据类型(如数组、结构体、类等)。*为指针类型说明符，说明以“标识符”命名的变量用于存放“类型”对象的地址。下面都是合法的指针定义：

```
int     * iptr1, * iptr2;          //iptr1 和 iptr2 是指向 int 型对象的指针
double  * dptr;                    //dptr 是一个指向 double 型对象的指针
float   * fptr;                    //fptr 是一个指向 float 型对象的指针
char    * cptr;                    //cptr 是一个指向 char 型对象的指针
```

不管指针是指向整型、浮点型、字符型，还是其他的数据类型，指针变量所表示的内存空间都只能存储一个内存地址。指针所关联的类型决定了通过指针变量访问对象时的读取方式，即读取内存的字节数，如果一个指针变量关联类型为int，则通过指针变量访问对象时，读取从指针指示的位置开始的连续4字节，并按int型数据进行解释。

指针在声明时可以初始化，也可以在声明后进行赋值。如：

```
int var = 18;
int * iptr = &var;                    //iptr 指向 int 型变量 var,存放 var 的地址
int * iptr1;                          //定义指针 iptr1
iptr1 = iptr;                         //将 iptr 中 var 的地址赋给 iptr1,iptr1 也指向变量 var
iptr1 = var;                          //非法:不能将 int 型变量赋给指针
```

由于指针的类型决定了指针所指对象的类型，因此，初始化或者赋值时必须保证类型匹配，指针只能初始化或赋值为同类型的变量地址或者指针。如：

```
double dval_1;
double * dptr1 = &dval_1;             //合法:用 double 型变量地址初始化 dptr1
double * dptr2 = dptr1;               //合法:用 double 型指针初始化 dptr2
int * iptr = dptr1;                   //非法:iptr 和 dptr1 所指向对象的类型不同
iptr = &dval_1;                       //非法:试图将一个 double 型变量的地址赋给 int 型指针
```

对变量的访问可以通过指针变量间接实现。间址运算符(*)用于访问指针指向的内存地址中存储的值。如果 iptr 是一个指针，那么 * iptr 就是访问 iptr 指向的内存地址中存储的值。用 * iptr 的这种访问方式，称为间接地址访问，简称间址访问；通过变量名访问对象，称为名访问。

若有“int * iptr=&var;”，则 var 的地址可以表示为 &var 或 iptr，对 var 的访问可以表示为 var(名访问)或 * iptr(间址访问)。

例 5.1 测试指针变量访问所指对象。

```
1   //exp5 - 1.cpp:测试指针变量访问所指对象
2   #include < iostream >
3   using namespace std;
4   int main()
5   {
6       int val_1 = 10;
7       int val_2 = 20;
8       int * ptr1 = &val_1;                    //将指针 ptr1 指向变量 val_1
9       int * ptr2 = &val_2;                    //将指针 ptr2 指向变量 val_2
10      * ptr1 = * ptr1 + * ptr2;               //间址访问 val_1 和 val_2
11      cout <<"Val_1:"<< val_1 << endl;        //名访问 val_1
12      cout <<"The Address of Val_1:"<< ptr1 << endl;
13      cout <<"Val_2:"<< val_2 << endl;        //名访问 val_2
14      cout <<"The Address of Val_2:"<< ptr2 << endl;
15  }
```

程序运行结果：

```
Val_1: 30
The Address of Val_1: 0x61ff04
Val_2: 20
The Address of Val_2: 0x61ff00
```

第 6 行和第 7 行定义和初始化了整型变量 val_1 和 val_2，第 8 行和第 9 行定义了指针变量 ptr1 和 ptr2，分别指向 val_1 和 val_2，第 10 行通过指针变量的间址访问，修改了 ptr1 所指变量 val_1 的值。

5.1.2 空指针

在C++编程中，空指针扮演着至关重要的角色，它是一种特殊的指针变量，不指向任何有效的内存地址(无论是对象还是函数)。空指针的主要用途在于初始化指针变量，这一做法能有效避免指针在未赋值时意外指向未定义的内存区域，从而防止潜在的内存访问错误和程序崩溃。

自C++11标准起，为了定义一个空指针，推荐使用nullptr关键字，它提供了类型安全和语义上的清晰性。然而，在C++11之前的代码中，或者为了兼容旧版代码，也可以使用整数0或者宏null(通常定义为0)来初始化空指针。尽管0和null在功能上与nullptr相似，但nullptr能够提供更严格的类型检查，减少因类型不匹配而导致的编译错误，因此在现代C++编程中被视为最佳实践。

例5.2 测试空指针。

```
1   //ex5-2.cpp:测试空指针
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int *ptr=nullptr;                    //定义空指针
7       if(ptr==nullptr){
8           cout<<"The pointer is null."<<endl;
9       }
10      else{
11          cout<<"The pointer is not null."<<endl;
12      }
13      int value=10;
14      ptr=&value;                          //为空指针赋值
15      if(ptr!= nullptr){
16          cout<<"Value pointed by ptr:"<<*ptr<<endl;
17      }
18      else{
19          cout<<"The pointer is null."<<endl;
20      }
21  }
```

程序运行结果：

```
The pointer is null.
Value pointed by ptr: 10
```

第6行定义了一个指向int类型的空指针ptr。第7～12行检查这个指针是否为空，并输出相应的消息。第14行ptr被赋值为一个有效的内存地址，即变量value的地址，第15～20行再次检查指针是否为空，第16行通过间址访问输出value的值。

5.1.3 void*指针

在C++中，void*指针是一种特殊的指针类型，它可以指向任何类型的数据。由于没有关联类型，编译器无法解释所指对象，因此，void*指针不具有类型安全，在程序中必须对其做显式类型转换，才可以按指定类型访问数据。

例 5.3 测试 void * 指针。

```
1   //ex5-3.cpp:测试 void* 指针
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int intValue=100;
7       double doubleValue=3.14;
8       void *ptr=nullptr;                    //定义一个 void* 指针
9       ptr=&intValue;                        //指向 int 型变量
10      cout<<"As int:"<<*(int *)ptr<<endl;
11      ptr=&doubleValue;                     //指向 double 型变量
12      cout<<"As double:"<<*(double *)ptr<<endl;
13      ptr=nullptr;                          //再次将 void* 指针设置为 nullptr
14      return 0;
15  }
```

程序运行结果：

```
As int: 100
As double: 3.14
```

第 8 行定义了一个 void * 类型的指针 ptr，让这个指针依次指向一个整数(第 9 行)和一个双精度浮点数(第 11 行)。每次指向不同类型的变量时，都需要将 void * 显式转换为相应的类型，才能正确访问对象。这个程序展示了 void * 指针的灵活性，但也展示了其使用时需要的谨慎性，因为不正确的类型转换可能导致访问出错。

5.1.4 指向指针的指针

指针用来存储变量的地址。而指向指针的指针，顾名思义，就是存储指针变量地址的指针。如果把指针看作指向一个数据存储位置的箭头，那么指向指针的指针就是指向这个箭头的另一个箭头。例如：

```
int **  iptr2;                    //声明一个指向 int* 类型的指针，即指向指针的指针
int *  iptr1;                     //声明一个指向 int 类型的指针
int i=3;                          //声明一个整型变量 i，并初始化为 3
iptr1=&i;                         //将 i 的地址赋值给 iptr1，此时 iptr1 指向 i
iptr2=&iptr1;                     //将 iptr1 的地址赋值给 iptr2，此时 iptr2 指向 iptr1，
                                  //即指向一个指向 int 类型的指针
```

例 5.4 测试多级指针。

```
1   //ex5-4.cpp:测试多级指针
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int **** p4, *** p3, ** p2, * p1, i=3;
                                  //声明 4 个指针变量和一个整型变量 i，并初始化 i 为 3
7       p4=&p3;                   //p4 指向 p3
8       p3=&p2;                   //p3 指向 p2
9       p2=&p1;                   //p2 指向 p1
10      p1=&i;                    //p1 指向 i
```

```
11      cout << **** p4 << endl;        //间址访问 i,打印出 i 的值
12  }
```

程序运行结果：

```
3
```

例 5.4 展示了如何使用四级指针。程序的目的是输出变量 i 的值，通过逐层间址访问这些指针来实现。因为 p4 指向 p3，p3 指向 p2，p2 指向 p1，而 p1 指向 i。所以 **** p4 实际上就是 i 的值。这个程序展示了指针和多级指针的基本概念，以及如何通过逐层间址访问来访问变量的值。

在实际编程中，使用如此多级指针的情况相对较少，通常在处理一些非常复杂的内存结构或特定的数据结构（如复杂的树状结构等）时才可能会用到类似的多级指针操作。

5.1.5 指向常量的指针

指向常量的指针也称为常量指针。定义形式为

```
const 类型 * 指针
```

或者

```
类型 const * 指针
```

常量指针可以获取变量或者常量的地址，但限制用指针间址访问对象方式为“只读”。

例 5.5 测试指向常量的指针。

```
1   //ex5 - 5.cpp:测试指向常量的指针
2   #include < iostream >
3   using namespace std;
4   int main()
5   {
6       const int val_1 = 10;
7       int val_2 = 15;
8       const int * ptr1 = &val_1;      //定义常量指针,指向常量
9       const int * ptr2 = &val_2;      //定义常量指针,指向变量
10      //int * ptr3 = &val_1;          //错误:不能用普通指针指向常量 val_1
11      // * ptr1 = 20;                 //错误:不能通过指向常量的指针修改常量 val_1
12      // * ptr2 = 20;                 //错误:不能通过指向常量的指针修改变量 val_2
13      val_2 = 20;                     //可以直接修改变量值
14      cout <<"Val_1:"<< * ptr1 << endl;
15      cout <<"Val_2:"<< * ptr2 << endl;
16      return 0;
17  }
```

程序运行结果：

```
Val_1: 10
Val_2: 20
```

例 5.5 展示了如何定义和使用指向常量的指针，ptr1 是一个常量指针，不能通过间址访问来修改所指向常量的值，ptr2 也是一个常量指针，它指向变量时，不能通过间址访问来修改所指变量的值，但可以通过名访问修改变量的值。

注意：把一个 const 对象的地址赋给一个普通的、非 const 对象的指针会导致编译

错误。

5.1.6 指针常量

指针常量的含义是，这个指针本身的值不能改变，即不能指向另一个地址。

指针常量的定义形式为

```
类型 * const 指针
```

const 写在“指针”之前，表示约束对指针本身的访问为只读。因为指针常量对间址访问没有约束，所以，指针常量不能指向常量。

例 5.6 测试指针常量。

```
1   //ex5 - 6.cpp:测试指针常量
2   #include < iostream >
3   using namespace std;
4   int main()
5   {
6      int val_1 = 10;
7      int val_2 = 20;
8      int * const ptr = &val_1;        //定义指针常量
9      //ptr = &val_2;                  //错误:不能改变指针常量的指向
10     * ptr = 30;                      //间址访问 val_1
11     cout <<"Val_1:"<< val_1 << endl;
12    return 0;
13  }
```

程序运行结果：

```
Val_1: 30
```

例 5.6 展示了如何定义和使用指针常量。ptr 是一个指针常量，它本身是常量，所以它的值(即它所指向的地址)不能改变，但可以通过它来修改所指向的值。

5.1.7 指向常量的指针常量

指向常量的指针常量，指针既不能改变指向，也不能通过它来修改它所指向的数据。这种指针既可以指向常量也可以指向变量。

指向常量的指针常量的定义形式为

```
const 类型 * const 指针
```

或者

```
类型 const * const 指针
```

例 5.7 测试指向常量的指针常量。

```
1   //ex5 - 7.cpp:测试指向常量的指针常量
2   #include < iostream >
3   using namespace std;
4   int  main()
5   {
6      const  int  MIN = 10;
7      int  max;
```

```
8       const int * const   pmax = &max;     //定义指向常量的指针常量 pmax,初始化为 max 的地址
9       // * pmax = 1000;                     //非法:间址访问约束为只读
10      max = 1000;                           //合法:修改变量值
11      cout <<" max = "<< * pmax << endl;
12      const   int * const pmin = &MIN;      //定义指向常量的指针常量 pmin,初始化为 min 的地址
13      //pmin = &max;                        //非法:不能修改指针常量
14      // * pmin = 0;                        //非法:间址访问约束为只读
15      cout <<"MIN = "<< * pmin << endl;
16  }
```

程序运行结果：

```
max = 1000
MIN = 10
```

在 main()函数中，第 8 行和第 12 行定义了两个指向常量的指针常量 pmax 和 pmin，分别指向可修改的普通变量 max 和不可修改的常量 MIN。第 9 行尝试通过 * pmax＝1000 来修改 pmax 所指向的值是非法的，因为 pmax 是指向常量的指针常量，虽然它指向的是可修改的变量 max。而直接修改变量 max 的值，如第 10 行 max＝1000 是合法的。第 13 行尝试通过 pmin＝&max 来改变 pmin 的指向是非法的，因为 pmin 是指向常量的指针常量；尝试通过 * pmin＝0 来修改 pmin 所指向的值也是非法的，因为 pmin 指向的是常量 MIN，其值是不能修改的。

5.2 引用

5.2.1 引用的定义

在 C++中，引用是一种特殊的变量类型，它是一个已存在对象的别名。引用本身不是一个独立的对象，它只是一个别名，所以在定义引用时必须进行初始化，并且一旦初始化后就不能重新绑定到另一个对象。定义引用的语法格式为

```
类型 & 引用名 = 对象名;
```

其中，& 为引用说明符。例如：

```
int a;
int * ptr;
int &refToA = a;                          //refToA 是 a 的引用,只能在定义时初始化
ptr = &a;                                 //ptr 指向 a
```

5.2.2 常引用

常引用是冠以 const 定义的引用，将约束对象用别名方式访问时为只读。常引用的定义形式为

```
const 类型 & 引用名 = 对象名;
```

例如：

```
int val = 356;
const int &ref_val = val;                 //ref_val 是 val 的常引用
ref_val = 450;                            //非法:不能通过常引用修改对象 val 的值
```

```
val = 450;                                    //合法:可以直接修改变量的值
```

例 5.8 引用测试。

```
1   //ex5 - 8.cpp:引用测试
2   #include < iostream >
3   using namespace std;
4   int main()
5   {
6       int a = 10;
7       int b = 20;
8       int& refToA = a;                      //refToA 是 a 的别名
9       int& refToB = b;                      //refToB 是 b 的别名
10      refToA = 100;                         //通过引用修改变量
11      refToB = 200;                         //通过引用修改变量
12      cout <<"a:"<< a << endl;              //输出 100,因为 refToA 是 a 的别名
13      cout <<"b:"<< b << endl;              //输出 200,因为 refToB 是 b 的别名
14      const int& constRefToA = a;           //定义一个常引用
15      //constRefToA = 1000;                 //尝试修改常引用绑定的值(编译错误)
16  }
```

程序运行结果:

```
a: 100
b: 200
```

例 5.8 中第 6～9 行定义了两个整型变量 a 和 b,然后定义了两个引用 refToA 和 refToB,分别绑定到 a 和 b。第 10 行和第 11 行通过引用 refToA 和 refToB 来修改 a 和 b 的值。第 14 行定义了一个常引用 constRefToA 绑定到 a,尝试修改常引用绑定的值会导致编译错误,因为常引用不能用来修改它绑定的对象。

5.3 指针、引用与函数

5.3.1 函数参数的指针传递与引用传递

函数定义中的形参被声明为指针类型时,称为指针参数。形参指针对应的实参是地址表达式。调用函数时,实参把对象的地址赋给形参名标识的指针变量,被调用函数可以在函数体内通过形参指针来间接访问实参地址所指向的对象。这种参数传递方式称为指针传递。

函数定义中的形参被声明为引用类型,称为**引用参数**。引用参数对应的实参应该为对象名。函数被调用时,形参不需要开辟新的存储空间,形参名作为引用(别名)绑定在实参标识的对象上。执行函数体时,对形参的操作就是对实参对象的操作。函数执行结束,撤销引用绑定。

例 5.9 函数参数的三种传递方式。

```
1   //ex5 - 9.cpp:函数参数的三种传递方式
2   #include < iostream >
3   using namespace std;
4   void swapByValue(int a,int b)             //值传递
5   {
6       int temp = a;
7       a = b;
```

```
8       b = temp;
9       cout <<"Inside swapByValue:a = "<< a <<", b = "<< b << endl;
10  }
11  void swapByPointer(int * a, int * b)  //指针传递
12  {
13      int temp = * a;
14      * a = * b;
15      * b = temp;
16      cout <<"Inside swapByPointer:a = "<< * a <<", b = "<< * b << endl;
17  }
18  void swapByReference(int& a, int& b)  //引用传递
19  {
20      int temp = a;
21      a = b;
22      b = temp;
23      cout <<"Inside swapByReference:a = "<< a <<", b = "<< b << endl;
24  }
25  int main()
26  {
27      int x = 10;
28      int y = 20;
29      cout <<"Before swap:x = "<< x <<", y = "<< y << endl;
30      //调用值传递函数,不会改变 x 和 y 的值
31      swapByValue(x,y);
32      cout <<"After swap by swapByValue:x = "<< x <<", y = "<< y << endl;
33      //调用指针传递函数,会改变 x 和 y 的值
34      swapByPointer(&x,&y);
35      cout <<"After swap by swapByPointer:x = "<< x <<", y = "<< y << endl;
36      //调用引用传递函数,会改变 x 和 y 的值
37      swapByReference(x,y);
38      cout <<"After swap by swapByReference:x = "<< x <<", y = "<< y << endl;
39  }
```

程序运行结果：

```
Before swap: x = 10, y = 20
Inside swapByValue: a = 20, b = 10
After swap by swapByValue: x = 10, y = 20
Inside swapByPointer: a = 20, b = 10
After swap by swapByPointer: x = 20, y = 10
Inside swapByReference: a = 10, b = 20
After swap by swapByReference: x = 10, y = 20
```

第 4～10 行的函数 swapByValue()有两个整型参数 a 和 b，这两个参数是通过值传递的。第 31 行调用 swapByValue()函数，将实参 x 和 y 的值分别赋值给形参 a 和 b，然后执行函数体，函数内部对 a 和 b 的任何修改都不会影响 main()函数中的变量 x 和 y，所以第 32 行输出的值仍然是 10 和 20。

第 11～17 行的函数 swapByPointer()有两个整型指针参数 a 和 b，这两个参数是通过指针传递的。第 34 行调用 swapByPointer()函数，将实参 x 和 y 的地址分别传递给形参指针 a 和 b，在函数体中通过形参指针直接操作实参，交换 * a 和 * b 的值，所以第 35 行输出的值变成了 20 和 10。

第 18～24 行的函数 swapByReference()有两个整型引用参数 a 和 b，这两个参数是通

过引用传递的。第 37 行调用 swapByReference()函数,将实参 x 和 y 分别与形参 a 和 b 绑定,a 作为 x 的别名,b 作为 y 的别名,在函数体中通过别名交换了 x 和 y 的值,所以第 38 行输出的值又变成了 10 和 20。

由例 5.9 可以看出,使用指针传递和引用传递,都可以直接在实参对象上操作,因此,可以直接改变实参对象。当不希望改变实参对象时,可以用 const 来保护实参。

例 5.10 使用 const 限定指针参数,保护实参对象。

```
1   //ex5-10.cpp:使用 const 限定指针参数,保护实参对象
2   #include<iostream>
3   using namespace std;
4   //函数声明:通过指针传递整数,并使用 const 保护它不被修改
5   void processValue(const int* ptr,int count){
6       int localSum=0;
7       for(int i=0;i<count;++i){
8           cout<<"Value:"<<*ptr<<",Sum so far:"<<localSum<<endl;
9           localSum +=  *ptr;
10      }
11      cout<<"Final sum after accumulating "<<count<<" times:"<<localSum<<endl;
12      //尝试修改 ptr 所指对象的值,这将导致编译错误
13      //*ptr=20;            //错误:ptr 是指向常量的指针,不能通过它来修改所指对象的值
14  }
15  int main(){
16      int value=10;
17      int count=5;
18      processValue(&value,count);
19      //尝试直接在 main()函数中修改 value,以展示保护效果
20      //value=30;                    //这行代码是合法的,因为它是在 value 的作用域内
21  }
```

程序运行结果:

```
Value:10,Sum so far:0
Value:10,Sum so far:10
Value:10,Sum so far:20
Value:10,Sum so far:30
Value:10,Sum so far:40
Final sum after accumulating 5 times:50
```

在第 5～14 行的 processValue()函数中,参数 const int* ptr 是一个指向整数的常量指针。使用 const 关键字表明,函数内部不能通过这个指针修改它所指对象的值,这保护了实参 value,确保它在函数调用期间不会被意外修改。第 18 行,在 main()函数中调用“processValue(&value,count);”时,传递的是 value 的地址。由于 ptr 是 const int*,因此任何尝试通过 ptr 修改 *ptr 的操作都会导致编译错误。

通过使用 const 关键字,能够有效地保护函数参数,防止在函数内部对实参的意外修改。尝试修改受保护实参的语句会导致编译错误,从而保护了实参的完整性。这种做法提高了代码的安全性和可读性。

例 5.11 使用 const 限定引用参数,保护实参对象。

```
1   //ex5-11.cpp:使用 const 限定引用参数,保护实参对象
```

```
2   #include <iostream>
3   using namespace std;
4   //函数声明:通过引用传递整数,使用 const 保护它不被修改
5   void processValue(const int& value){
6       cout <<"Value:"<< value << endl;
7       //尝试修改 value 所引用的值,这将导致编译错误
8       //value = 20;                          //错误:value 是 const 引用,不能通过它来修改值
9   }
10  void processExpression(const int& value){
11      cout <<"Expression Value:"<< value << endl;
12  }
13  int main(){
14      const int constantValue = 10;
15      processValue(constantValue);          //传递常量
16      int variableValue = 20;
17      processValue(variableValue);          //传递变量
18      processExpression(5 + 3);             //传递表达式
19      int anotherVariable = 30;
20      processValue(anotherVariable + 10);   //表达式结果作为实参
21  }
```

程序运行结果：

```
Value:10
Value:20
Expression Value:8
Value:40
```

第 5～9 行的 processValue()函数接收一个 const int& 类型的参数，这意味着它是一个对整数的常引用参数，在函数体内不能修改实参对象；第 15 行和第 17 行分别传递 constantValue 和 variableValue 给函数 processValue()时，由于第 5 行的形参 value 是常引用，因此任何尝试修改它的操作都会导致编译错误。

第 20 行将表达式 anotherVariable+10 的值与 processValue()函数的形参 value(const int& 类型)绑定，第 18 行调用的 processExpression()函数同样接收一个 const int& 类型的参数，传递一个表达式(5+3)给形参 value。当实参为常量或表达式时，形参的引用参数必须加 const，即为常引用参数。

通过使用常引用参数可以保护函数参数不被修改，同时接收常量、变量，甚至是临时表达式作为实参。这种灵活性使得常引用参数在 C++中非常有用，尤其是在需要保护数据不被修改，同时又希望函数能够接收多种类型的参数时。

5.3.2 函数指针

1. 函数的地址

在 C++中，函数被视为存储在内存中的一段可执行代码，每个函数都有一个唯一的地址。对于已经定义的函数，函数的名称就是函数的入口地址，调用函数的语句为：

```
函数地址(实参表)
```

其中，“函数地址”实质上是一个地址表达式，可以是函数名，也可以是能够表示函数入口地址的表达式。

例 5.12 函数地址测试程序。

```
//ex5-12.cpp:函数地址测试程序
#include <iostream>
using namespace std;
void simple()
{
    cout <<"This is a simple program."<< endl;
}
int main()
{
    cout <<"Call function:"<< endl;
    simple();                          //名方式调用
    (& simple)();                      //地址方式调用
    ( * &simple)();                    //间址方式调用
    cout <<"Address of function:"<< endl;
    cout << simple << endl;            //函数名是地址
    cout << &simple << endl;           //取函数地址
    cout << * &simple << endl;         //函数地址所指对象
}
```

程序运行结果：

```
Call function:
This is a simple program.
This is a simple program.
This is a simple program.
Address of function:
00007FF60963146A
00007FF60963146A
00007FF60963146A
```

从例 5.12 可以看出，simple、&simple、* &simple 都是函数的入口地址，以下调用语句可以达到相同的效果：

```
simple();
(&simple)();
( * &simple)();
```

注意：后两种调用方式中，第一个括号不能省略，它使得地址表达式的计算先于参数结合。

2. 函数的类型

函数的类型是指函数的接口，包括函数的参数定义和返回类型。例如，以下函数的函数类型相同：

```
double max(double,double);
double min(double,double);
double average(double,double);
```

这些函数都有两个 double 类型参数，返回值为 double 类型，它们的类型为

```
double(double,double)
```

一般地，表示函数类型的形式为

```
类型(形参表)
```

C++中,可以用关键字 typedef 定义函数类型名。函数类型名定义的一般形式为

```
typedef 类型 函数类型名(形参表);
```

其中,“函数类型名”是用户定义标识符。例如:

```
typedef double functionType(double,double);   //定义 functionType 为 double(double,double)类型
```

此时,如果定义:

```
functionType  max,min,average;
```

则等价于前面的 3 个函数原型声明。

3. 函数指针

函数指针是表示函数入口地址的指针。要定义指向某一类函数的指针变量,可以用以下两种说明语句:

```
类型(*指针变量名)(形参表);
```

或

```
函数类型 *指针变量名;
```

对于上述已经定义的 functionType,可以定义指向这一类函数的指针变量:

```
double(*fp)(double,double);
```

或

```
functionType *fp;
```

都是定义一个函数指针变量 fp,可以存放一个 double(double,double)类型函数的入口地址。

注意:以下两个语句说明有不同的含义。

```
double(*fp)(double,double);          //定义指向 double(double,double)类型的函数指针 fp
double *fp(double,double);           //定义函数 fp()的原型,其返回值为 double 类型的指针
```

例 5.13 用函数指针调用不同函数。

```
1   //ex5-13.cpp:用函数指针调用不同函数
2   #include<iostream>
3   using namespace std;
4   int add(int a,int b){
5       return a+b;
6   }
7   int substract(int a,int b){
8       return a-b;
9   }
10  int multiply(int a,int b){
11      return a*b;
12  }
13  typedef int functionType(int,int);                  //定义函数类型
14  functionType*operatorp;                             //函数指针声明
15  int main()
```

```
16  {
17      int num1 = 10,num2 = 5;
18      operatorp = add;                                    //函数指针指向 add()函数
19      cout <<"Add:"<< operatorp(num1,num2)<< endl;        //使用函数指针调用 add()函数
20      operatorp = substract;                              //函数指针指向 substract()函数
21      cout <<"Substract:"<< operatorp(num1,num2)<< endl; //使用函数指针调用 substract()函数
22      operatorp = multiply;                               //函数指针指向 multiply()函数
23      cout <<"Multiply:"<< operatorp(num1,num2)<< endl;  //使用函数指针调用 multiply()函数
24  }
```

程序运行结果：

```
Add: 15
Substract: 5
Multiply: 50
```

例 5.13 中第 4～12 行定义了三个类型相同的函数 add()、substract()和 multiply()，分别实现加、减和乘运算。第 14 行定义函数指针 operatorp，第 18 行、第 20 行和第 22 行分别将函数指针指向不同的函数，实现不同的运算。

例 5.14 使用函数指针参数调用函数。

```
1   //ex5 - 14.cpp:使用函数指针参数调用函数
2   #include < iostream >
3   using namespace std;
4   int add(int a,int b){
5       return a + b;
6   }
7   int substract(int a,int b){
8       return a - b;
9   }
10  int multiply(int a,int b){
11      return a * b;
12  }
13  typedef int functionType(int,int);                      //定义函数类型
14  int operate(int a,int b,functionType  * operatorp)      //定义一个函数,使用函数指针
                                                            //operatorp 作为参数
15  {
16   return operatorp(a,b);}
17  int main()
18  {
19      int num1 = 10,num2 = 5;
20      cout <<"Add:"<< operate(num1,num2,add)<< endl;   //函数指针指向 add,调用 add()函数
21      cout <<"Substract:"<< operate(num1,num2,substract)<< endl; //函数指针指向 substract,
                                                                   //调用 substract()函数
22      cout <<"Multiply:"<< operate(num1,num2,multiply)<< endl;   //函数指针指向 multiply,
                                                                   //调用 multiply()函数
23  }
```

程序运行结果：

```
Add: 15
Substract: 5
Multiply: 50
```

函数指针的一个重要用法是用于函数参数。例 5.14 中第 14～16 行定义的 operate 是一个通用函数，接收两个整型参数 a 和 b，以及一个函数指针 operatorp。函数指针 operator

的类型是 int(*)(int,int),表示它指向具有两个整型参数并返回一个整数的函数。

5.4 数组

数组是由一定数目的同类型元素顺序排列而成的自定义数据类型。在计算机中,一个数组在内存中占据一片连续的存储区域,数组名就是这块存储空间的首地址。

5.4.1 一维数组

1. 一维数组的定义

一维数组的说明格式为

```
类型  标识符[下标表达式];
```

其中,“标识符”是用户自定义的数组名;“类型”规定了存放在数组中的元素的类型,数组定义中的类型名可以是内置数据类型,也可以是用户定义的数据类型,如类类型;“下标表达式”指定数组元素的个数,必须是大于或等于 1 的常量表达式,该常量表达式只能包含整型常量、枚举常量或者用常量表达式初始化的整型常量对象。例如:

```
const int max_size = 512,min_size = 20; //定义常量 max_size 和 min_size
int arr_size = 30;                      //定义变量 arr_size
const int brr_size = get_size();        //常量 brr_size 的值需要运行时通过 get_size()函数返回
char in_buffer[max_size];               //合法:max_size 为常量
float out_buffer[min_size + 1];         //合法:常量表达式
double salaries[arr_size];              //非法:arr_size 不是常量
int test_values[brr_size];              //非法:brr_size 的值需要运行时才给出
int buf_size[get_size()];               //非法:不是常量表达式
```

虽然 arr_size 是用常数进行初始化,但由于 arr_size 本身不是常量,因此,用该变量定义数组维数是非法的。brr_size 虽然是 const 对象,但它的值要运行时调用 get_size()函数后才知道,因此,用它定义数组维数也是非法的。

说明数组后,C++数组元素的值是内存的随机状态值,可以在定义时为数组提供一组用逗号分隔的初值,这些初值可用{}括起来,称为初始化列表。例如:

```
int arr_1[3] = {0,1,2};                 //三个元素分别初始化为 0、1 和 2
int arr_2[3] = {2};                     //第一个元素初始化为 2,剩余元素初始化为 0
int arr_3[] = {1,2,3}                   //合法:编译器根据列出的元素个数确定数组长度为 3
int arr_4[10] = {0};                    //合法:将全部元素初始化为 0
int arr_5[3] = {1,2,3,4};               //非法:初始化元素个数超出数组长度
static int arr_8[6];                    //合法:自动把静态数组各元素值初始化为 0
const int arr_9[5] = {2,4,6,8,10};      //数组元素约束为只读
```

如果指定了数组长度,那么初始化列表提供的元素个数不能超过数组长度,如 arr_5 的初始化不合法。如果数组长度大于列出元素的初值个数,则只初始化前面的数组元素,其余元素初始化为 0。

2. 以下标方式访问一维数组

一维数组元素可用下标运算符([])来访问,数组元素下标从 0 开始,对于一个包含 10 个元素的数组,下标值为 0～9。

例 5.15 用下标运算符访问一维数组元素。

```
//ex5-15.cpp:用下标运算符访问一维数组元素
#include<iostream>
using namespace std;
int main()
{
    const int array_size=10;
    int ia[array_size];
    for(int i=0;i<array_size;i++)
        ia[i]=i;
}
```

一个数组不能用另外一个数组初始化，也不能将一个数组直接赋值给另外一个数组，但是可以使用循环，实现对应元素的复制。

例 5.16 数组的复制。

```
//ex5-16.cpp:数组的复制
#include<iostream>
using namespace std;
int main()
{
  int arr1[5]={ 1,2,3,4,5 };
  int arr2[5];
  //arr2=arr1;                    //非法:不能将一个数组直接赋值给另一个数组
  for(int i=0;i<5;++i)            //使用循环复制初始化数组
  {
      arr2[i]=arr1[i];
  }
  for(int i=0;i<5;++i)
  {
      cout<<"arr2["<<i<<"]:"<<arr2[i]<<" ";
  }
}
```

程序运行结果：

```
arr2[0]: 1 arr2[1]: 2 arr2[2]: 3 arr2[3]: 4 arr2[4]: 5
```

在使用数组时，必须保证其下标值在正确范围内，数组越界会导致程序致命的错误。

3. 以指针方式访问一维数组

在 C++ 中，数组和指针有着密切的联系，可以用指针来访问和操作数组中的元素。

对于一个已定义的数组，数组名是数组存储空间的首地址，如图 5.1 所示。因此，有

$$a==\&a[0] \quad a+1==\&a[1] \quad a+i==\&a[i]$$

$$*a==a[0] \quad *(a+1)==a[1] \quad *(a+i)==a[i]$$

上式中的==表示符号两边的内容相同。可以通过定义指针变量访问数组元素，例如：

```
int arr1[]={0,2,4,6,8};
int *ip=arr1;                     //指针 ip 指向 arr1[0]
ip=&arr1[4];                      //ip 指向 arr1 的最后一个元素
```

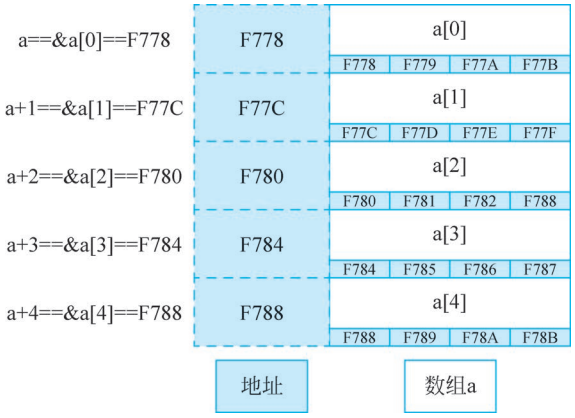

图 5.1 数组名与数组元素地址的关系

可以通过指针的算术运算来获取指定元素的存储地址。在指向数组某个元素的指针上加上(或者减去)一个整型数值,就可以计算出指向数组的另一个元素的指针值。例如:

```
ip = arr1;                          //合法:指针 ip 指向 arr1[0]
int * ip2 = ip + 4;                 //合法:ip2 指向 arr1[4]
int * ip2 = arr1 + 10;              //非法:数组的长度只有 4,arr1 + 10 为无效地址
```

要注意指针进行算术运算时不能超出数组长度,否则会出现访问越界的错误。

例 5.17 用不同的方式访问数组。

```
1   //ex5 - 17.cpp:用不同的方式访问数组
2   #include <iostream>
3   using namespace std;
4   int main()
5   {
6     int arr[] = { 10,20,30,40,50 };
7     int * ptr = arr;                                  //指针指向数组的首元素 arr[0]
8     for(int i = 0;i < 5;++i,ptr++){
9         cout <<"Element "<< i <<":"<< * ptr <<" ";    //使用指针的间址方式遍历数组
10  }
11    cout << endl;
12    ptr = &arr[0];                                    //指针重新指向数组的首元素
13    for(int i = 0;i < 5;++i){
14        cout <<"Element "<< i <<":"<< ptr[i]<<" ";    //使用指针的下标方式访问数组
15    }
16    cout << endl;
17    for(int i = 0;i < 5;++i){
18        cout <<"Element "<< i <<":"<< * (ptr + i)<< endl; //使用指针的间址方式访问数组
19    }
20    cout << endl;
21  }
```

程序运行结果:

```
Element 0: 10 Element 1: 20 Element 2: 30 Element 3: 40 Element 4: 50
Element 0: 10 Element 1: 20 Element 2: 30 Element 3: 40 Element 4: 50
Element 0: 10 Element 1: 20 Element 2: 30 Element 3: 40 Element 4: 50
```

例 5.17 展示了如何使用指针来访问数组。指针提供了一种灵活的方式来访问数组元

素，但也需要小心使用，以避免越界访问和内存泄漏等问题。

4. 指针数组

当数组元素的类型为指针类型时，称为指针数组。使用指针数组便于对一组相关对象的地址进行管理。

指针数组的说明形式为

```
类型  *标识符[表达式];
```

其中，“类型”表示指针数组元素的关联类型，可以为各种 C++ 系统允许的数据类型，包括函数类型。“标识符”表示数组名。例如：

```
int * pi[3];                                    //数组元素是关联类型为 int 的指针
double * pf[5];                                 //数组元素是关联类型为 double 的指针
char * ps[7];                                   //数组元素是关联类型为 char 的指针
```

在 C++ 中，互不相关的变量在系统中分配的内存往往是离散的，当需要对类型相同的变量进行统一处理时，可以使用指针数组管理它们的地址，通过指针数组对这些变量进行间址访问。

例 5.18 测试指向基本类型的指针数组。

```
1   //ex5-18.cpp:测试指向基本类型的指针数组
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int * ptrArray[5];                      //声明一个包含 5 个整型指针的数组
7       //创建 5 个整型变量
8       int a=10,b=20,c=30,d=40,e=50;
9       //将这些整型变量的地址赋值给指针数组的元素
10      ptrArray[0]=&a,ptrArray[1]=&b,ptrArray[2]=&c,ptrArray[3]=&d,ptrArray[4]=&e;
11      //通过指针数组访问并打印这些整型变量的值
12      for(int i=0;i<5;++i){
13          cout<<"Value at ptrArray["<<i<<"]:"<<*ptrArray[i]<<endl;
14      }
15  }
```

程序运行结果：

```
Value at ptrArray[0]: 10
Value at ptrArray[1]: 20
Value at ptrArray[2]: 30
Value at ptrArray[3]: 40
Value at ptrArray[4]: 50
```

第 6 行声明了一个名为 ptrArray 的指针数组，它能够存储 5 个指向 int 类型的指针。第 8 行创建了 5 个整型变量 a、b、c、d 和 e 并初始化，第 10 行将它们的地址赋值给 ptrArray 的相应元素。最后，第 12～14 行通过指针数组访问并打印这些整型变量的值。

当指针数组存放数组地址时，可以通过这个指针数组访问这些数组的元素。可以使用指针数组管理几个类型相同的数组。

例 5.19 利用指针数组管理数组的测试程序。

```
1   //ex5-19.cpp:利用指针数组管理数组的测试程序
2   #include<iostream>
```

```
3   using namespace std;
4   int main()
5   {
6       int *ptrArray[3];                          //声明指针数组
7       //创建3个数组
8       int arr1[3]={ 1,2,3 };
9       int arr2[3]={ 4,5,6 };
10      int arr3[3]={ 7,8,9 };
11      //将数组的地址赋值给指针数组的元素
12      ptrArray[0]=arr1;
13      ptrArray[1]=arr2;
14      ptrArray[2]=arr3;
15      //通过指针数组访问并输出数组元素的值
16      for(int i=0;i<3;++i){
17          cout<<"Array "<<i+1<<":" ;
18          for(int j=0;j<3;++j){
19              cout<<*(ptrArray[i]+j)<<" ";
20          }
21          cout<<endl;
22      }
23  }
```

程序运行结果：

```
Array 1: 1 2 3
Array 2: 4 5 6
Array 3: 7 8 9
```

例5.19中，第6行定义的ptrArray是一个长度为3的指针数组，其数组元素关联类型int*，因此，每个元素可以分别存放int类型元素的数组地址。第8～10行定义的arr1、arr2、arr3是长度为3、元素类型为int的数组。第12～14行将数组arr1、arr2、arr3的地址分别赋值给指针数组ptrArray的三个元素。第16～22行通过指针数组访问arr1、arr2、arr3三个数组的元素。

第19行的*(ptrArray[i]+j)是这样求值的：首先通过ptrArray[i]获得数组的首地址，然后ptrArray[i]+j得到当前数组第j个元素的地址，最后通过*(ptrArray[i]+j)进行间址运算，得到数组元素的值。

当然，利用指针数组也可以管理长度不同的数组，前提是指针的关联类型和数组的元素类型必须相同。也可以使用指向数组的指针数组管理数组。

例5.20 使用指向数组的指针数组管理数组。

```
1   //ex5-20.cpp:使用指向数组的指针数组管理数组
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       //声明一个长度为3的指向数组的指针数组,所指向的数组包含3个整型元素
7       int (*ptrArray[3])[3];
8       //创建3个整型数组,并初始化
9       int arr1[3]={ 1,2,3 };
10      int arr2[3]={ 4,5,6 };
11      int arr3[3]={ 7,8,9 };
```

```
    //将 3 个数组的地址赋值给指针数组的元素
    //ptrArray[0] = arr1;Error:不能将 int * 类型的值分配到 int( * )[3]类型的实体
    ptrArray[0] = &arr1;
    ptrArray[1] = &arr2;
    ptrArray[2] = &arr3;
    //通过指针数组访问并输出三个数组的元素
    for(int i = 0;i < 3;++i){
        cout <<"Array "<< i + 1 <<":" ;
        for(int j = 0;j < 3;++j){
            cout << * ( * ptrArray[i] + j)<<" ";
        }
        cout << endl;
    }
}
```

程序运行结果与例 5.19 的运行结果相同。

例 5.20 中，arr1、arr2、arr3 的各元素都是通过指向数组的指针数组 ptrArray 的元素来访问的。 * ptrArray[i]表示某数组的起始地址，* (* ptrArray[i]+j)表示访问该数组的第 j 个元素。

5.4.2 二维数组

1. 二维数组的定义

二维数组的说明格式为

```
类型 数组名[常量表达式 1][常量表达式 2];
```

其中，“常量表达式 1”指定二维数组第一维的长度，“常量表达式 2”指定二维数组第二维的长度。二维数组表示矩阵，第一维表示行数，第二维表示列数。

例如：

```
int arr1[3][4];                    //3 行 4 列的 int 类型数组
double arr2[5][5];                 //5 行 5 列的 double 类型数组
char arr3[20][20];                 //20 行 20 列的 char 类型数组
```

二维数组可以看成一个一维数组，该一维数组的每一个元素又是一个一维数组。对于数组 arr1，可以看成一个有 3 个元素的一维数组：arr[0]、arr[1]、arr[2]，每个元素都是长度为 4 的一维整型数组，如图 5.2(a)所示。

二维数组中元素是按照“行主序”的方式存储的，即先存储第一行的元素，然后是第二行的元素，以此类推。如图 5.2(b)所示，数组 a 的存放次序是：a[0][0]，a[0][1]，a[0][2]，a[0][3]，a[1][0]，a[1][1]，…，a[2][2]，a[2][3]。

2. 二维数组的初始化

和一维数组一样，可以使用由花括号括起的初始化列表来初始化二维数组的元素。例如：

```
int arr1[3][4] = {
    {0,1,2,3},                     //初始化第 0 行
    {4,5,6,7},                     //初始化第 1 行
    {8,9,10,11}                    //初始化第 2 行
};
```

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

(a) 二维数组

FB98	a[0][0]	FB98	FB99	FB9A	FB9B
FB9C	a[0][1]	FB9C	FB9D	FB9E	FB9F
FBA0	a[0][2]	FBA0	FBA1	FBA2	FBA3
FBA4	a[0][3]	FBA4	FBA5	FBA6	FBA7
FBA8	a[1][0]	FBA8	FBA9	FBAA	FBAB
FBAC	a[1][1]	FBAC	FBAD	FBAE	FBAF
FBB0	a[1][2]	FBB0	FBB1	FBB2	FBB3
FBB4	a[1][3]	FBB4	FBB5	FBB6	FBB7
FBB8	a[2][0]	FBB8	FBB9	FBBA	FBBB
FBBC	a[2][1]	FBBC	FBBD	FBBE	FBBF
FBB0	a[2][2]	FBC0	FBC1	FBC2	FBC3
FBC4	a[2][3]	FBC4	FBC5	FBC6	FBC7

(b) 二维数组的存储

图 5.2 二维数组的存储方式

可以省略用来标志每一行的花括号，例如：

```
int arr1[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};      //初始化 arr1
```

可以只对部分元素初始化，例如：

```
int arr1[3][4] = {{0},{1},{2}};
```

该语句只初始化了数组 arr1 的每行的第一个元素，其余元素被初始化为 0。

```
int arr1[3][4] = {1,2,3,4};
```

该语句初始化了数组 arr1 第一行的元素，其余元素被初始化为 0。

3. 以下标方式访问二维数组

二维数组元素带有两个下标表达式，具体格式如下：

```
数组名[表达式 1][表达式 2]
```

其中，“表达式 1”说明了元素所在的行，“表达式 2”说明了元素所在的列。

例 5.21 使用下标访问二维数组。

```
1   //ex5 - 21.cpp:使用下标访问二维数组
2   # include < iostream >
3   using namespace std;
4   int main()
5   {
6     int arr[3][4] = {                                  //定义一个 3 行 4 列的二维数组
7         {1,2,3,4},
8         {5,6,7,8},
9         {9,10,11,12}
```

```
    };
    for(int i = 0;i < 3;i++){                              //遍历行
        for(int j = 0;j < 4;j++){                          //遍历列
            cout << arr[i][j]<<" ";                        //访问并输出二维数组的所有元素
        }
        cout << endl;                                      //每打印完一行后换行
    }
    cout <<"Accessing an individual element:"<< endl;
    cout <<"The value of arr[1][2] is:"<< arr[1][2]<< endl; //访问输出第二行第三列的元素
    arr[1][2] = 99;                                        //修改第二行第三列的元素为 99
    cout <<"The array after modification:"<< endl;         //再次输出数组以查看修改后的结果
    for(int i = 0;i < 3;i++){
        for(int j = 0;j < 4;j++){
            cout << arr[i][j]<<" ";
        }
        cout << endl;
    }
}
```

程序运行结果：

```
1 2 3 4
5 6 7 8
9 10 11 12
Accessing an individual element:
The value of arr[1][2] is: 7
The array after modification:
1 2 3 4
5 6 99 8
9 10 11 12
```

4. 以指针方式访问二维数组

以指针方式访问二维数组，可以从一维数组的结构推导出来。从图 5.3 可以看出，二维数组 a 是由元素 a[0]、a[1]、a[2]组成的一维数组，所以，a 是 a[0]、a[1]、a[2]的首地址(指针)，有

a==&a[0]　　a+1==&a[1]　　a+2==&a[2]

*a==a[0]　　*(a+1)==a[1]　　*(a+2)==a[2]

元素 a[0]是一维数组，有 4 个整型元素，分别为 a[0][0]、a[0][1]、a[0][2]和 a[0][3]。a[0]是这个一维数组的指针，有

a[0]==&a[0][0]　　a[0]+1==&a[0][1]　　a[0]+2==&a[0][2]　　a[0]+3==&a[0][3]

*a[0]==a[0][0]　　*(a[0]+1)==a[0][1]　　*(a[0]+2)==a[0][2]　　*(a[0]+3)==a[0][3]

同理，a[1]、a[2]具有类似的特点。

a[1]==&a[1][0]　　a[1]+1==&a[1][1]　　a[1]+2==&a[1][2]　　a[1]+3==&a[1][3]

*a[1]==a[1][0]　　*(a[1]+1)==a[1][1]　　*(a[1]+2)==a[1][2]　　*(a[1]+3)==a[1][3]

a[2]==&a[2][0]　　a[2]+1==&a[2][1]　　a[2]+2==&a[2][2]　　a[2]+3==&a[2][3]

*a[2]==a[2][0]　　*(a[2]+1)==a[2][1]　　*(a[2]+2)==a[2][2]
*(a[2]+3)==a[2][3]

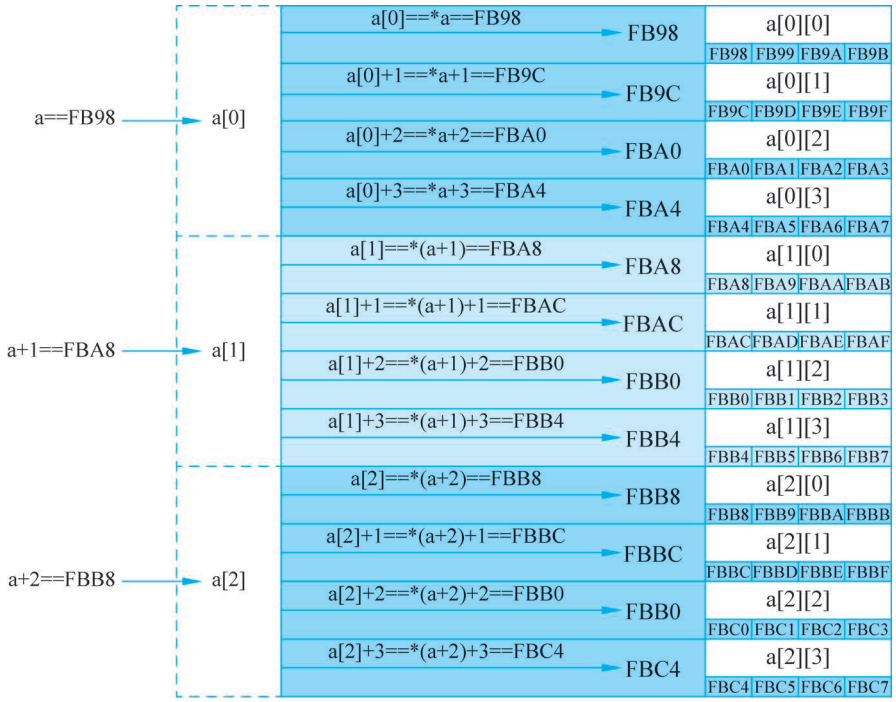

图 5.3　二维数组的指针表示

例 5.22　使用指针访问二维数组。

```
1   //ex5-22.cpp:使用指针访问二维数组
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int arr[3][4]={
7           {1,2,3,4},
8           {5,6,7,8},
9           {9,10,11,12}
10      };
11      int rows=3;
12      int cols=4;
13      cout<<"Accessing elements using pointers:"<<endl;   //使用指针访问输出所有元素
14      for(int i=0;i<rows;i++){
15          for(int j=0;j<cols;j++){
16              cout<<*(arr[i]+j)<<" ";                     //访问元素 arr[i][j]
17          }
18          cout<<endl;
19      }
20      cout<<"Accessing elements using pointers (another way):"<<endl;
                                                            //使用指针访问(另一种方式)
21      for(int i=0;i<rows;i++){
```

```
        for(int j = 0;j < cols;j++){
            cout << * ( * (arr + i) + j)<<" ";
        }
        cout << endl;
    }
}
```

程序运行结果：

```
Accessing elements using pointers:
1 2 3 4
5 6 7 8
9 10 11 12
Accessing elements using pointers (another way):
1 2 3 4
5 6 7 8
9 10 11 12
```

使用指针访问二维数组时，可能出现越界访问而导致程序崩溃或者行为未定义。如：

```
int arr[3][4];
int val = * ( * (arr + 3) + 4);                    //错误:访问了不存在的元素,即越界访问
```

因此，在使用指针访问二维数组时，要格外谨慎。

5.4.3 字符串

1. 字符串的存储

C++从 C 语言继承下来的一种通用结构称为 C 风格字符串(C-style character string)，它是以空字符('\0')结束的字符数组。另外，也可以用字符指针来管理字符串。

用字符数组存放字符串，有不同的初始化方式。

```
char str0[ ] = {'H','e','l','l','o'};//逐个字符对数组元素赋初始值,系统不会自动添加结束符'\0'
char str1[ ] = {'H','e','l','l','o','\0'};      //逐个字符对数组元素赋初始值
char str2[ ] = {"Hello"};                       //用串常量赋初值,自动添加结束标志'\0'
char str3[ ] = "Hello";                         //可省略{},直接用串常量赋初值
```

采用逐个字符对数组元素进行初始化时，系统不会自动添加结束符'\0'，可能会导致不可预料的错误。

实际应用中，字符串的长度变化很大，将字符指针作为串地址，为管理字符串提供了方便：

```
char * pstr1 = str1;                            //字符指针指向字符串 str1 的第一个字符
char * pstr2 = str2;                            //字符指针指向字符串 str2 的第一个字符
const char * pstr3 = "Hello";                   //用字符串常量初始化字符指针
```

2. 字符串的访问

字符串可以作为一个整体进行输入输出。字符串中的各个元素可以通过下标或指针的间址方式进行访问。

例 5.23 字符串的访问。

```
1   //ex5 - 23.cpp:字符串的访问
2   #include < iostream >
3   using namespace std;
4   int main()
5   {
6       char str1[100];                         //定义一个字符串变量 str1,以字符数组表示字符串
7       cout <<"Enter a string without spaces:";
```

```
8       cin >> str1;                          //使用 cin 读取用户输入的字符串,不包括空格
9       cout <<"You entered:"<< str1 <<'\n'; //输出 str1
10      char str2[ ] = "Hello,C++world!";   //定义一个字符串变量 str2,以字符数组表示字符串
11      char * ptr;
12      for(ptr = str2; * ptr != '\0';ptr++) //访问字符串中的每个字符
13      {
14          cout << * ptr;
15      }
16      cout <<'\n';
17      int length = 0;
18      for(ptr = str2; * ptr != '\0';ptr++) //计算字符串长度
19      {
20          length++;
21      }
22      cout <<"String length is:"<< length << endl;
23      char firstChar = str2[0];             //访问特定位置的字符:第一个字符
24      char lastChar = str2[length - 1];     //访问特定位置的字符:最后一个字符
25      cout <<"First character:"<< firstChar <<'\n';
26      cout <<"Last character:"<< lastChar <<'\n';
27      str2[7] = 'c';                        //修改字符串内容,将'C++'改为'c++'
28      cout <<"Modified string:"<< str2 <<'\n';
29  }
```

程序运行结果：

```
Enter a string without spaces: Hello!
You entered: Hello!
Hello, C++ world!
String length is: 17
First character: H
Last character: !
Modified string: Hello, c++ world!
```

例 5.23 演示了如何访问字符串。第 6～9 行实现了字符串的整体输入输出,使用 cin 和 cout 分别读取用户输入的字符串(不包括空格)和输出字符串；第 10～16 行使用字符指针遍历字符串并打印每个字符；第 17～22 行演示了如何计算字符串的长度；第 23～26 行访问字符串的第一个和最后一个字符；第 27、28 行演示了修改字符串中的特定字符。

3. 字符串的赋值

可以像数值型数组一样,用循环语句将一个字符数组的各个元素赋值给另一个字符数组对应元素,从而实现字符串的赋值。

例 5.24 字符串的赋值。

```
1   //ex5 - 24.cpp:字符串的赋值
2   #include < iostream >
3   #include < cstring >
4   using namespace std;
5   int main()
6   {
7   char str1[ ] = "Hello";
8   int len = 0;
9   while (str1[len] != '\0')              //计算 str1 的长度
10  {
11    len++;
12  }
```

```
13  char str2[256];                          //创建一个足够大的数组 str2 来存储 str1 的副本
14  for(int i = 0;i < len;++i)               //手动赋值
15  {
16      str2[i] = str1[i];
17  }
18  str2[len] = '\0';                        //在 str2 的末尾添加结束符
19  cout <<"str2:"<< str2 << endl;
20  }
```

程序运行结果：

```
str2: Hello
```

4. C 风格字符串的标准库函数

表 5.1 列出了 C 语言标准库提供的一系列处理 C 风格字符串的库函数，表中的 errno_t 是能够存储错误代码的无符号整数类型，size_t 是 C 和 C++标准库中定义的一个无符号整数类型，用于表示对象的大小；rsize_t 是 size_t 的别名。要使用这些标准库函数，必须包含相应的 C 头文件：#include <cstring>，这里，cstring 是 string.h 头文件的 C++版本。

表 5.1　处理字符串的库函数

函 数 原 型	功　　能
errno_t strcpy_s(char * dest,rsize_t destsz,const char * src);	将 src 指向的字符串安全地复制到 dest 指向的位置，destsz 是 dest 的大小
errno_t strncpy_s(char * dest,rsize_t destsz,const char * src,rsize_t count);	将 src 指向的字符串的前 count 个字符安全地复制到 dest 指向的位置，destsz 是 dest 的大小
errno_t strcat_s(char * dest,rsize_t destsz,const char * src);	将 src 指向的字符串安全地连接到 dest 指向的字符串的末尾，destsz 是 dest 的大小
errno_t strncat_s(char * dest,rsize_t destsz,const char * src,rsize_t count);	将 src 指向的字符串的前 count 个字符安全地连接到 dest 指向的字符串的末尾，destsz 是 dest 的大小
int strcmp(const char * s1,const char * s2);	比较两个字符串，如果 s1 等于 s2 则返回 0，若 s1 小于 s2 则返回负数，若 s1 大于 s2 则返回正数
int strncmp(const char * s1,const char * s2,size_t n);	比较两个字符串的前 n 个字符
size_t strlen(const char * s);	返回字符串的长度，不包括结尾的空字符

例 5.25　使用 C 风格字符串的标准库函数。

```
1   //ex5 - 25.cpp:使用 C 风格字符串的标准库函数
2   #include <iostream>
3   #include <cstring>                          //包含 C 风格字符串操作函数
4   using namespace std;
5   int main()
6   {
7       char str1[50] = "Hello";
8       char str2[50] = "C++World";
9       char str3[100];
10      char str4[50];
11      int result;
12      result = strcpy_s(str3,sizeof(str3),str1);    //使用 strcpy_s()函数复制字符串
13      if(result == 0){
14          cout <<"After strcpy_s:"<< str3 << endl;
15      }
```

```
16      else{
17          cout <<"Error occurred during strcpy_s."<< endl;
18      }
19      result = strncpy_s(str4,sizeof(str4),str2,3); //使用 strncpy_s()函数安全复制字符串
20      if(result == 0){
21          cout <<"After strncpy_s:"<< str4 << endl;
22      }
23      else{
24          cout <<"Error occurred during strncpy_s."<< endl;
25      }
26      result = strcat_s(str3,sizeof(str3)," ");      //在 str3 的末尾添加一个空格
27      if(result == 0){
28          cout <<"After strcat_s (adding space):"<< str3 << endl;
29      }
30      else{
31          cout <<"Error occurred during strcat_s (adding space)."<< endl;
32      }
33      result = strcat_s(str3,sizeof(str3),str2);     //使用 strcat_s()函数连接字符串
34      if(result == 0){
35          cout <<"After strcat_s:"<< str3 << endl;
36      }
37      else{
38          cout <<"Error occurred during strcat_s."<< endl;
39      }
40      result = strcmp(str1,str3);                    //使用 strcmp()函数比较字符串
41      if(result == 0){
42          cout <<"strcmp:str1 and str3 are equal."<< endl;
43      }
44      else{
45          cout <<"strcmp:str1 and str3 are not equal."<< endl;
46      }
47  }
```

程序运行结果：

```
After strcpy_s: Hello
After strncpy_s: C++
After strcat_s (adding space): Hello
After strcat_s: Hello C++ World
strcmp: str1 and str3 are not equal.
```

第 12 行使用 strcpy_s()函数将 str1 的内容复制到 str3 中。第 19 行使用 strncpy_s()函数从 str2 复制最多 3 个字符到 str4 中。strncpy_s()函数允许指定复制的最大字符数。第 26 行使用 strcat_s()函数在 str3 的末尾添加一个空格。第 33 行使用 strcat_s()函数将 str2 的内容连接到 str3 的末尾。第 40～46 行使用 strcmp()函数比较 str1 和 str3 的内容，如果 strcmp()函数返回 0，则表示两个字符串相等，如果 strcmp()函数返回非 0 值，则表示两个字符串不相等。

注意，strcpy_s()、strncpy_s()、strcat_s()和 strncat_s()函数需要检查目标缓冲区的大小，防止缓冲区溢出，是比 strcpy()、strncpy()、strcat()和 strncat()函数更安全的版本。但是这几个函数是 C++11 标准的一部分，不是所有的编译器都支持。另外，在不同的编译器中可能有不同的实现或名称，例如在某些编译器中，它们可能被命名为 strcpy_s、strcat_s 和 strncpy_s，而在其他编译器中可能需要包含特定的头文件或使用不同的命名空间。在使用

这些函数之前，请确保查阅相关编译器文档以获取正确的用法。

5. C风格字符串的注意事项

（1）内存管理。当使用字符数组时，需要确保数组的大小足够容纳字符串及其终止符（'\0'）。当使用字符指针时，要特别注意指针指向的内存是否有效。

（2）边界检查。许多C风格字符串操作函数不进行边界检查，这可能导致缓冲区溢出和安全漏洞。因此，使用这些函数时需要格外小心，或者使用更安全的替代函数。

5.4.4 数组与函数

1. 数组作为函数参数

当数组名作为函数参数时，C++进行传址处理。调用函数时，形参数组名接收实参数组的地址，函数通过形参指针对实参数组元素间址访问。

例 5.26 假设使用一个数组来记录几名学生的成绩，现在欲计算学生的平均分。请使用数组名作为函数参数编程实现。

```
1   //ex5-26.cpp:用数组名作为函数参数计算平均成绩
2   #include<iostream>
3   using namespace std;
4   double calculateAverage(double scores[],int size);        //函数原型声明,计算平均值
5   int main()
6   {
7       //学生成绩数组
8       double C1_scores[]={ 85.5,90.0,78.2,92.5,88.0 };
9       int size=sizeof(C1_scores)/sizeof(C1_scores[0]);   //计算数组中元素的数量
10      double average=calculateAverage(C1_scores,size);   //计算平均值
11      cout<<"The average C1_score is:"<<average<<endl;   //输出平均值
12  }
13  //函数定义,计算平均值
14  double calculateAverage(double scores[],int size)
15  {
16      double sum=0.0;                                    //用于累加成绩的变量
17      for(int i=0;i<size;++i){
18          sum += scores[i];                              //累加数组中的成绩
19      }
20      return sum/size;                                   //计算平均值并返回
21  }
```

程序运行结果：

```
The average C1_score is: 86.84
```

例5.26中，calculateAverage()函数接收一个double类型的数组scores和一个int类型的size参数，分别代表成绩数组和数组的大小。在main()函数中，定义了一个包含学生成绩的数组C1_scores，并计算了数组的大小。然后，调用calculateAverage()函数，并传入成绩数组和数组大小，计算平均值。

第10行中，C1_scores是数组名，而根据C++的规则，C1_scores表示数组第一个元素的地址，因此函数传递的是地址。由于数组的类型是double，因此，接收C1_scores的形参类型必须是double指针，即double*，所以，函数原型声明也可以是这样：

```
double calculateAverage(double *scores,int size);
```

当(且仅当)用于函数头或者函数原型中,double * scores 和 double scores[]的含义才相同。它们都意味着 scores 是一个指针。因此,上述程序中 calculateAverage()函数也可以用如下代码实现:

```
1   double calculateAverage(double scores*,int size)        //函数头
2   {
3       double sum=0.0;
4       for(int i=0;i<size;++i){
5           sum += *scores;
6           scores++;
7       }
8       return sum/size;
9   }
```

第 6 行的 scores++表示将指针指向数组的下一个元素。

例 5.27 数组作为函数参数,观察形参和实参的大小。

```
1   //ex5-27.cpp:数组作为函数参数
2   #include<iostream>
3   using namespace std;
4   //函数用于输出形参的大小
5   void test(int ap[],double bp[],char cp[]){
6       cout<<"sizeof(ap)="<<sizeof(ap)<<endl;          //输出形参 ap 的大小
7       cout<<"sizeof(bp)="<<sizeof(bp)<<endl;          //输出形参 bp 的大小
8       cout<<"sizeof(cp)="<<sizeof(cp)<<endl;          //输出形参 cp 的大小
9   }
10  int main(){
11      int intArray[10]={1,2,3,4,5,6,7,8,9,10};        //整型数组
12      double doubleArray[10]={0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0}; //双精度浮
                                                                          //点型数组
13      char charArray[10]={'a','b','c','d','e','f','g','h','i','j'};     //字符型数组
14      //输出实参的大小
15      cout<<"sizeof(intArray)="<<sizeof(intArray)<<endl;  //输出实参 intArray 的大小
16      cout<<"sizeof(doubleArray)="<<sizeof(doubleArray)<<endl;  //输出实参 doubleArray
                                                                  //的大小
17      cout<<"sizeof(charArray)="<<sizeof(charArray)<<endl;//输出实参 charArray 的大小
18      //调用 test()函数
19      test(intArray,doubleArray,charArray);
20  }
```

程序运行结果:

```
sizeof(intArray)=40
sizeof(doubleArray)=80
sizeof(charArray)=10
sizeof(ap)=4
sizeof(bp)=4
sizeof(cp)=4
```

例 5.27 中,第 15～17 行输出 intArray、doubleArray 和 charArray 占用的内存大小。对于 intArray,每个整型数通常占用 4 字节(这取决于编译器和机器架构,但 4 字节是最常见的),所以 sizeof(intArray)会输出 40。对于 doubleArray,每个双精度浮点型数通常占用 8 字节,所以 sizeof(doubleArray)会输出 80。对于 charArray,每个字符通常占用 1 字节,所

以 sizeof(charArray)会输出 10。

在 test()函数中，sizeof(ap)、sizeof(bp)和 sizeof(cp)会输出相同的值。这是因为在 C++中，数组作为函数参数时会被退化为指针，所以 sizeof 操作符实际上返回的是指针的大小(即一个内存地址的字节数)，而不是数组的大小。内存地址的字节数取决于机器架构，通常是 4 字节(32 位系统)或 8 字节(64 位系统)。

由上面几个示例程序可见，在数组作为函数参数的示例中，并没有将实参数组内容传递给函数的形参，而是将实参数组的位置(起始地址)、包含的元素种类(类型)以及元素数目(size 变量)传递给函数，有了这些信息，函数便可以访问实参数组。将数组地址作为实参可以节省复制整个数组所需的时间和内存。

但是，在函数中直接访问实参增加了破坏实参的风险，因此，可以使用 const 来约束形参，起到保护实参的目的。

例 5.28 对数组中学生的成绩进行修改，并输出学生成绩。

```
1   //ex5-28.cpp:对数组中学生的成绩进行修改,并输出学生成绩
2   #include<iostream>
3   using namespace std;
4   void printScores(const double scores[],int size);          //函数声明,打印成绩
5   void scaleScores(double scores[],int size,double factor);  //函数声明,修改学生成绩并输出
6   int main(){
7       double C1_scores[]={ 85.5,90.0,78.2,92.5,88.0 };       //学生成绩数组
8       int size=sizeof(C1_scores)/sizeof(C1_scores[0]);       //计算数组中元素的数量
9       cout<<"Original scores:";                              //输出原始成绩
10      printScores(C1_scores,size);
11      scaleScores(C1_scores,size,1.05);                  //修改成绩,每个成绩乘以 1.05 的系数
12      cout<<"Scaled scores:";                                //输出修改后的成绩
13      printScores(C1_scores,size);
14  }
15  //函数定义,打印成绩
16  void printScores(const double scores[],int size){
17      for(int i=0;i<size;++i){
18          cout<<scores[i]<<" ";
19      }
20      cout<<endl;
21  }
22  //函数定义,修改学生成绩
23  void scaleScores(double scores[],int size,double factor){
24      for(int i=0;i<size;++i){
25          scores[i] *= factor;                               //将每个成绩乘以系数
26      }
27  }
```

程序运行结果：

```
Original scores: 85.5 90 78.2 92.5 88
Scaled scores: 89.775 94.5 82.11 97.125 92.4
```

例 5.28 中，第 16～21 行定义的 printScores()函数的第一个参数使用 const 关键字约束，意味着在函数体内不能修改数组的内容。

2. 函数与二维数组

将二维数组作为参数函数，数组名仍被视为地址，相应的形参是一个指针。例如：

```
int myArray[3][4] = {{1,2,3,4},{6,7,8,9},{2,4,6,7}};
int total = sum(myArray,3);
```

Myarray 是一个二维数组，可以看成具有 3 个元素的一维数组，每个元素又是由 4 个整型数据组成的一维数组，因此，Myarray 是指向具有 4 个整型数据的一维数组的常指针，由此可以确定 sum()函数原型：

```
int sum(int( * arr)[4],int size);
```

或

```
int sum(int arr[][4],int size);
```

以上两种原型的第一个参数都表示，arr 是指针变量，它指向由 4 个整型数据组成的一维数组，调用函数时，将实参数组 Myarray 传递过来即可。

例 5.29 二维数组作为函数参数。

```
1   //ex5 - 29.cpp:二维数组作为函数参数
2   #include < iostream >
3   using namespace std;
4   //函数用于计算二维数组中所有元素的和
5   int sumOfArray(int arr[][10],int rows)
6   {
7       int sum = 0;
8       for(int i = 0;i < rows;++i){
9           for(int j = 0;j < 10;++j){
10              sum  += arr[i][j];
11          }
12      }
13      return sum;
14  }
15  //函数用于打印二维数组的所有元素
16  void printArray(const int arr[][10],int rows)
17  {
18      for(int i = 0;i < rows;++i){
19          for(int j = 0;j < 10;++j){
20              cout << arr[i][j]<<" ";
21          }
22          cout << endl;
23      }
24  }
25  int main()
26  {
27          int myArray[3][10] = {
28          {1,2,3,4,5,6,7,8,9,10},
29          {11,12,13,14,15,16,17,18,19,20},
30          {21,22,23,24,25,26,27,28,29,30}
31      };
32      int totalSum = sumOfArray(myArray,3);                    //计算二维数组中所有元素的和
33      cout <<"The sum of all elements in the array is:"<< totalSum << endl;
34      cout <<"The elements of the array are:"<< endl;
35      printArray(myArray,3);                                   //打印二维数组的所有元素
```

```
}
```

程序运行结果：

```
The sum of all elements in the array is: 465
The elements of the array are:
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

例 5.29 中，sumOfArray()函数和 printArray()函数都有一个二维数组和一个整型变量作为参数，其中 printArray()函数中的第一个参数被声明为 const，意味着在函数内部不能修改数组的内容，达到保护实参的目的。

函数形参接收高维数组的地址后，也可以用不同的方式进行处理，其关键在于不同维数的数组名代表不同逻辑级别的指针，不同逻辑级别的指针移动时具有不同的偏移量。

例 5.30 数组的降维处理。

```
//ex5-30.cpp:数组的降维处理
#include<iostream>
using namespace std;
//函数用于求第 i 行到第 j 行元素的和,将二维数组降维为一维数组来计算
int sumRows(int* oneDArray,int cols,int i,int j)
{
    //检查索引是否在有效范围内
    if(i<0||j>=cols||i>j){
        cout<<"Invalid row indices."<<endl;
        return 0;                                    //返回 0
    }
    int sum=0;
    for(int row=i;row<= j;++row){
        for(int col=0;col<cols;++col){
            sum += oneDArray[row*cols+col];
        }
    }
    return sum;
}
int main()
{
    const int Rows=3;
    const int Cols=4;
    int twoDArray[Rows][Cols]={
        {1,2,3,4},
        {5,6,7,8},
        {9,10,11,12}
    };
    int i=0;                                         //起始行
    int j=2;                                         //结束行
    int result=sumRows(&twoDArray[0][0],Cols,i,j);   //求第 i 行到第 j 行元素的和
    cout<<"The sum from row "<<i<<" to row "<<j<<" is:"<<result<<endl;
}
```

程序运行结果：

```
The sum from row 0 to row 2 is: 78
```

例 5.30 中，sumRows()函数接收一个指向整型数据的指针 oneDArray、列数 cols，以及起始行索引 i 和结束行索引 j，函数通过遍历指定区间的数组元素，将每个元素累加到变量 sum 中。main()函数定义了一个二维数组 twoDArray，将数组第一个元素的地址作为实参传递给形参 oneDArray，调用 sumRows()函数计算指定区间的和，并将结果存储在变量 result 中。

3. 指向函数的指针数组

函数可以用名调用，也可以用指针调用。因此，同类型的函数就可以用指针数组管理。

例 5.31 指向函数的指针数组。

```
1   //ex5-31.cpp:指向函数的指针数组
2   #include<iostream>
3   using namespace std;
4   const double PI=3.1415;                                    //使用 const 定义常量 PI
5   double Square_Girth(double l){return 4*l;}                 //计算正方形的周长
6   double Square_Area(double l){return l*l;}                  //计算正方形的面积
7   double Round_Girth(double r){return 2*PI*r;}               //计算圆的周长
8   double Round_Area(double r){return PI*r*r;}                //计算圆的面积
9   typedef double (*ft)(double);                              //定义函数类型
10  int main()
11  {
12      double x=1.23;
13      ft pfun[4]={Square_Girth,Square_Area,Round_Girth,Round_Area};   //函数指针数组
14      for(int i=0;i<4;i++){
15          if(pfun[i]){                                       //检查函数指针是否为 nullptr
16              cout<<(*pfun[i])(x)<<endl;
17          }
18          else{
19              cout<<"Function pointer is null."<<endl;
20          }
21      }
22  }
```

程序运行结果：

```
4.92
1.5129
7.72809
4.75278
```

第 9 行使用 typedef 定义了函数指针类型 ft，它指向的函数接收一个 double 类型参数并返回一个 double 类型的值。第 13 行定义了一个函数指针数组 pfun，元素类型为 ft 类型，并初始化为 Square_Girth、Square_Area、Round_Girth 和 Round_Area。第 14 行的 for 语句中，随着 i 值的变化，由 cout 语句调用了不同的函数，并输出返回结果。第 16 行的函数调用表达式(*pfun[i])(x)可以写成(pfun[i])(x)。

5.5 动态存储

很多时候，程序设计无法提前预知需要多少内存来存储需要处理的信息，所需内存的大小要在运行时才能确定。在 C++中，可以使用 new 运算符为给定类型的对象动态分配堆

内的内存。由 new 运算符分配的内存，可以使用 delete 运算符释放。

new 与 delete 的一般语法形式为

```
指针变量 = new 类型
delete 指针变量
```

在这里，new 按照指定类型的长度分配存储空间，并返回所分配空间的首地址。“类型”可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。

例如：

```
int * pvalue1 = new int(10);              //动态分配一个 int 类型存储空间，并初始化为 10
char * pvalue2 = new char;                //动态分配一个 char 类型存储空间
double * pvalue3 = new double;            //动态分配一个 double 类型存储空间
//…
delete pvalue1;                           //释放 pvalue1 所指向的存储空间
delete pvalue2;                           //释放 pvalue2 所指向的存储空间
delete pvalue3;                           //释放 pvalue3 所指向的存储空间
```

如果需要申请动态数组，则使用数组类型：

```
int * pvalue4 = new int[4];               //分配长度为 4 的 int 类型数组存储空间
//…
delete[ ]pvalue4;                         //释放 pvalue4 所指向的存储空间
```

例 5.32 使用指针动态申请内存。

```
1   //ex5 - 32.cpp:使用指针动态申请内存
2   #include < iostream >
3   using namespace std;
4   int main()
5   {
6       int * ptr = new int(10);          //动态分配一个整型对象的内存，并初始化为 10
7       cout << * ptr << endl;
8       * ptr = 20;                       //修改内存中的值
9       cout << * ptr << endl;
10      int * arr = new int[5];           //动态分配具有 5 个元素的 int 类型数组内存，数组的
                                          //起始地址由 arr 指向
11      for(int i = 0;i < 5;++i){         //访问动态数组
12          arr[i] = i * i;
13      }
14      for(int i = 0;i < 5;++i){         //输出数组元素
15          cout << arr[i]<<" ";
16      }
17      cout << endl;
18      delete ptr;                       //释放动态整型对象的内存
19      delete[ ] arr;                    //释放动态数组的内存
20  }
```

程序运行结果：

```
10
20
0 1 4 9 16
```

第 6 行和第 10 行分别使用 new 来分配一个 int 类型对象和一个数组对象的内存，第 18 行和第 19 行分别使用 delete 来释放它们。使用 new 分配的内存应使用 delete 来释放，如果忘记释放内存，则可能会导致内存泄漏。

在 C++ 中，创建动态二维数组可以通过使用指针的指针（即指向指针的指针）和 new 操作符来实现。

例 5.33 使用指针的指针创建动态二维数组。

```
1   //ex5-33.cpp:使用指针的指针创建动态二维数组
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int rows,cols;
7       cout<<"Enter the number of rows and columns:";
8       cin>>rows>>cols;
9       int** arr=new int*[rows];    //分配指针数组的内存,每个元素指向二维数组的一行
10      for(int i=0;i<rows;++i){
11          arr[i]=new int[cols];    //分配每行的内存,行的起始地址由指针数组元素保存
12      }
13      for(int i=0;i<rows;++i){     //访问内存,给动态数组各元素赋值
14          for(int j=0;j<cols;++j){
15              arr[i][j]=i*cols+j; //赋值
16          }
17      }
18      for(int i=0;i<rows;++i){     //访问内存,输出动态数组各元素的值
19          for(int j=0;j<cols;++j){
20              cout<<arr[i][j]<<" ";
21          }
22          cout<<endl;
23      }
24      for(int i=0;i<rows;++i){     //释放二维动态数组的内存
25          delete[] arr[i];         //释放每行的内存
26      }
27      delete[] arr;                //释放指向行的指针数组的内存
28  }
```

程序运行结果：

```
Enter the number of rows and columns: 3 5
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
```

使用指针的指针和 new 操作符来动态分配二维数组的内存，需要程序员自己负责内存的分配和释放，如果忘记释放内存，则可能会导致内存泄漏。

例 5.34 动态申请字符串的存储空间。

```
1   //ex5-34.cpp:动态存储示例
2   #include<iostream>
3   #include<cstring>
4   using namespace std;
```

```
5   int main()
6   {
7       const char * source = "Hello,China!";
8       char * str;
9       str = new char[strlen(source) + 1];                    //动态分配内存
10      if(str == nullptr){
11          cerr <<"Memory allocation failed.\n";
12          return 1;
13      }
14      strcpy_s(str,strlen(source) + 1,source);               //复制字符串
15      cout <<"Copied string:"<< str << endl;                 //使用字符串
16      delete[] str;                                          //释放内存
17  }
```

程序运行结果：

```
Copied string: Hello, China!
```

第 9 行使用 new 分配内存后，第 10 行检查返回的指针是否为 nullptr，以检测内存分配是否成功。第 16 行使用 delete[]来释放动态数组的内存，以防内存泄漏。

5.6 应用举例

本节通过应用示例对前述的知识进行综合运用。

例 5.35 选择排序法。

排序是计算机程序设计中的一种重要算法，按关键字从小到大排序称为升序排序，反之则称为降序排序。排序后的数据给处理带来极大的方便，可以提高数据处理的效率。

排序的方法有很多，本例介绍的选择排序法思路如下：

首先将 n 个待排序的整数放在数组 a[0]，a[1]，a[2]，…，a[n－1]中，然后进行以下步骤：第一趟，在 a[0]～a[n－1]中找出一个最小元素，设它是 a[min_idx]，则把 a[min_idx]与 a[0]交换，使得 a[0]最小；第二趟，在 a[1]～a[n－1]中找到最小元素 a[min_idx]，把它与 a[1]交换；其余类推，直到在 a[n－1]和 a[n]之中找到最小值。选择排序的算法可以描述为

```
for(i = 0;i < n - 1;i++)
{   从 a[i]到 a[n - 1]找到最小元素 a[min_idx]
    把 a[min_idx]与 a[i]交换
}
```

下面细化寻找最小元素算法。在每趟寻找中，设一个变量 min_idx，记录当前最小元素的下标：

```
for(j = i + 1;j < n;j++)
    if(a[j]< a[min_idx]) min_idx = j
```

对数组一趟搜索完成后，找到当前最小元素 a[min_idx]，然后将 a[i]和 a[min_idx]交换。对具有 5 个元素(47，44，5，38，3)的整数序列进行选择排序的过程如图 5.4 所示，i 的取值为 0、1、2、3，即需排序 4 轮，每轮都是找剩下元素中的最小值，并交换到合适的位置。

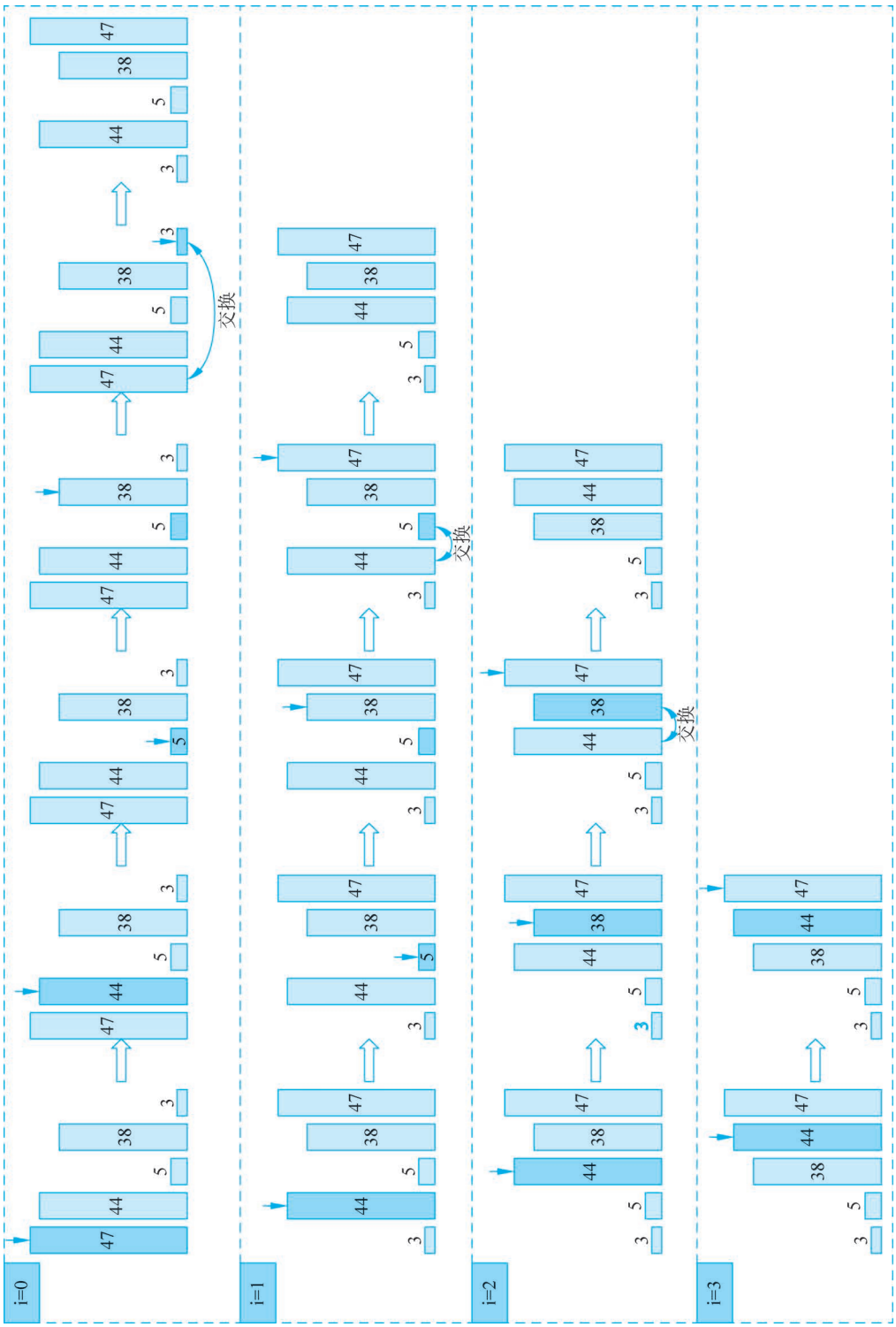

图 5.4 选择排序法排序过程

根据以上算法，可以写出如下程序。

```
//ex5-35.cpp:选择排序法
#include<iostream>
#include<cstdlib>                    //包含标准函数库,用于生成随机数
#include<ctime>                      //包含时间函数库,用于生成随机数种子
using namespace std;

void sort(int[],int);                //函数原型声明
int main(){
  int i,a[10];
  srand((int)(time(0)));             //设置随机数种子,使用当前时间作为实参调用 srand()函数
   for(i=0;i<10;i++){                //使用随机函数初始化数组 a
       a[i]=rand() % 100;
   }
   for(i=0;i<10;i++){                //输出原始序列
       cout<<a[i]<<"    ";
   }
   cout<<endl;
   sort(a,10);                       //调用排序函数,对数组 a 进行升序排序
   cout<<"Ordered:"<<endl;
   for(i=0;i<10;i++){                //输出排序后序列
       cout<<a[i]<<"    ";
   }
   cout<<endl;
   return 0;
}                                    //程序结束
   //函数定义,按照升序排序
void sort(int x[],int n){
   for(int i=0;i<n-1;i++){
       int min_idx=i;                //初始化最小元素索引为当前索引
       for(int j=i+1;j<n;j++){
           if(x[j]<x[min_idx]){      //找到更小的元素,更新最小元素索引
               min_idx=j;
           }
       }
       if(min_idx!= i){              //如果最小元素不在当前位置,则进行交换
           int temp=x[i];
           x[i]=x[min_idx];
           x[min_idx]=temp;
       }
   }
}
```

程序运行结果：

```
5   76   28   18   47   6   58   22   70   32
Ordered:
5   6   18   22   28   32   47   58   70   76
```

例 5.35 中用随机数对数组进行初始化。随机数经常用于产生测试、模拟数据。在实际应用中，有许多生成随机数的方法，线性同余法就是其中之一。用线性同余法生成随机数，随机数序列中的第 k 个数 r_k，可由它的前一个数 r_{k-1} 计算出来，计算公式如下：

$$r_k = (\text{multiplier} \times r_{k-1} + \text{increment}) \% \text{modulus}$$

例如，如果有

$$r_k = (25\,173 \times r_{k-1} + 13\,849) \% 65\,536$$

则可以产生 65 536 个互不相同的整型随机数。对这个公式稍作修改，还可以得到其他形式的随机数。但是，以上公式产生的序列并不是真正的随机数，而是伪随机数，因为给定 r_0 的值后，总能得到唯一的 r_k 的值。所以，把公式改为

$$r_k = (\text{multiplier} \times \text{number} + \text{increment}) \% \text{modulus}$$

其中，number 称为“种子”。这样，就可以产生接近真实的随机数了。

C++标准库< cstdlib >中提供两个用于产生随机数的函数：

rand()随机函数：返回 0～32 767 的随机值。该函数没有参数。

srand(number)种子函数：要求一个无符号整型参数设置随机数生成器的启动值。

为了使种子值变化敏感，通常用系统时间作为 srand()函数的参数。time()函数在< ctime >文件中定义。

第 9 行函数调用 time(0)表示用 0 作为实参，返回用整型数表示的系统当前时间。

例 5.36 冒泡排序法。

冒泡排序法也是常见的排序算法之一。冒泡排序法的排序过程就是对相邻元素进行比较调整。例如，对具有 n 个元素的数组 a 按升序排列的方法是，首先将 a[0]和 a[1]进行比较，如果为逆序(即 a[0]> a[1])，则 a[0]与 a[1]交换，然后比较 a[1]和 a[2]，以此类推，直到 a[n－2]和 a[n－1]进行过比较为止。这个过程称为一趟冒泡排序，其结果使得最大值放在最后一个位置 a[n－1]上。然后进行第二趟冒泡排序，对 a[0]～a[n－2]进行同样的操作，其结果使得次大值放在 a[n－2]的位置上，以此类推，直到所有的元素都排好序。整个过程就像烧开水一样，较小值像水中的气泡一样逐渐往上冒，每趟都有一个“最大”的值沉到水底。对具有 5 个元素(47，44，5，38 和 3)的整数序列进行冒泡排序法的过程如图 5.5 所示，i 的取值为 0、1、2、3，即需排序 4 趟，每趟都是对剩下没有排序好的元素进行两两比较，如果为“逆序”则进行交换。

根据以上分析，可以写出如下程序。

```
1   //ex5－36.cpp:冒泡排序法
2   #include < iostream >
3   #include < cstdlib >
4   #include < ctime >
5   using namespace std;
6   //冒泡排序函数定义
7   void bubbleSort(int * arr,int size){
8       bool wasSwapped;                    //局部变量,用于监控是否发生了交换
9       for(int i = 0;i < size－1;i++){
10          wasSwapped = false;             //每轮开始时重置 wasSwapped 为 false
11          for(int j = 0;j < size－i－1;j++){
12              if(arr[j]> arr[j＋1]){
13                  //使用临时变量交换两个元素的位置
14                  int temp = arr[j];
15                  arr[j] = arr[j＋1];
16                  arr[j＋1] = temp;
```

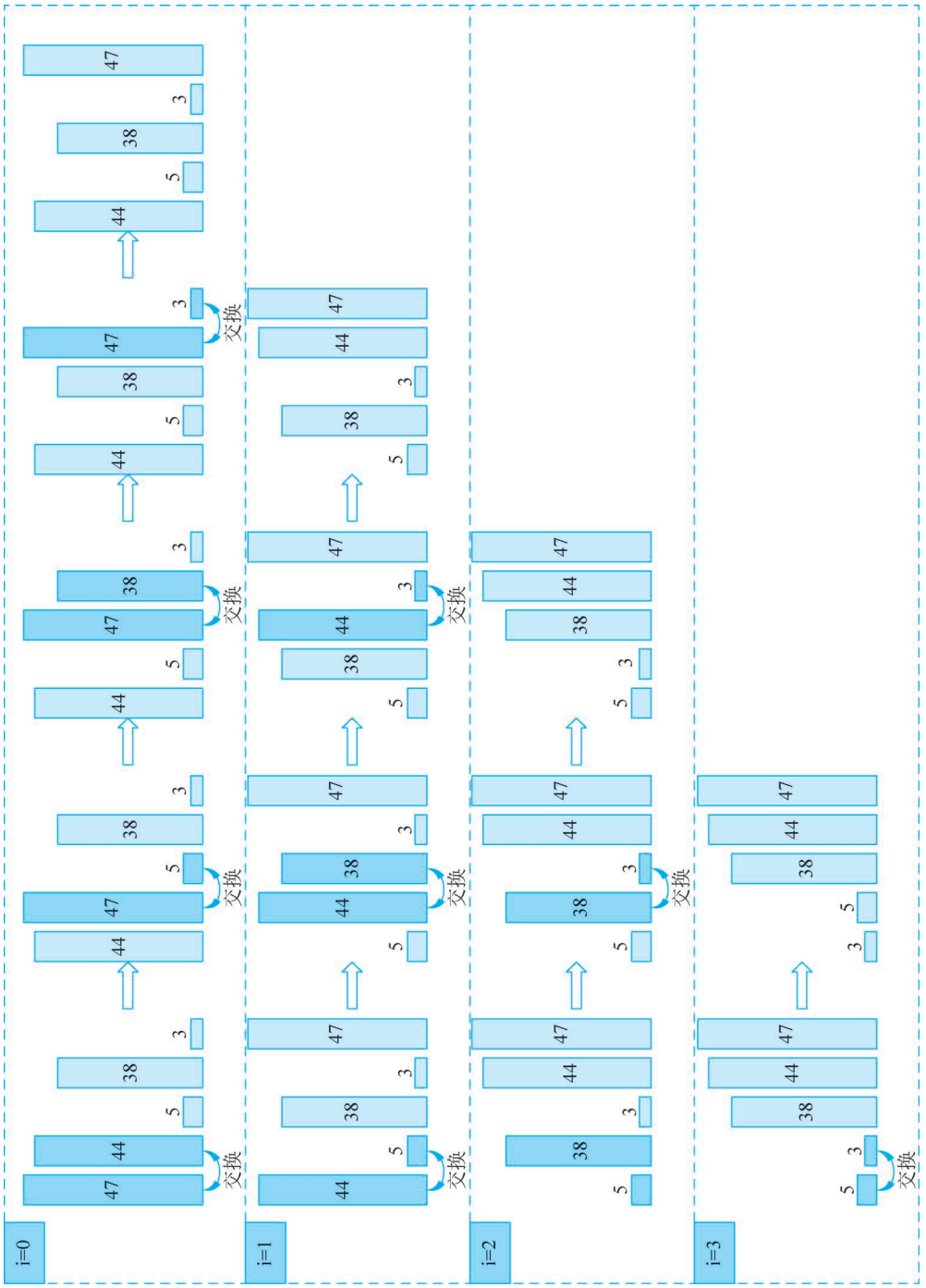

图 5.5　冒泡排序法排序过程

```
17                  wasSwapped = true;        //发生了交换,设置 wasSwapped 为 true
18              }
19          }
20          //如果在一轮遍历中没有发生交换,则数组已经排序完成,可以提前结束排序
21          if(!wasSwapped){
22              break;
23          }
24      }
25  }
26  int main(){
27      const int ARRAY_SIZE = 10;
28      int * arr = new int[ARRAY_SIZE];
29      srand((int)(time(nullptr)));          //使用当前时间作为随机数种子
30      for(int i = 0;i < ARRAY_SIZE;i++){
31          arr[i] = rand() % 100;            //生成 0 到 99 之间的随机数
32      }
33      cout <<"Original array:"<< endl;
34      for(int i = 0;i < ARRAY_SIZE;i++){
35          cout << arr[i]<<" ";
36      }
37      cout << endl;
38      bubbleSort(arr,ARRAY_SIZE);           //调用冒泡排序函数
39      cout <<"Sorted array:"<< endl;        //输出排序后的数组
40      for(int i = 0;i < ARRAY_SIZE;i++){
41          cout << arr[i]<<" ";
42      }
43      cout << endl;
44      delete[ ] arr;                        //释放动态数组的内存
45  }
```

程序运行结果：

```
Original array:
38 2 66 76 58 16 16 52 16 22
Sorted array:
2 16 16 16 22 38 52 58 66 76
```

在第 7～25 行定义的 bubbleSort()函数中，wasSwapped 是监控变量，用于监控在一轮遍历中是否发生了交换，每轮开始时重置 wasSwapped 为 false，如果在一轮遍历中没有发生交换，则数组已经排序完成，可以提前结束排序。

例 5.37 矩阵相乘。

求两矩阵的乘积 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$。设 $\boldsymbol{A}$、$\boldsymbol{B}$ 分别为 $m\times p$ 和 $p\times n$ 的矩阵，则 $\boldsymbol{C}$ 是 $m\times n$ 的矩阵。按矩阵乘法的定义，结果矩阵 $\boldsymbol{C}$ 的第 i 行和第 j 列的元素 c_{ij} 由下面公式定义：

$$C_{ij}=\sum_{k=1}^{p}A_{ik}\times B_{kj}\quad(i=1,2,\cdots,m;\ j=1,2,\cdots,n)$$

若有一个 4×3 的矩阵 $\boldsymbol{A}$ 乘以一个 3×2 的矩阵 $\boldsymbol{B}$，将得到一个 4×2 的矩阵 $\boldsymbol{C}$。在程序中，可以用二维数组表示矩阵。根据以上分析，可以写出如下程序。

```
1   //ex5 - 37.cpp:矩阵相乘
2   #include < iostream >
3   #include < iomanip >
4   using namespace std;
```

```
const int m = 4,p = 3,n = 2;
int a[m][p],b[p][n],c[m][n];
bool multimatrix(const int a[m][p],const int arow,const int acol,const int b[p][n],
  const int brow,const int bcol,int c[m][n],const int crow,const int ccol); //函数原型声明
int main(){
    int i,j;
    cout <<"Please input A:\n";
    for(i = 0;i < m;i++){
        for(j = 0;j < p;j++){
            cin >> a[i][j];
        }
    }
    cout <<"\nPlease input B:\n";
    for(i = 0;i < p;i++){
        for(j = 0;j < n;j++){
            cin >> b[i][j];
        }
    }
    //输出矩阵 A
    cout <<"\nMatrix A:\n";
    for(i = 0;i < m;i++){
        for(j = 0;j < p;j++){
            cout << setw(5)<< a[i][j];
        }
        cout << endl;
    }
    //输出矩阵 B
    cout <<"\nMatrix B:\n";
    for(i = 0;i < p;i++){
        for(j = 0;j < n;j++){
            cout << setw(5)<< b[i][j];
        }
        cout << endl;
    }
    if(!multimatrix(a,m,p,b,p,n,c,m,n)){          //函数调用
        cout <<"Illegal matrix multiply.\n";
        return 1;                                 //返回非 0 值表示错误
    }
    //输出结果矩阵 C
    cout <<"\nMatrix C (Result of A * B):\n";
    for(i = 0;i < m;i++){
        for(j = 0;j < n;j++){
            cout << setw(5)<< c[i][j];
        }
        cout << endl;
 }}
//函数定义
bool multimatrix(const int a[m][p],const int arow,const int acol,
    const int b[p][n],const int brow,const int bcol,int c[m][n],
    const int crow,const int ccol){
    if(acol != brow||crow != arow||ccol != bcol) return false;
```

```
56      for(int i = 0;i < crow;i++){
57          for(int j = 0;j < ccol;j++){
58              c[i][j] = 0;                          //初始化 c[i][j]为 0
59              for(int k = 0;k < acol;k++){
60                  c[i][j] += a[i][k] * b[k][j];
61              }
62          }
63      }
64      return true;
65  }
```

程序运行结果：

```
Please input A:
2 3 1 4 5 6 7 8 2 3 5 7

Please input B:
2 4 5 6 8 7

Matrix A:
    2    3    1
    4    5    6
    7    8    2
    3    5    7

Matrix B:
    2    4
    5    6
    8    7

Matrix C (Result of A * B):
   27   33
   81   88
   70   90
   87   91
```

例 5.37 中，第 51～65 行的 multimatrix()函数中矩阵乘法计算是通过三层嵌套循环完成的，第 56 行的外层循环遍历矩阵 C 的行，第 57 行的中间层循环遍历矩阵 C 的列，第 59 行的内层循环用于计算矩阵 C 中每个元素的值，它是矩阵 A 的一行与矩阵 B 的一列对应元素乘积的和。第 55 行判断两个矩阵是否满足相乘的条件，不满足时提前退出。

例 5.38 输出杨辉三角形。

二项式$(a+b)^n$展开式的项数由幂n决定，n次幂有$n+1$项，各项系数具有如下特点：第一项和最后一项系数等于 1，其余各项系数可以从$n-1$次幂系数表递推计算出来，其计算公式如下：

$$yh_i^n = yh_{i-1}^{n-1} + yh_i^{n-1}$$

即n次幂的第i项系数可以由$n-1$次幂的第$i-1$项系数和第i项系数相加得到。把$n(n=0\sim k)$次幂的二项式系数表排列在一起的数据阵列称为杨辉三角形。5 次幂的杨辉三角形如下所示。

```
1
1   1
1   2   1
1   3   3    1
1   4   6    4    1
1   5   10   10   5   1
```

由于每行元素都是在上一行的基础上计算出来的，因此可以用一维数组进行迭代。数组长度根据二项式的幂次决定，可以使用动态数组存放系数表。根据以上分析，可以写出

以下输出杨辉三角形的程序。

```
1   //ex5-38.cpp:输出杨辉三角形
2   #include<iostream>
3   using namespace std;
4   //函数定义
5   void yhtriangle(int* yh,int n)
6   {
7      for(int i=0;i<n;i++)
8      {
9          yh[i]=1;                                    //每行的第一个和最后一个元素为1
10          for(int j=i-1;j>0;j--)
11          {
12              yh[j]=yh[j-1]+yh[j];                   //中间元素是上一行相邻两元素之和
13          }
14          //输出i次幂二项式系数表
15          for(int k=0;k<= i;k++){
16              cout<<yh[k]<<"  ";
17          }
18          cout<<endl;
19      }
20  }
21  int main(){
22      int n;
23      cout<<"Please input power (0<n<= 20):\n";
24      do {
25          cin>>n;
26      } while (n<0||n>20);
27      int* yh=new int[n+1];                          //申请内存
28      yh[0]=1;                                       //初始化第一行的第一个元素为1
29      yhtriangle(yh,n);                              //调用函数,输出杨辉三角形
30      delete[] yh;                                   //释放内存
31  }
```

程序运行结果：

```
Please input power (0 < n <= 20):
6
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
```

例5.38中，第5～20行是输出杨辉三角形的函数，外层循环遍历每一行元素，内层循环计算中间元素的值，最后打印出每一行元素。main()函数中，第27行通过动态内存分配来存储杨辉三角形的一行，第30行释放内存。

例5.39 字符串排序。

排序可以使用前述的选择排序法或冒泡排序法，字符串比较可以调用字符串比较函数strcmp()。本例使用冒泡排序法对字符串进行排序。程序代码如下：

```
1   //ex5-39.cpp:使用冒泡排序法对字符串进行排序
2   #include<iostream>
3   #include<cstring>                                  //用于strcmp()函数
```

```
4   using namespace std;
5   void bubbleSort(const char * arr[],int n)
6   {  //冒泡排序函数,用于对C风格字符串数组进行排序
7       for(int i=0;i<n-1;++i){
8           for(int j=0;j<n-i-1;++j){
9               if(strcmp(arr[j],arr[j+1])>0){
10                  const char * temp=arr[j];       //交换arr[j]和arr[j+1]
11                  arr[j]=arr[j+1];
12                  arr[j+1]=temp;
13              }
14          }
15      }
16  }
17  int main()
18  {
19      //创建一个C风格字符串数组
20      const char * name[]={"Li Hua","Zhang Ming","Liu Lei","Sun Fei","He Xiaoming"};
21      int n=sizeof(name)/sizeof(name[0]);
22      bubbleSort(name,n);                         //使用冒泡排序法对数组进行排序
23      cout<<"Sorted names:"<<endl;                //输出排序后的数组
24      for(int i=0;i<n;++i){
25          cout<<name[i]<<endl;
26      }
27  }
```

程序运行结果：

```
Sorted names:
He Xiaoming
Li Hua
Liu Lei
Sun Fei
Zhang Ming
```

第5～16行是冒泡排序函数，函数的第一个形参是指向字符数组的指针，第二个参数指定参加排序的字符串的个数。第9行比较两个字符串的大小时，调用了字符串比较函数strcmp()。

本章小结

指针是用来存储内存地址的变量。指针指向它存储的内存地址。指针定义说明了指针指向对象的类型。指针的间址运算即对指针指向的内存单元进行访问。

引用是一种特殊的变量类型，它是一个已存在对象的别名，一旦定义后不能重新绑定到另一个对象。

C++的函数名表示了函数的入口地址，通过将函数指针作为参数，可以调用不同名称的同类型函数。

数组是同类元素的集合，在内存中占据一片连续的存储区域，通过下标可以访问数组元素。指针和数组紧密相关。一维数组的名称表示了数组第一个元素的地址，是一个常指针。数组可以通过下标方式访问，也可以通过指针方式访问。

C风格字符串是以空字符('\0')结束的字符数组。可以用字符数组、字符串常量和字

符指针来表示字符串。C 语言标准库提供了一系列处理 C 风格字符串的库函数。

new 运算符允许在程序运行时为数据对象请求内存。如果不再需要动态分配的内存空间，则可以使用 delete 运算符释放。

习题 5

习题 5