第6章 指针

[redacted]针的概念

指针即内存单元的地址。要理解指针的概念，首先要了解计算机存储器存储数据的情况。在计算机运行时，程序与数据是存放在内存中的。内存的基本存储单元是字节，计算机的内存就是由许多字节组成的，每个字节都有编号，这些编号就是地址，也就是指针。CPU 就是通过地址寻找到数据及指令的。

当我们通过语句定义了一个变量（如 int x=3）时；系统将在内存中为这个变量开辟一个空间以存放变量的值，这个变量就具备了几方面的特性：变量名，变量的类型，变量的值，变量存放的地址。其中变量名与变量的地址是对应关系，就像学生的姓名与其学号有一个对应关系一样。

在用高级语言编写程序时，一般情况下，通过变量名就可以访问变量。但为了增加程序设计的灵活性和提高程序的效率，有时需要对内存的地址进行操作，这就需要使用指针变量。

[redacted]量的概念

指针变量也是变量的一种，与其他变量不同的是，指针变量存放的是变量的地址。

例如，在内存中依次存放了一组数据，每个数据的存储地址如图 6-1 所示，如果将某个数据的地址作为变量的值存放在内存中，则这样的变量就是指针变量。

地址	内容
2000	1
2004	2
2008	3
2012	4
2016	5
2020	2000

变量的地址

图 6-1　变量在内存中的存储

[redacted]量的定义

C++定义指针变量的格式如下：

基类型　*变量名

说明：

（1）基类型指的是指针变量所指变量的类型。

（2）*是一个说明符，用于定义指针变量。

例如：int　*p1;

float　*p2;

char　*p3;

若定义了如下变量：

```
int   x;
float   y;
char   z;
```

则可通过赋值语句将变量 x、y、z 的地址赋给指针变量 p1、p2、p3：

```
p1=&x;p2=&y;p3=&z;
```

此时，可以形象地说 p1 指向了 x，p2 指向了 y，p3 指向了 z。

[redacted]基本操作

与指针有关的操作有取地址、间接访问、赋值和指针变量的运算。

1. 取地址

运算符：&

功能：获取变量的地址。

例如，定义整型变量 x 和指向整型变量的指针变量 p 的语句如下。

```
int   x;
int   *p;
```

可以通过取地址运算符&将 x 的地址赋给指针变量 p。

```
p=&x;
```

此时，p 指向了 x。

2. 间接访问

运算符：*

功能：访问指针变量所指向的变量。

例如：

```
int   x;          //定义整型变量 x
```

```
int   *p=&x;      //定义指向整型变量的指针 p，同时将 p 初始化为变量 x 的地址。
*p=3;             //此语句执行后将给变量 x 赋值 3，相当于 x=3。
```

可见，使用间接访问运算符“*”，可以通过指针变量访问它所指的变量，如图 6-2 所示。

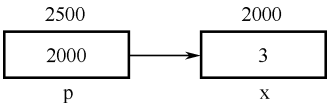

图 6-2　p 单元存放单元 x 的地址

值得注意的是：在通过指针变量给其他变量赋值之前，必须使指针有明确的指向，否则可能会出现严重问题。请比较如下两段程序：

```
int   x;                                    int   x;
int   *p=&x;                                int   *p;
*p=3;          ----------------正确          *p=3;          -------------错误
```

请注意，右边程序段中的 p 是一个无向指针变量，它的指向不确定，如果它刚好指向程序的某个使用中的单元，则这个单元的值被改变，程序就会出现预料之外的结果。

3. 赋值

在 C++中，对指针变量的赋值只能有 3 种操作：同类型指针变量之间的赋值运算、通过取地址运算符&取变量的地址赋给指针变量、给指针变量赋 0 值。

例如：

```
int   x;
int   *p1,*p2,*p3;
p1=&x;
p2=p1;
p3=0;
```

程序段执行后，p1 获得变量 x 的地址， p2 亦获得 x 的地址，即 p1 指向 x，p2 亦指向 x，p3 的值为 0。

给指针变量赋 0 值还可以写成 p=NULL。

NULL 是一个符号常量，代表数据 0，在 iostream 头文件中已定义：#define NULL 0。

两个指针变量之间进行赋值时，类型必须匹配，否则编译时会出错。

4. 指针变量的运算

指针变量是一种特殊的变量，它存放的是地址，所以指针变量的运算是关于地址的运算。指针变量的运算种类是有限的，只能进行算术运算和关系运算。

（1）算术运算

指针变量可以执行的算术运算有指针变量加或减一个整型值、指针变量自加或自减、指向同一存储区的两个指针变量相减。

① 指针变量加上或减去一个整型值

设 p 是已定义的指针变量，n 是一个整型值，则可以有：

```
p+n、p-n
```

p+n 的运算结果为：指针变量 p 的值加上 n 个它所指的变量所占的字节，结果为 p 所指变

量之后（地址增加方向）第 n 个变量的地址。

例如：

```
int   x,y,z;
int   *p=&x;*p1;
p1=p+2;
```

程序段执行之后，p1 的值为 p 所指变量之后的第 2 个变量的地址，即 z 的地址，如图 6-3 所示。

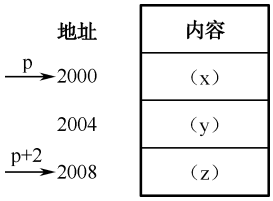

图 6-3　p+i 表示第 i 个单元

在上例中，假设在内存中，x、y、z 三个变量的存放情况如图 6-3 所示，则 p 的值为 2000，p、p1 的基类型均为整型，每个变量占 4 个字节，故 p1=2000+2×4=2008。

p-n 的运算规则同上，但执行的是减法，故结果为 p 所指变量之前（地址减小方向）第 n 个变量的地址。

② 指针变量自加或自减

```
p++，p--
```

与普通变量的自增、自减运算相同，指针变量的自增自减运算是使指针变量的值增 1 或减 1，即 p++相当于 p=p+1，p--相当于 p=p-1。

但是，请注意，这里的加 1 或减 1 的含义是一个 p 所指向的变量类型所占的字节数。也就是说，如果 p 指向了某个变量，p++则使 p 指向下一个变量，或 p--使 p 指向前一个变量。

③ 两个同类型指针相减

这一运算仅在两个指针指向同一连续存储区时才有意义，其结果为两指针相距的数据个数，如图 6-4 所示。

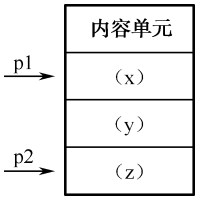

图 6-4　两个单元的距离

$$p1-p2=\frac{p1\text{所指变量的地址}-p2\text{所指变量的地址}}{\text{一个该类型变量所占字节}}$$

指针相减运算通常用于以指针方式访问数组元素的场合，可以计算两个指向同一数组的指针相距的元素个数。

（2）关系运算

两个同类型指针可以进行关系运算，但仅在两个指针指向同一个数组时，比较才有意义。

例如：

```
int     a[4]={1,2,3,4,},*p=&a[0],*q=&a[3],sum=0;
while(p<=q){sum+=*p;p++;}
```

指针还可以与 NULL 作相等或不等比较。通常用于访问链表节点时判别是否到达链表的尾节点。

例如：if(p= =NULL)　p=p1;

[redacted]变量访问变量

在 C++程序中，如果要访问某个变量，可以通过变量名来访问，也可以通过变量的地址来

访问。

【例 6.1】　通过指针访问变量。

```
#include<iostream>
using    namespace std;
int    main(    )
{       int      a,b;
        int      *pointer_1,*pointer_2;
        a=100;b=10;
        pointer_1=&a;    pointer_2=&b;
        cout<<a<<′      ′ <<*pointer_1;
*pointer_2=50;     //通过指针变量将变量 b 的值改变为 50
cout<<b<<′      ′ <<*pointer_2;
return 0;
}
```

运行结果为：

```
      100    100
       50     50
```

【例 6.2】　通过指针交换变量的值。

```
#include<iostream>
using    namespace std;
int    main(    )
{       int *p1,*p2,t,a,b;
        p1=&a;    p2=&b;
        cin>>a>>b;
        if(a<b)    {    t=*p1;*p1=*p2;*p2=t;}
         cout<<*p1<<′      ′ <<*p2;
         cout<<a<<′      ′ <<b;
          return 0;
}
```

若输入 4　5

运行结果为：

```
 5   4
 5   4
```

[redacted]为函数参数

从上节的两个例子可以看出，在定义了某种类型的变量，并定义了同类型的指针变量后，则对变量的访问既可以用变量名，也可以用指向该变量的指针。那么，定义指针变量有什么意义呢？

请分析下面的例 6.3。

【例 6.3】　输入两个整数 a 和 b，判别其大小，如果 a<b 则将两数对换。

```
#include<iostream>
using   namespace std;
void   swap(int   p1, int   p2)
{     int     temp;
      temp=p1;     p1=p2;     p2=temp;
}
int     main(   )
{   int     a,b;
       cin>>a>>b;
       if(a<b)   swap(a,b);       //如果 a<b，则调用 swap 函数将两数对换。
       cout<<a<<'   '<<b;
        return 0;
}
```

在上面的程序中，函数 swap 实现两数对换功能，在主函数中输入两个整型数，判别其大小，若 a<b，则调用 swap 函数。但上机运行发现，两个数没能实现对换。这是因为，实参向形参传递数据是值传递，实参 a、b 的值对应传递给了 p1、p2，而 p1、p2 的值在函数 swap 的运行过程中得到了对换，但它们不能影响 a、b 的值。因为实参与形参占据的是不同的内存单元，如图 6-5 所示。

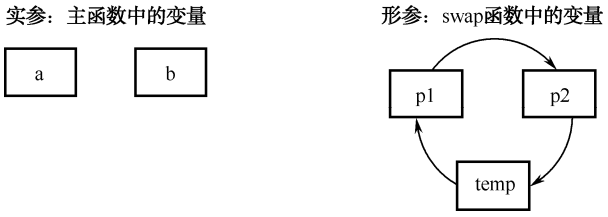

图 6-5　形参与实参的区别

这个问题可以通过运用指针得到解决。

【例 6.4】 通过指针变量间接修改其指向的变量，从而交换两个数。

```
#include<iostream>
using   namespace std;
swap(int   *p1, int   *p2)
{     int     temp;
      temp=*p1;     *p1=*p2;     *p2=temp;
}
 int   main(   )
{   int     a,b,*pointer1=&a,*pointer2=&b;
    cin>>a>>b;
    if(a<b)
          swap(pointer1,pointer2);       //如果 a<b，则调用 swap 函数将两数对换。
    cout<<a<<'   '<<b;
return 0;
 }
```

注意，在例 6.4 中，函数的参数是指针变量，那么在调用 swap 函数时，是把变量 a、b 的地址传递给了 p1、p2，因此在函数 swap 中，通过指针变量 p1、p2 访问的是主函数中变量 a、

b，从而实现了 a、b 两数对换，如图 6-6 所示。

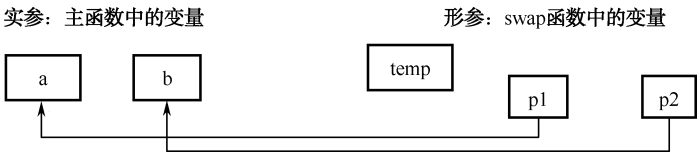

图 6-6　通过指针实现两个变量交换

[redacted]与一维数组

在前面的学习中，我们已经了解到，数组是连续存放的一组同类型数据，数组名代表数组的首地址，如果定义一个指针变量存放数组首地址，则这个指针变量指向数组首元素，程序中就可以通过指针变量访问数组元素。

【例 6.5】 通过指针变量访问数组元素。

```
#include<iostream>
using    namespace std;
int    main( )
{ int    a[10];              //定义含有 10 个整型数的数组
  int    *p;                 //定义指向整型数据的指针变量，
  p=a;                       //使指针变量 p 指向数组的首地址
  for (i=0; p+i<a+9;i++)
    cin>>*(p+i);             //顺序输入 10 个整型数据给数组 a 的 10 个元素，
  for (;p<a+9;p++)
    cout<<*p;
  return 0;
}
```

在例 6.5 中，要注意以下两个问题。

（1）主函数第 3 句 p=a，是将数组首地址赋值给指针变量 p，由于数组名实质上代表数组的首地址，因此不用取地址运算符，如果写成 p=&a 是错误的。但如果将某个数组元素的地址赋值给指针变量，则需要取地址运算。例如：p=&a[0];或 p=&a[5];。

（2）例中的两个 for 循环的作用分别是逐个给数组 a 的元素赋值，以及逐个输出数组元素的值，均使用了指针变量访问数组元素，但它们的使用形式不同。第 1 个 for 循环中，p 始终指向首元素，对其他元素的访问通过 p+i 计算地址实现。而第 2 个 for 循环中，指针变量 p 开始时指向首元素，每完成一次输出，通过运算 p++使 p 指向下一个元素。

在前面的学习中我们知道，对数组元素的访问可以通过数组名+下标的形式，如：cin>>a[0]; 或 cout<<a[i]; 。现在我们又了解到，可以通过指针变量对数组元素进行访问，下面把对数组元素的访问方式归纳如下。

（1）下标法。

数组名[下标]。如 a[0]、a[5]等。

在 C++中，方括号[ ]也是运算符——数组元素访问运算符，也称下标运算符。a[i]的运算规则为：先按 a+i×d 计算数组中第 i 个元素的地址，然后访问它。其中，d 为一个该类型的数

组元素所占字节数。例如，若 a 为整型数组，则 d=4，若 a 为双精度型数组，则 d=8。

根据下标运算符[ ]的运算规则，若定义了指向数组首元素的指针变量 p，也可以通过 p[0]、p[5]的形式访问数组中的元素。

（2）地址法。

定义指针变量指向数组首元素，然后通过*(p+i)的形式访问第 i 个元素（参见例 6.5）。

由于数组名也具备地址的性质，故例 6.5 中的*(p+i)也可改为*(a+i)。这两种方式实现的效果相同，不同之处在于：p 是指针变量，它的指向是可变的，而 a 是指针常量，它已被定义指向数组首地址，不能改变。

（3）指针法。

定义指向数组首元素的指针变量，通过*p 的形式访问 p 所指向的数组元素，并通过 p++运算使 p 指向下一个元素（参见例 6.5）。

比较以上 3 种方法，下标法比较直观，但效率不高，因为每次访问数组元素前都要根据下标计算元素的地址。地址法的运行效率与下标法相同。指针法不太直观，但它的运行效率比前两者高，因为不需要计算元素的地址，而 p++这样的运算是比较快的。

[redacted]针与字符串

在 C++中，字符串可以用数组来处理，因此也可以通过指向字符的指针来处理字符串。

【例 6.6】 将一个字符串逆序输出，用指针处理。

```
#include<iostream>
using   namespace std;
int   main( )
{ char    c[10]=“ABCDEFGHI”;    //定义含有 10 个字符型数据的数组
   char    *p;    //定义指向字符型数据的指针变量，
  p=&c[9];          //使指针变量 p 指向数组的最末一个元素
  for (; p>=c;p--)
cout<<*p;      //自尾向前逐个输出字符串中的字符
  return 0;
}
```

运行结果：IHGFEDCBA

在程序中，定义了指向字符型数据的指针变量 p，并使它指向数组最末一个元素，在循环中，通过 p--改变 p 的指向，由后向前逐个输出字符。

请注意比较下面的例 6.7 中指针变量的使用。

【例 6.7】 用指针变量输出字符串。

```
#include<iostream>
using   namespace std;
int   main( )
{   char    c[10]=“ABCDEFGHI”;    //定义含有 10 个字符型数据的数组
    char    *p;                   //定义指向字符型数据的指针变量，
    p=&c[0];                      //指针变量 p 指向数组的第一个元素
    cout<<p<<endl;
```

```
    return 0;
}
```

运行结果：ABCDEFGHI

如果指针变量 p 是字符型指针，那么当它作为 cout 的输出项时，输出的是它所指向的字符串，直到遇到“\0”才停止。

[redacted]的

指针变量可以指向某个元素，也可以指向一组连续存放的元素，即指向一维数组。指向一维数组指针变量定义的格式如下：

```
基类型　（*变量名）[m];
```

例如：

```
int　(*p1)[5];
double　(*p2)[6];
```

p1 为指向 5 个连续存放的整型数据的指针变量，p2 为指向 6 个连续存放的双精度数据的指针变量。

以这种格式定义的指针变量不是指向一个元素，而是指向一组元素。若有运算 p1++，则 p1 指向下一组元素，即当 p1 增值 1 时，其地址增加一组数据所占的字节数。

指向由多个元素组成的一维数组的指针变量也称为行指针。

指向由多个元素组成的一维数组的指针变量可以用于处理多维数组。

[redacted]与二维数组

一个 m 行 n 列的二维数组，可以看成由 m 个元素组成的一维数组，而这个数组的每个元素又是由 n 个元素组成的一维数组。这种分解多维数组的方式对于用指针变量处理多维数组带来了很大方便。

假设定义了如下 2 行 3 列的数组：

```
int　a[2][3];
```

我们可以将它看作是由两个元素组成，每个元素是一个由 3 个元素组成的一维数组，如图 6-7 所示。

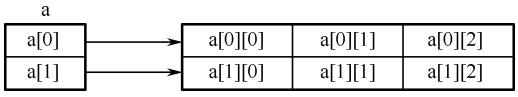

图 6-7　二维数组的表示与存储

a[0]为第一行的数组名，它指向第一行的首元素，即 a[0]的值为元素 a[0][0]的地址；a[1]为第二行的数组名，它指向第二行的首元素，a[1]的值为元素 a[1][0]的地址。

请注意二维数组名 a 的性质是什么？由于组成 a 的元素是一维数组，因此 a 是指向一维数组的指针变量，可称为行指针。a+1 则指向下一行。*a 和*(a+1)则等价于 a[0]和 a[1]，是分别指向 a[0][0]元素和 a[1][0]元素的指针，称为元素指针。

【例 6.8】 通过行指针变量访问二维数组。

```
#include<iostream>
using   namespace std;
int main()
{
    int   a[3][3]={1,3,5,7,9,2,4,6,8};
    int    (*p)[3];
    int   i,j;
    p=a;          //a、p 均为指向 3 个整型数的行指针变量，可以互相赋值
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            cout<<p[i][j];
    return 0;
}
```

[redacted]与指针数组

1. 概念

指针变量中存放着某个变量的地址，我们称指针指向了该变量。指针变量可以指向整型、实型、字符型或其他类型的变量，如果它指向的是指针变量，那么它就是指向指针的指针变量，即多级指针。如图 6-8 所示，x 是整型变量，其存放在地址为 2000 的内存单元中；p 是指向 x 的指针变量，ps 是指向 p 的指针变量，这样 ps 是二级指针。如果定义指针 pd 指向 ps，则 pd 是三级指针。

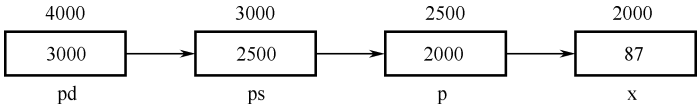

图 6-8　指向指针的指针

2. 多级指针的定义

```
int   x=87;
int *p=&x;              //定义指向整型变量的一级指针
int   **ps=&p;          //定义二级指针
int   ***pd=&ps;        //定义三级指针
```

3. 使用多级指针访问变量

【例 6.9】 使用多级指针访问变量。

```
#include<iostream>
using   namespace std;
int main()
{
int   a[9]={1,3,5,7,9,2,4,6,8};
```

```
int     *p1=a;
int      **p2=&p1;
cout<<**p2<<′ \t′ ;
p1=&a[4];
cout<<**p2<<endl;
return 0;
}
```

运行结果为：

```
1        9
```

在例 6.9 中，p2 为二级指针，对它进行一次间址运算*p2，得到它所指向的指针变量 p1，第二次间址运算**p2，相当于*p1，结果为 p1 所指向的变量。

4．指针数组

数组是若干同类型数据的集合，若组成数组的元素都为指针变量，则这个数组就是指针数组。指针数组的定义形式如下。

```
int     *p[5];
```

定义了一个指针数组，内含 5 个指针变量。

【例 6.10】 利用一维指针数组引用二维数组中的元素。

```
#include<iostream>
using    namespace std;
int main()
{
      int    a[3][3]={1,3,5,7,9,2,4,6,8};
      int i,j;
      int     *p[3];
      for(i=0;i<3;i++) p[i]=a[i];              //将每行的行首地址赋值给指针数组中的指针变量
      for(i=0;i<3;i++)
      { for(j=0;j<3;j++)
            cout<<p[i][j]<<′  ′ ;
         cout<<endl;
      }
      return 0;
      }
```

运行结果：

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

例 6.10 中，指针数组各元素的指向如图 6-9 所示。

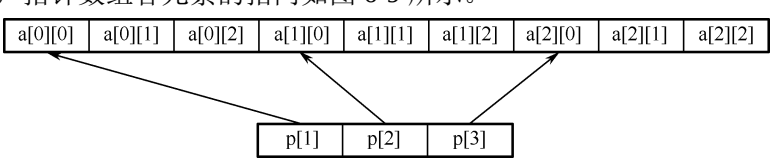

图 6-9　各指针与数组各行的对应关系

指针数组一般用于处理二维数组。使用指针数组处理多个字符串比用二维数组处理字符串更灵活方便。

【例 6.11】 将 5 个字符串按字典顺序排序后输出。

```
#include<iostream>
using   namespace std;
int main()
{      char     *s[5]={ "Assembly Language Programming ",
                        "Introduction of Computer Science",
                        "Computer Operating System",
                        "Constitute of Computer",
                        "technology of Network"};
       char   *p;
       int   i,j,k;
       for (i=0;i<4;i++)
       {     k=i;
             for(j=i+1;j<5;j++)
                 if(strcmp(s[k],s[j])>0)   k=j;
             if(k!=i)
             {       p=s[i];s[i]=s[k];s[k]=p;   }
       }
     for(i=0;i<5;i++)
         cout<<s[i]<<endl;
     return 0;
}
```

运行结果：

```
Assembly Language Programming
Computer Operating System
Constitute of Computer
Introduction of Computer Science
technology of Network
```

在例 6.11 中，定义了含 5 个字符型指针变量的指针数组，每个指针指向一个字符串。其内存中的指向关系如图 6-10 所示。程序采用选择法排序。排序时，改变的是指针数组各元素的指向而字符串的顺序不变。经排序后指针数组中各指针的指向关系如图 6-11 所示。

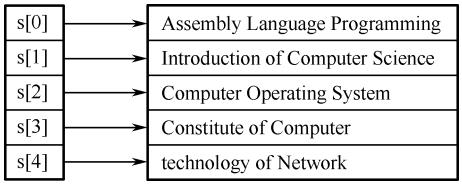

图 6-10　指针数组与多个字符串

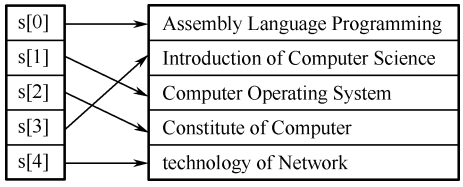

图 6-11　排序后的指针数组与多个字符串

[redacted]指针的函数

函数在调用结束时可以通过 return 返回一个值，这个返回值可以是普通类型变量，也可以是指针变量。当返回一个指针变量时，该函数就称为指针类型的函数。

指针类型函数的定义格式：

```
数据类型　　*函数名（参数表）
{
　　函数体
}
```

例如：

```
float　*max(float　*x,float　*y )
{　　if(*x>*y)return　x;
　　else　return y;
}
```

定义了一个返回 float 型指针的函数。

在定义返回指针的函数时，要特别注意变量的作用域问题。如下面的函数是不正确的。

```
char　*getname(　)
{　char　name[20];
　　cout<<"Enter your name";
　　cin>>name;
　　return name;
}
```

这个函数有错误，因为 name 是局部变量，当函数结束时，它所占用的空间被释放，虽然其首地址返回到主函数中，但不能通过这个首地址访问原字符串。

[redacted]指针

1. 函数指针的概念

在程序运行时，不仅数据要占据内存空间，程序的代码也被调入内存并占据一定的空间。一个函数被编译连接后生成一段二进制代码，该段代码被存入内存时的首地址称为函数的入口地址。而函数名就表示函数的首地址。

函数的首地址即函数指针。

2. 指向函数入口地址的指针变量

如果定义指针变量用于存放函数的首地址，这样的指针变量称为函数指针。

函数指针的定义格式：

```
数据类型　（*指针变量名）（形参列表）
```

例：　int　(*p)(int　x,int　y);

p 为函数指针变量，指向一个有两个整型参数、返回值为整型的函数。

定义了函数指针变量之后，就可以通过它调用函数。函数名实质上也是函数指针，通过函数名调用函数我们在前面的章节中已经学过。但函数名是指针常量，它的指向不能改变。而函数指针变量则可以改变指向，因此在程序中可以给函数指针赋不同的值，使它能调用不同的函数，以提高程序设计的灵活性。

3. 函数指针的使用

在使用函数指针进行函数调用之前，要对它进行赋值，使指针指向一个已经存在的函数的起始地址。

给函数指针赋值的一般形式如下：

```
函数指针名=函数名;
```

通过函数指针调用它所指向的函数的一般形式如下：

```
函数指针名（实参表）;
```

【例 6.12】 根据输入要求执行不同函数。

```
#include<iostream>
using   namespace std;
int main()
{    int   i;
     void   (*p)();
      void   print_star();
      void     print_message();
     while(1)
      {      cin>>i;
             if(i==3) break;                          //输入 3 则退出程序
             switch(i)
             {      case 1: p=print_star;      break;     //为函数指针赋值
                    case 2: p=print_message;      break;
                    default: cout<<"输入错误，请输入 1～3 的数字"<<endl;
                             continue;                  //若输入错误则结束本次循环，进行下一次循环
             }
             p();                                     //通过函数指针调用函数
}
return 0;
}
void   print_star()
{   cout<<”**********”<<endl; }
void   print_message()
{   cout<<"Hello   C++!"<<endl; }
```

程序根据用户输入执行不同的函数。通过指向函数的指针 p 实现函数的调用，而 p 的指向由用户输入的数字决定，在 switch 结构中赋不同的值。

在上面的例子中，使用函数名调用函数也可实现相同的功能，为什么要定义一个函数指针呢？事实上，函数指针主要用于作为函数的参数。C++允许在调用一个函数时把另一个函数作为参数传给被调用函数。这时，被调用函数的形参定义为函数指针，调用的实参为一个函数名。这样就可以在调用一个函数的过程中根据给定的不同实参调用不同的函数。

【例 6.13】 函数指针作为函数的参数。

用一个函数 process 处理两个数据，第 1 次调用求出两数中的大者，第 2 次调用求出两数中的小者，第 3 次调用求出两数之和。我们可以分别写 3 个函数 max、min、add 实现这 3 种功能，在调用函数 process 时，分别以不同的函数名作为实参，以完成不同的功能。

```
#include<iostream>
using   namespace std;
int max(int   x, int   y);
int min(int   x, int   y);
int add(int   x, int   y);
int process(int   x, int   y, int   (*p)( int   i, int   j));
int   main()
{     int   a,b,c;
      cout<<"Enter   a ,b:"<<endl;
       cin>>a>>b;
       c=process(a,b,max);
       cout<<"max="<<c<<endl;
       c=process(a,b,min);
       cout<<"max="<<c<<endl;
c=process(a,b,add);
       cout<<"max="<<c<<endl;
}
int add(int   x, int   y)
{     return   x+y;     }
int   max(int   x, int   y)
{     if(x>y)     return   x;
        else   return   y;
}
 int   min(int   x, int   y)
 {     if(x<y)   return   x;
        else   return   y;
}
int process(int   x, int   y, int   (*p)( int   i, int   j))
{   int   z;
    z=p(x,y);
return   z;
}
```

从例 6.13 中可以看到，不论是求最大值、最小值，还是求和，process 函数的语句均无须变动，而只是在调用时给出不同的实参，它就可以实现不同的处理。这样就增加了函数使用的灵活性。

[redacted]概念的总结

指针是 C++中一个非常重要的数据类型，它提供了一种通过地址访问内存单元的手段，使用它可以对各种类型数据进行快速有效的操作。有些数据结构通过指针可以很方便地实现，而

用其他方法则比较难。但是，指针又是较难掌握的概念。如果使用不当，带来的破坏性也很大，而且产生的错误和故障比较隐蔽，难以排查。因此学习中要注意弄清概念，多思考，多比较，深入理解指针的本质，只有掌握其使用方式，才能写出灵活高效、质量优良的程序。下面对本章讲述的与指针相关的概念进行归纳。

1. 指针和指针变量

指针即内存单元的地址。存放内存单元地址的变量就是指针变量。

2. 各种有关指针的数据类型的定义

```
int   *p;
```

定义 p 为指向整型数据的指针变量。

```
int    *p[n];
```

定义 p 为指针数组，其中含 n 个指向整型数据的指针变量。

```
int    (*p)[n];
```

定义 p 为指向连续存放的 n 个整型数据的指针变量，也可称为行指针。

```
int    **p;
```

定义 p 为一个二级指针变量，指向整型指针。

```
int    (*p)()
```

定义 p 为指向函数的指针变量，该函数返回整型值。

3. 有关指针的运算

指针变量是存放地址的变量，因此指针的运算是关于地址的运算，其运算结果是地址值。指针的运算结果与指针变量的基类型有关。例如，有如下定义：

```
int   a[2][3]={{1,2,3},{4,5,6}};
int   *p1=&a[0][0];
int   (*p2)[3]=&a[0];
```

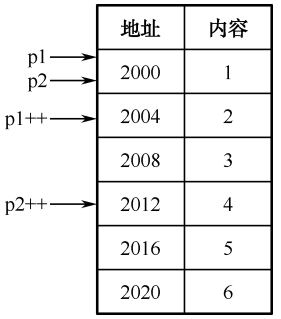

图 6-12　不同基类型的指针

p1、p2 获得同一个元素的地址，但它们的基类型不同，p1 指向单个整型变量，p2 指向 3 个连续存放的整型变量。假设数组 a 的首地址为 2000，则定义后 p1、p2 的值均为 2000，若对它们执行自加运算 p1++、p2++，p1 指向第 2 个数组元素，其值为 2000+4=2004，而 p2 指向第二行的第 1 个元素，其值为 2000+12=2012，如图 6-12 所示。

指针变量所能进行的运算是有限的。一般来说，指针变量可以实现的运算有如下 4 种。

（1）赋值运算。

（2）指针变量加（或减）一个整数，包括自加、自减运算。

（3）在一定条件下，两个指针变量相减。

（4）在一定条件下两指针变量进行关系运算。

[redacted]用

引用（Reference）是 C++的一个重要概念。它不是定义一个新的变量，而是为一个已定义的变量起一个别名。

1. 引用的定义

引用定义的格式如下：

```
类型　　&引用名=变量名
```

假设已定义整型变量 a：

```
int　a;
```

若需定义 a 的引用 b，可用如下语句：

```
int　&b=a;                          //声明 b 是 a 的引用
```

此时 b 和 a 占内存中的同一个存储单元，它们具有同一地址。

2. 引用的性质

引用是另一变量的别名，因此引用与被引用变量都代表同一变量。所有对引用的操作实际上都是施加在被引用的变量上。

【例 6.14】 引用和变量的关系。

```
#include<iostream>
using　namespace std;
int　main()
{　　int　a;
　　int　&x=a;
　　cin>>a;
　　cout<<"x="<<x<<endl;
　　x=x+5;
　　cout<<"a="<<a<<endl;
}
```

若输入的数据为 5，则运行结果为：

```
x=5
a=10
```

在例 6.14 中，变量 a 的值由键盘输入，而 x 是 a 的引用，它们占据同一内存单元，因此输出 x 即是输出 a 的值，x=x+5 也是施加在 a 上的运算。

3. 对引用变量的规定

在使用引用时，应注意如下几个问题。

（1）引用在定义时必须初始化。

（2）引用变量一经定义，不能再作为其他变量的引用（别名）。

下面的用法是错误的：

```
int　a1，a2;
```

```
int   &b=a1;          //定义 b 是 a1 的引用
      &b=a2;          //错误，不能使 b 又变成 a2 的引用（别名）
```

4. 引用作为函数参数

引用变量的主要用途是作为函数的参数或函数的返回值。

【例 6.15】 引用作为函数参数实现两数交换。

```
#include <iostream>
using namespace std;
int main( )
{     void swap(int &,int &);
      int i=3,     j=5;
      swap( i, j );
      cout<<″ i=″ <<i<<″     ″ <<″ j=″ <<j<<endl;
      return 0;
 }
void    swap(int &a, int &b)                    //形参是引用类型
{     int temp;
      temp=a;
      a=b;
      b=temp;
 }
```

运行结果：i=5　　j=3

与例 6.4 比较可以发现，引用作为函数参数，效果与指针变量作为函数参数一样，可以修改被调用函数中的变量的值。实质上，引用是一种“隐式指针”，通过它可以访问它引用的变量，且不需*运算符。引用变量的主要特点是与原变量保持地址一致。

指针是通过地址间接访问某个变量，而引用通过别名直接访问某个变量。使用引用作为函数参数，其语法更简单、直观、方便。因此，引用可以部分代替指针的操作。有些过去只能用指针来处理的问题，现在可以用引用来代替，从而降低了程序设计的难度。

指针是 C++语言的一个重要特色。通过指针可以对各种类型的数据进行方便灵活的访问。但同时，指针又是最容易令人困惑并导致程序出错的原因之一。因此在学习指针的过程中，必须明确概念，把握本质。本章介绍了指针的基本概念以及指针变量的应用。指针变量的本质是通过地址访问程序实体（如变量、数组、函数等），通常用于函数的参数传递。通过指针变量访问内存单元有利于提高程序的效率。

引用是一种隐式指针，使用引用部分代替指针，可以使程序语句更简单、安全、直观。

1．输入 3 个整型变量 i、j、k，设置 3 个指针分别指向这 3 个变量，并通过指针交换 3 个