第5章

指　　针

指针又称指针变量，是C++语言的精髓，也是其最强大的功能之一。因为有了指针，C++语言可以和汇编语言比效率。指针同时也是最棘手的功能之一，尽管有时候容易被误用，但在C++编程中起到至关重要的作用。在教学过程中，指针这部分内容常常给学生带来困惑，本章将详细介绍指针在C++程序中的用法。

5.1 指针的概念及声明格式

5.1.1 指针的概念

在2.2.3节中介绍过变量包含三个要素，分别是变量类型、变量名和变量的值，其中变量的值包括变量的数据值和变量的地址值。可以把变量的地址值存入另外一个变量，存储变量的地址值的变量称为指针变量。

1. 指针变量的相关术语

- 指针变量与一般变量不同，它的值是某个变量在内存中的地址。
- 一个指针变量存放了哪个变量的地址，就说这个指针指向了哪个变量。
- 指针变量必须初始化或者赋值(指向了变量)后，才能进行间接访问操作。

2. 内存空间的访问方式

- 通过变量名访问。
- 通过地址访问(通过指针间接访问它所指向变量的值)。

“&”是取地址符，例如：

```
int x;
```

语句声明整型变量 x，则 &x 表示变量 x 在内存中的地址。x 是整型数据，占 4 字节的内存单元，那么 x 的地址就可以存放在一个指针变量里，指针变量与其他类型变量一样，也必须先进行指针变量的声明。

5.1.2 指针变量声明格式

1. 指针变量声明

指针变量声明的一般格式如下：

```
存储类型  数据类型  *指针名=初始值(另一个变量的地址);
```

或者

```
存储类型  数据类型  *指针名;
```

例如：

```
int *pi=&x;
```

2. 指针的赋值

- 上面的语句声明了指针变量 pi，pi 是指针名，int 是数据类型，声明指针变量可以存储 int 类型变量的地址，不可以用其他类型变量的地址给指针变量 pi 赋初值或赋值。
- 上面的语句虽然没有给指针变量 pi 指定存储类型，但默认的存储类型为 auto 类型，即存储类型为自动类型，pi 是自动类型局部指针变量，存放在栈空间，由编译系统管理 pi 变量的所占用的内存单元。
- 声明语句中变量 pi 前的星号“*”非常重要，有了星号“*”，才能说明 pi 是指针变量，才能存放另外一个整型变量的地址，所有在声明语句中的星号“*”只是一个标记。
- 上面的语句声明了指针变量 pi，并且给该指针变量赋初值，初始值为 x 变量的地址，x 变量必须是声明了 int 类型的变量，那么就可以说 pi 指向了 int 类型的变量 x。
- 定义指针变量的时候可以赋初值，也可以不赋初值。如果赋初值，需要注意的是，初值变量必须在指针初始化之前已声明过，且变量类型应与指针类型一致。也可以用一个已赋初值的指针变量去初始化另一个同类型的指针变量。
- 既然 pi 是一个指针变量，那么 pi 也会占用内存单元，因此变量 pi 也有地址，&pi 就是指针变量 pi 的地址。不管指向什么类型变量的指针，所占用的空间大小是一样的，在 32 位编译器中都占用 4 个字节的内存单元，在 64 位编译器中都是占用 8 个字节的内存单元。
- 如果指针没有赋初值，则需在使用指针间接访问它所指向的变量前，一定要赋值。这很好理解，指针没有指向变量，怎么能间接访问其指向变量的数据值呢？因此说通过指针间接访问前，指针不能为空值。

3. 程序代码及运行结果

(1) 程序代码

【例 5.1】 指针定义。

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

```
int main()
{
    int x = 100;                                    //声明 x 变量,并赋初值 100
    int *pi = &x;                                   //声明 pi 指针变量,并赋值为 x 的地址
    cout << "x="<<x<< endl;                         //输出 x 变量的数据值
    cout << "x 的地址是"<<&x<<endl;                 //输出 x 变量的地址值
    cout << "*pi 是"<< *pi<<endl;                   //通过指针间接访问 x 变量的值
    cout << "pi="<<pi<<endl;                        //pi 变量的值,就是 x 的地址
    cout << "pi 的地址是"<<&pi << endl;             //pi 变量的地址值
    return 0;
}
```

(2) 程序分析及运行结果

以上代码在 VS2019(32 位编译器)环境下运行结果如图 5.1 所示,在 Dev-C++(64 位编译器)环境下运行结果如图 5.2 所示。

```
x=100
x的地址是00F2FBFC
*pi是100
pi=00F2FBFC
pi的地址是00F2FBF0
```

图 5.1 程序段运行结果(32 位编译器)

```
x=100
x的地址是0x6ffe0c
*pi是100
pi=0x6ffe0c
pi的地址是0x6ffe00
```

图 5.2 程序段运行结果(64 位编译器)

以 32 位编译器为例,分析上面程序的执行结果可以看出,pi 指针保存的是 x 变量的地址,第二条输出语句和第四条输出语句的结果是一样的,都是 x 变量的地址,00F2FBFC 是 8 个十六进制字符,一个 32 位的长整型数据,表示指向的变量在内存中的地址。

对于第二条输出语句

```
cout << *pi << endl;
```

输出结果为 x 变量的值 100,是属于通过指针变量间接地访问了变量 x 的值。在此你可能会有疑惑,这是声明语句 int *pi=&x 带来的困惑。在该声明语句中,看似 *pi 赋值为 &x,怎么到了输出 *pi 时,*pi 就成了 x 变量的值了呢?这是因为 int *pi=&x 语句为声明语句,声明语句中的“*”号只是标志,表示紧跟其后的变量 pi 是指针类型变量,是指针类型变量就可以把另外一个变量的地址赋值给它。离开了声明语句,此星号“*”就成了通过指针访问变量的取内容运算符。因此 cout<<*pi<<endl;是输出 pi 指针指向的内容。例 5.1 代码段中 x 变量的数据值、x 变量的地址值、pi 变量的数据值、pi 变量的地址值,在内存中的存储示意图如图 5.3 所示。

画出 pi 变量指向 x 变量的逻辑示意图很简单,逻辑示意图如图 5.4 所示。

说明:两个小矩形框分别表示 pi 变量和 x 变量,画一个箭头,箭尾画在 pi 指针变量的小矩形框里面,箭头指向变量 x 的小矩形框,不要指到 x 变量的矩形框里,这样就表示 pi 指针变量存放的是 x 变量的地址。

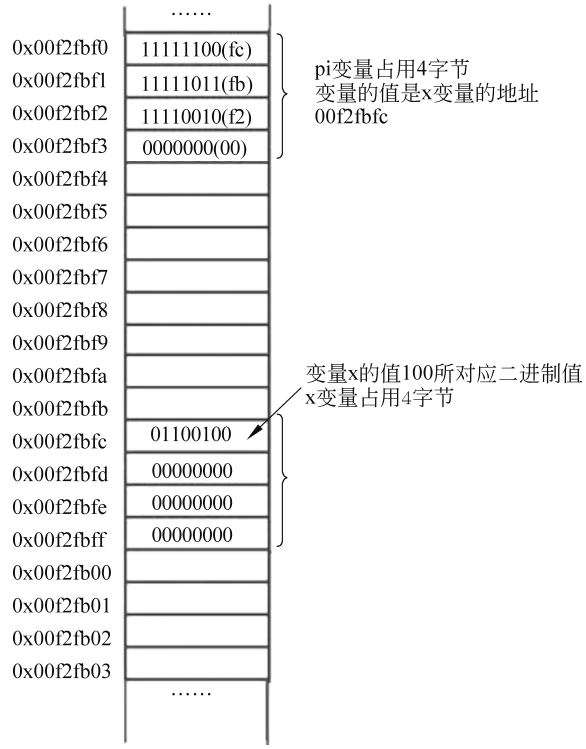


图 5.3 pi 指针指向 x 变量的物理内存存储示意图

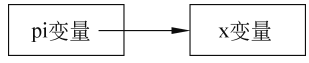


图 5.4 i 指针指向 x 变量的逻辑示意图

5.1.3 声明指向不同数据类型的指针

指针变量可以声明为指向任何数据类型的变量，即指针变量的值可以是任何数据类型变量的地址值。例如：

```
int *pi;
```

语句声明 pi 是一个指向 int 型变量的指针。

```
float *pl;
```

语句声明 pl 是一个指向 float 型变量的指针。

```
char *pc;
```

语句声明 pc 是一个指向 char 型变量的指针。

```
char (*pa)[3];
```

语句声明 pa 是一个指向一维数组的指针，此一维数组是长度为 3 的字符数组。

```
char *pm[3];
```

语句声明 pm 是一个指针数组，pm 是数组名，数组的三个元素存放字符变量的地址。

```
int (*pf)();
```

语句声明 pf 是一个指向函数的指针，该函数的返回的数据类型为 int 型数值。也就是说指针不仅可以指向数据变量，也可以指向代码段，函数就是代码段，函数名就是函数的地址。

```
int * fun();
```

语句声明函数的返回值是整型的指针变量。

```
int **pp;
```

语句声明 pp 是一个指向指针变量的指针，即二级指针。

要存储一个指针变量的地址，就需要一个二级指针，如果 pi 是指向整型变量的指针，pp 变量就可以存储 pi 变量的地址。一个星号“*”，* pp 表示间接访问 pp 指向变量的值，实际上是取得了 pi 变量的值，pi 变量是 x 变量的地址，因此**pp 才可以间接访问到 x 变量的值。

```
struct Date
{
    int year,month,day;                    //数据类型为整型的三个结构成员
};                                         //定义结构类型 Date
Date * pd;
```

在此加一个回车，换行语句声明指向结构变量的指针。

如果结构类型成员的指针类型是自身结构类型，称这种结构为自引用结构，也是定义链表的格式。例如：

```
struct Node
{
    int data;
    struct Node   * next;                  //结构类型成员是指向自身结构的结构类型的指针变量
};
```

称这种结构类型为自引用结构类型。

指向对象的指针如下：

```
class Person                               //定义 Person 类
{
private:                                   //私有访问权限
     char * name;                          //姓名   类的成员是指向字符的指针也叫字符串
     int age;                              //年龄
     char   gender;                        //性别
public:                                    //公有访问权限
    Person(char * n,int nl,char xb)        //类的构造函数
    {
        strcpy(name,n);
        age=nl;
        gender=xb;
    }
};
Person * pa;
```

语句声明 pa 是指向 Person 类对象的指针变量。

```
void * pv;
```

语句声明 pv 是空指针类型，允许声明指向 void 类型的指针，该指针可以被赋予任何类型对象的地址。

例如：

```
void *pv;
```

语句声明 pv 是 void 类型的指针。

```
int  *pint, i;
```

语句声明 pint 是整型变量的指针。

```
pv=&i;
```

语句声明 void 类型指针可以指向整型变量。

```
pint=(int *)pv;
```

语句说明用 void 指针给 int 指针赋值时，需要类型强制转换。

5.1.4 上机练习

【上机目的】

- 掌握声明指针变量的语法格式。
- 学会给指针变量赋初值和赋值，会使用指针间接访问其所指向的变量。
- 理解声明指针变量语句中的“*”与通过指针变量取内容运算符“*”之间的区别。

分析下列程序的运行结果，并上机操作，检查分析结果是否正确。

```
#include <iostream>
using namespace std;
int main()
{
    int *p;
    int x=57;
    cout<<"1:x="<<x<<endl;
    p=&x;
    cout<<"2:*p="<<*p<<",x="<<x<<endl;
    *p=58;
    cout<<"3:*p="<<*p<<",x="<<x<<endl;
    cout<<"4:p 的地址="<<&p<<endl;
    cout<<"5:p 的值="<<p<<endl;
    cout<<"6:x 的地址="<<&x<<endl;
    cout<<"7:x 的值="<<x<<endl;
    return 0;
}
```

【思考与练习】

1. 简答题

(1) 什么是指针？它的值和类型是如何规定的？

(2) 声明指针变量的一般格式是什么？举例说明声明不同指针类型的方法。

(3) 如何给不同类型的指针赋值和赋初值？

2. 单选题

(1) 已知语句：

```
int i, j=2; *p=&j;
```

可完成 i=j 赋值功能的语句是（　　）。

A. i=*p;　　B. p*=*&j;　　C. i=&j;　　D. i=**p;

(2) 若有以下声明语句，0<i<4，则（　　）是错误的赋值。

```
int a[4][10]; *p, *q[4];
```

A. p=a;　　B. q[i]=a[i];

C. p=a[i];　　D. q[i]=&a[2][0];

5.2 指针的运算

尽管指针中存放的是变量的地址，指针也能进行算术运算和关系运算。

5.2.1 使用指针访问数组元素

数组名是一个指针常量。一个数组的数组名就是该数组首元素的地址。指针常量不同于指针变量，指针常量的值在程序执行过程中不能被修改，而指针变量的值可以被修改。例如：

```
int a[10], *pa;
```

pa=&a[0]或 pa=a 两个赋值语句都是让 pa 指针指向数组 a 的首元素，经过上述声明及赋值后，可以使用指针访问数组元素。*pa 是 a[0]，*(pa+1)是 a[1]，…，*(pa+i)是 a[i]，因此 a[i]、*(pa+i)、*(a+i)、pa[i]是等价的，都表示数组下标为 i 的数组元素，但是不能写 a++，因为 a 是数组名，是数组首地址，是常量，常量不能被修改。

【例 5.2】 用指针访问数组元素。

```
#include <iostream>
using namespace std;
int main()
{
    int a[5]={5,4,3,2,1};
    int i,j;
    i=a[0]+a[4];                    //用下标法读取数组元素
    j=*(a+2)+*(a+4);                //用指针读取数组元素，等价于 j=a[2]+a[4]
    cout<<i<<endl<<j;
    return 0;
}
```

程序的输出结果为：

```
6
4
```

5.2.2 指针的算术运算

1. 指针与整数的加减运算

指针 p 加上或减去整型数据 n，是指针当前指向位置的后面或前面第 n 个数据的地址。这种运算的结果值取决于指针指向的数据类型。若 p 为字符指针，p+1 是指向下一个字符，实际增加 1 个字节。若 p 为整型指针，p+1 是指向下一个整数，实际增加 4 个字节。同样，若 p 为 double 型指针，p+1 是指向下一个 double 型数据，实际增加 8 个字节。

【例 5.3】 指针与整数相加。

```
#include <iostream>
using namespace std;
int main()
{
    int *pi,a[10]={0,1,2,3,4,5,6,7,8,9};
    double *pd,b[10]={9.1,8.1,7.1,6.1,5.1,4.1,3.1,2.1,1.1,0.1};
    pi=a;              //pi 为整型数组 a 的首地址
    pd=b;              //pd 为 double 型数组 b 的首地址
    for(int i=0;i<10;i++)
        cout<<pi+i<<" "<<*(pi+i)<<"----"<<pd+i<<" "<<*(pd+i)<<endl;
    return 0;
}
```

例 5.3 程序的运行结果如图 5.5 所示。

```
0x6ffd60 0----0x6ffd90 9.1
0x6ffd64 1----0x6ffd98 8.1
0x6ffd68 2----0x6ffda0 7.1
0x6ffd6c 3----0x6ffda8 6.1
0x6ffd70 4----0x6ffdb0 5.1
0x6ffd74 5----0x6ffdb8 4.1
0x6ffd78 6----0x6ffdc0 3.1
0x6ffd7c 7----0x6ffdc8 2.1
0x6ffd80 8----0x6ffdd0 1.1
0x6ffd84 9----0x6ffdd8 0.1
```

图 5.5 例 5.3 程序运行结果

程序分析：该程序声明了一个 int 型指针 pi 和一个 double 型指针 pd，分别声明了 int 型和 double 型数组，并且给数组赋初值。首先，让 pi 指向 int 整型数组的首元素，pd 指向 double 型数组的首元素。接着，将两个指针分别与 0 到 9 相加，显示指针与整数相加的结果，从图 5.5 可以看出，指针(地址)加上一个整数结果仍然是地址，地址的变化与指针所指向的类型有关，因为 int 型数据长度是 4 字节，double 型数据长度是 8 字节，因此 pi 加 1 按 4 字节增加，pd 加 1 按 8 字节增加。注意，在显示器上输出指针变量的值时，每次的运行结果不相同。读者每次运行该程序时 pi 和 pd 的值与图 5.5 中 pi 和 pd 的值也不一样。

2. 两个指针指向同一数组的数据元素时可做减法运算

两个指向同一数组元素的指针可以做减法运算，结果为两个指针之间相差数据元素的个数。例如 p 指向数组的首元素，q 指向数组的尾元素，q－p+1 则是数组的长度。

【例 5.4】 两个同类型的指针相减。

```
#include <iostream>
using namespace std;
int main()
{
    int *p,*q,a[10]={0,1,2,3,4,5,6,7,8,9};
    p=a;                                    //p 指针指向第 1 个数组元素
    q=&a[9];                                //q 指向第 10 个数组元素
```

```
    cout<<"数组的长度是: "<<q-p+1<<endl;   //输出数组的长度 10
    return 0;
}
```

注意：两个指针不能相加，如果想计算 p 和 q 两个指针的中间地址，用如下语句：

```
pm=p+(q-p)/2;
```

pm 是中间数组元素的地址值。计算中间数组元素下标与计算中间元素的指针不一样，如果 low 是数组元素的低下标，high 是数组元素的高下标，那么中间数组元素的下标为 mid=(low+high)/2，在数组中进行二分查找就是采用这个公式计算两个数组元素的中间元素的下标。

3. 指针的增 1 和减 1 运算

p++等价于 p=p+1，与 p+1 有区别，p++的结果是 p 指向下一个元素，p+1 的结果只是计算下一个元素的地址，而 p 指针本身的值不变，说明如下。

(1) y=*pi+1 和 y=*(pi+1)的区别

pi+1 结果是间接访问 pi 指针变量内容再加 1，(pi+1)结果是间接访问 pi 指针后面一个数据的内容，如果 pi 指针如图 5.6 所示。

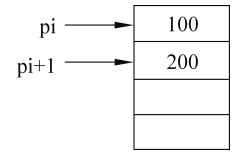


图 5.6　y=*pi+1 和 y=*(pi+1)

```
y=*pi+1;
```

则 y 的值是 101。

```
y=*(pi+1);
```

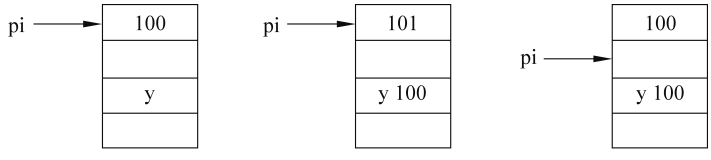
则 y 的值为 200。

(2) y=(*pi)++和 y=*pi++的区别

y=(*pi)++先间接地访问 pi 所指向变量的值，因为++是后置运算符，所以把 pi 所指向变量的值赋值给 y 变量后，pi 所指向变量的值又进行了加 1 运算。

y=*pi++先去访问 pi 指针所指向变量值赋值给 y 变量，然后是 pi 指针加 1。

假如指针 pi 如图 5.7(a)所示，在执行完 y=(*pi)++后，y 的值为 100，而 *pi 值为 101，指针 pi 的值不变，运算结果如图 5.7(b)所示。如果执行 y=*pi++，在运行后 y 的值也是 100，pi 执向了下一个单元，运算结果如图 5.7(c)所示。



(a) pi指针　(b) y=(*pi)++后　(c) y=*pi++后

图 5.7　y=(*pi)++语句与 y=*pi++语句的运行结果

5.2.3　指针的关系运算

两个同类型的指针可以进行==、!=、<、<=、>和>=这样的关系运算，指针的关系运算实际就是比较地址。如果 p 指向数组的第一个元素，q 指向数组的第 2 个数组元素，则关系表达式 p<q 的结果为真。

例 5.5 将定义数组元素逆置函数。函数的功能为将长度为 n 的 int 型数组中数组元素进行逆置。假如有长度为 n 的数组 array[n](n 是一个整型常量),数组元素逆置是指用 array [n－1]、array [n－2]、array[n－3]、……、array [2]、array [1]和 array[0]共 n 个数组元素分别给 array[0]、array[1]、array[2]、……、array[n－3]、array[n－2]和 array[n－1]共 n 个元素赋值,即把数组中的 n 个元素进行前后交换。比如数组 b[10]＝{10,20,30,40,50,60,70,80,90,100}共 10 个数据,b 数组逆置后,则 b[0]＝100,b[1]＝90,b[2]＝80,……,b[7]＝30,b[8]＝20,b[9]＝10。

【例 5.5】 指针的关系运算,分析程序的运行结果。

```
#include <iostream>
using namespace std;
void reverse(int a[],int n)
{
    int *p,*q,t;
    p=&a[0];                              //p 指向首元素
    q=&a[n-1];                            //q 指向尾元素
    while(p<q)                            //指针关系运算
    {                                     //p 指针指向变量的值与 q 指针指向变量的值进行交换
        t=*p;
        *p=*q;
        *q=t;
        p++;                              //修改 p 指针的值,p 指针后移
        q--;                              //修改 q 指针的值,q 指针前移
    }
}
int main()
{
    int b[10]={1,2,3,4,5,6,7,8,9,10};  //初始化 b 数组
    cout<<"逆置前:";
    for(int i=0;i<10;i++)
    cout<<b[i]<<"\t";                     //输出逆置前的 10 个数组元素
    cout<<endl;
    reverse(b,10);                        //调用数组逆置函数
    cout<<"逆置后:";
    for(int i=0;i<10;i++)
    cout<<b[i]<<"\t";                     //输出逆置后的 10 个数组元素
    cout<<endl;
    return 0;
}
```

例 5.5 数组逆置函数的运行结果如图 5.8 所示。

```
逆置前:1    2    3    4    5    6    7    8    9    10
逆置后:10   9    8    7    6    5    4    3    2    1
```

图 5.8 数组逆置函数运行结果