



西北工业大学
NORTHWESTERN POLYTECHNICAL UNIVERSITY

程序设计基础 Programming in C++


U10G13027/U10G13015

主讲：魏英，计算机学院

思考：

A、B、C三人欲借用旅馆的房间。A先到达旅馆，在服务台登记了房间，房号是5818。然后A电话通知了B，但没有通知C。

B和C该怎样找到A呢？

方法：B可以直接到5818找到A。

C可以到**旅馆的服务台**查询到A的房间号是5818，再找到A。

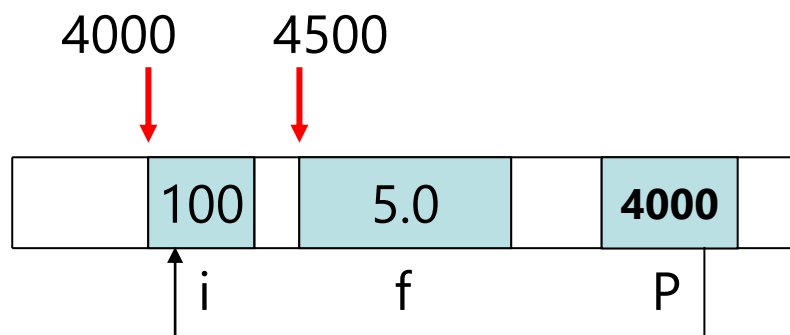
- ▶ 7.1 指针与指针变量
- ▶ 7.2 指针的使用及运算
- ▶ 7.3 指针与数组
- ▶ 7.4 指针与字符串
- ▶ 7.5 指针与函数
- ▶ 7.6 动态内存
- ▶ 7.7 带参数的main函数
- ▶ *7.8 引用类型

- ▶ 可以按变量名称**直接**访问内存单元，也可以通过指针**间接**访问内存单元。

7.1.1 地址和指针的概念

```
int i; //定义整型变量  
double f; //定义双精度浮点型变量
```

图7.1 变量的内存形式



按对象名称存取对象的方式称为对象直接访问。

一个对象的地址称为该对象的指针。

通过对象地址存取对象的方式称为指针间接访问。

7.1.2 指针变量

- ▶ C++将专门用来存放对象地址的变量称为指针变量，定义形式如下：

指针类型 *指针变量名,

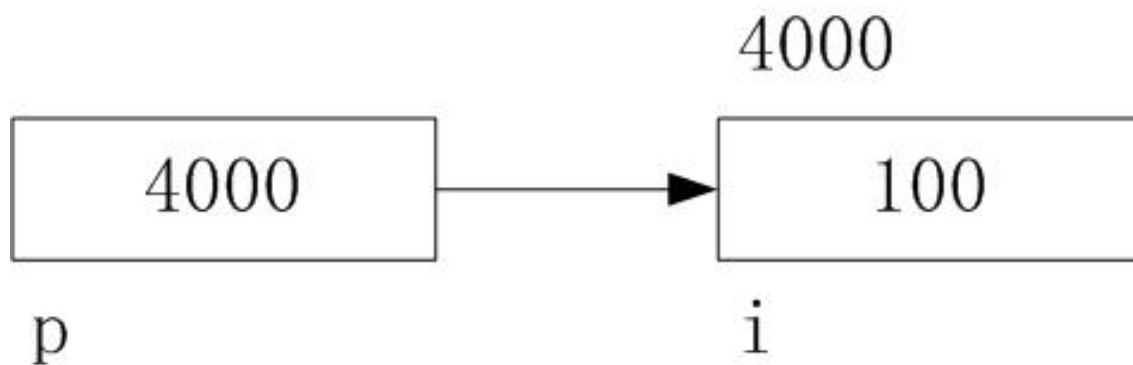
- ▶ 例如：

```
int *p1, *p2; //定义p1和p2为指针变量
int *p, k; //定义p为指针变量，k为整型变量
```

7.1.2 指针变量

通过指针变量，可以间接访问（或间接存取）对象。

图7.2 通过指针变量间接访问

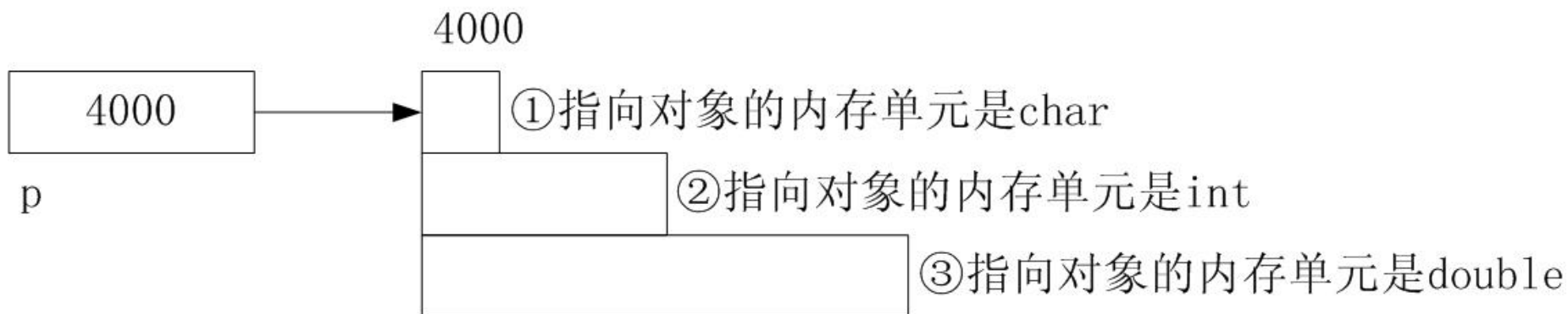


7.1.2 指针变量

假定指针变量p的值是4000，下面三种写法

- ① `char *p;`
- ② `int *p;`
- ③ `double *p;`

图7.3 指向类型的含义



- ▶ 获取变量的地址
- ▶ 可以通过取地址运算（&）获取对象的地址。

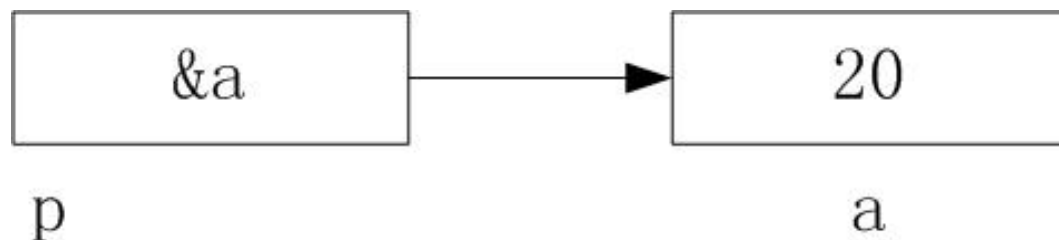
7.2.1 获取对象的地址

表7-1 取地址运算符

运算符	功能	目	结合性	用法
&	取地址	单目	自右向左	&expr

```
int a=20, *p; //定义指针变量  
p = &a ; //指针变量p指向a
```

图7.4 &运算的含义



7.2.1 获取对象的地址

例7.1

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int i=400;
6      cout<<"i="<<i<<" ,&i="<<&i<<endl;
7      return 0;
8  }
```

7.2.1 获取对象的地址

例7.1

```
1  #include <iostream>
2  using namespace std;
3  int main()
```

程序运行屏幕

i=400,&i=0012FF7C



7.2.2 指针的间接访问

- ▶ 指针的间接访问
- ▶ 通过间接引用运算（*）可以访问指针所指向的对象或内存单元

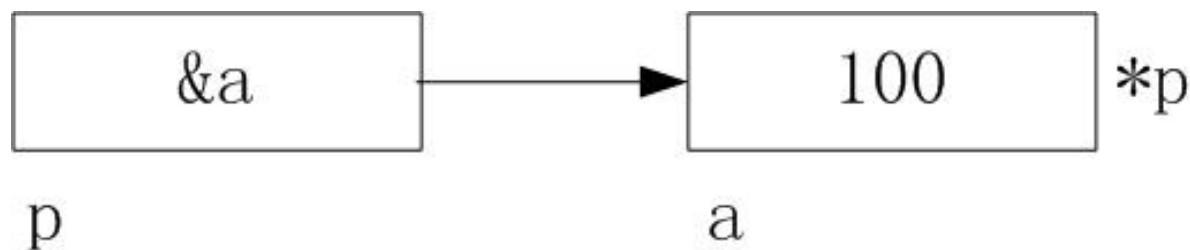
表7-2 间接引用运算符

运算符	功能	目	结合性	用法
*	间接引用	单目	自右向左	*expr

7.2.2 指针的间接访问

```
int a, *p=&a;  
a=100; //直接访问a（对象直接访问）  
*p=100; //*p就是a，间接访问a（指针间接访问）  
*p=*p+1; //等价于a=a+1
```

图7.5 *运算的含义



7.2.2 指针的间接访问

► 【例7.2】 已知：

```
int a, *p=&a;
```

► 则① $\&*p$ 的含义是什么？② $*\&a$ 的含义是什么？

$\&*p \longleftrightarrow p \longleftrightarrow \&a$

$*\&a \longleftrightarrow a \longleftrightarrow *p$

7.2.2 指针的间接访问

例7.3

```
1  int main()
2  {
3      int i=100, j=200;
4      int *p1, *p2;
5      p1=&i, p2=&j; //p1指向i, p2指向j
6      *p1 = *p1 + 1; //等价于i=i+1
7      p1=p2; //将p2的值赋值给p1, 则p1指向j
8      *p1 = *p1 + 1; //等价于j=j+1
9      return 0;
10 }
```

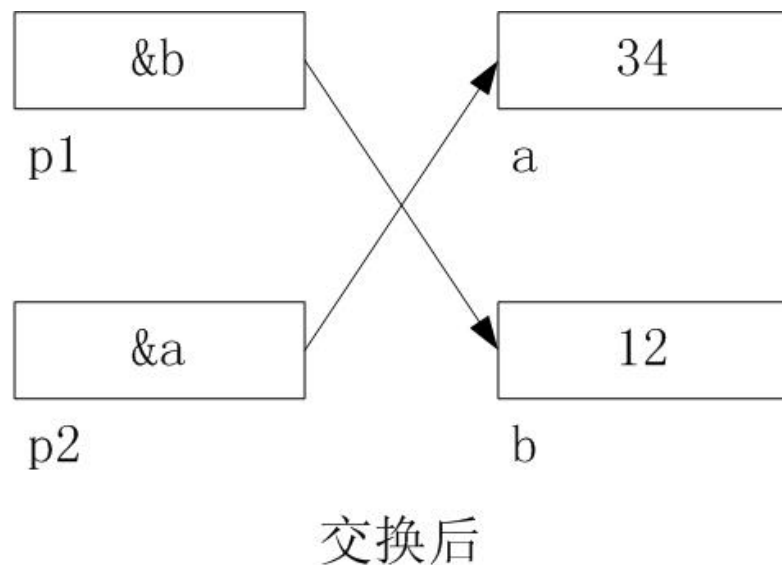
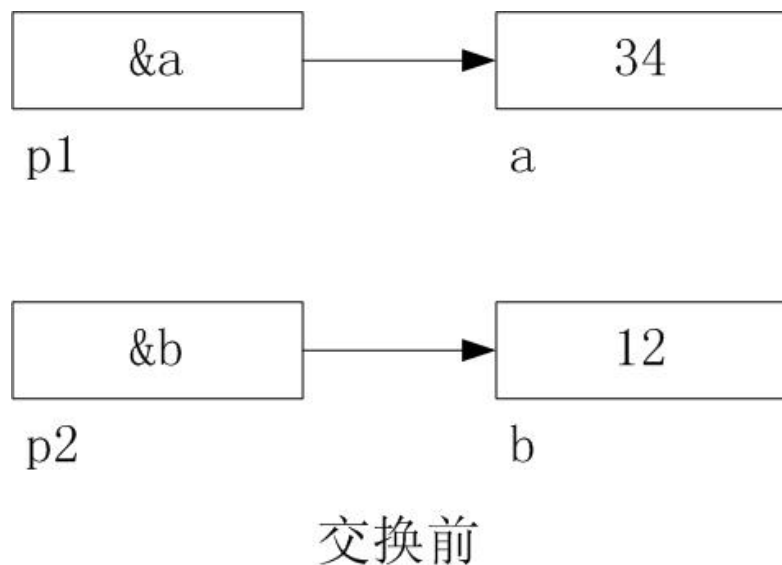

7.2.2 指针的间接访问

例7.4

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a,b,*p,*p1,*p2;
6      p1=&a, p2=&b; //p1指向a, p2指向b
7      cin>>*p1>>*p2;
8      if(*p1>*p2) p=p1, p1=p2, p2=p;
9      cout<<"a="<<a<<" ,b="<<b<<endl;
10     cout<<"min="<<*p1<<" ,max="<<*p2<<endl;
11     return 0;
12 }
```

7.2.2 指针的间接访问

图7.7 指针交换示意



7.2.3 指针变量的初始化与赋值

- ▶ 可以在定义指针变量时对其初始化，一般形式为：

指针类型 *指针变量名=地址初值,

```
int a;  
int *p=&a; //p的初值为变量a的地址  
int b, *p1=&b; //正确, p1初始化时变量b已有地址值
```

7.2.3 指针变量的初始化与赋值

- ▶ 由于指针数据的特殊性，其初始化和赋值运算是有约束条件的，只能使用以下四种值：
- ▶ （1）0值常量表达式，例如：

```
int a, z=0;  
int *p1=a; //错误，地址初值不能是变量  
p1=z; //错误，整型变量不能作为指针，即使值为0  
p1=4000; //错误，整型数据不能作为指针  
p1=null; //正确，指针允许0值常量表达式  
p1=0; //正确，指针允许0值常量表达式
```

- ▶ (2) 相同指向类型的对象的地址。例如：

```
int a, *p1;  
double f, *p3;  
p1=&a; //正确  
p3=&f; //正确  
p1=&f; //错误, p1和&f指向类型不相同
```

- ▶ (3) 相同指向类型的另一个有效指针。例如：

```
int x, *px=&x; //正确
int *py=px; //正确, 相同指向类型的另一个指针
double *pz;
py=px; //正确, 相同指向类型的另一个指针
pz=px; //错误, pz和px指向类型不相同
```

- ▶ (4) 对象存储空间后面下一个有效地址，如数组下一个元素的地址。

- ▶ 指针指向一个有确定存储空间的对象（称为已知对象），则该指针是**有效**的；即如果对该指针使用间接引用运算，总能够得到这个已知对象。
- ▶ 指针理论上可以为任意的地址值，若一个指针不指向程序中任何已知对象，称其指向未知对象。未知对象的指针是**无效**的，无效的指针使用间接引用运算几乎总会导致崩溃性的异常错误。

7.2.4 指针的有效性

- ▶ (1) 如果指针的值为0，称为0值指针，又称空指针（null pointer），空指针是无效的。

```
int *p=0;  
*p=2; //空指针间接引用将导致程序产生严重的异常错误
```

- ▶ (2) 如果指针未经初始化，或者没有赋值，称为野指针，那么该指针是无效的。

```
int *p;  
*p=100; //错误，p为无效指针，不能间接引用
```


7.2.4 指针的有效性

- ▶ 一个指针还没有初始化，称为“**野指针**”（wild pointer），大多数的编译器都对此产生警告。
- ▶ 例如：

```
int *p; //p是野指针
*p=2; //几乎总会导致程序产生严重的异常错误
```

- ▶ 一个指针曾经指向一个已知对象，在对象的内存空间释放后，虽然该指针仍是原来的内存地址，但指针所指已是未知对象，称为“**迷途指针**”（dangling pointer）。例如：

7.2.4 指针的有效性

```
char *p=NULL; //p是空指针, 全局变量
void fun()
{
    char c; //局部变量
    p = &c; //指向局部变量c, 函数调用结束后, c的空间
    释放, p就成了迷途指针
}
void caller()
{
    fun();
    *p=2; //p现在是迷途指针
}
```

- ▶ (1) 指针加减整数运算
- ▶ 设p是一个指针，n是一个整型量，则p+n的结果是一个指针，指向p所指向对象的后面的第n个对象；而p-n的结果是一个指针，指向p所指向对象的前面的第n个对象。

```
int x, n=3, *p=&x;
```

```
p+1 //指向存储空间中变量x后面的第1个int型存储单元
```

```
p+n //指向存储空间中变量x后面的第n(3)个int型存储单元
```

```
p-1 //指向存储空间中变量x前面的第1个int型存储单元
```

```
p-n //指向存储空间中变量x前面的第n(3)个int型存储单元
```

7.2.5 指针运算

例7.5

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10], n=3, *p=&a[0];
6      cout<<*p<<" "<<*(p+1)<<endl;
7          //输出指针p指向的元素及后一个元素
8      cout<<*(p+n)<<endl;
9          //输出指针p指向元素后第n个元素
10     return 0;
11 }
```

- ▶ (2) 指针变量自增自减运算
- ▶ 设p是一个指针变量，其自增自减运算包括p++、++p、p--、--p形式。

```
int a[10], *p=&a[3];  
p++ //运算后p指向a[4]  
++p //运算后p指向a[4]  
p-- //运算后p指向a[2]  
--p //运算后p指向a[2]
```

- ▶ (3) 两个指针相减运算
- ▶ 设 $p1$ 、 $p2$ 是**相同类型**的两个指针（常量或变量），则 $p2-p1$ 的结果为两个指针之间对象的个数，如果 $p2$ 的地址值大于 $p1$ 结果为正，否则为负。
- ▶ (4) 指针的关系运算
- ▶ 设 $p1$ 、 $p2$ 是同一个指向类型的两个指针（常量或变量），则 $p2$ 和 $p1$ 可以进行关系运算，用于比较这两个地址的位置关系。

- ▶ C++ 程序员更偏爱使用指针来访问数组元素，这样做的好处是运行效率高、写法简洁。

7.3.1 指向一维数组元素的指针

- ▶ 1. 一维数组元素的地址
- ▶ 数组由若干个元素组成，每个元素都有相应的地址，通过取地址运算（&）可以得到每个元素的地址。

```
int a[10];  
int *p=&a[0]; //定义指向一维数组元素的指针  
p=&a[5]; //指向a[5]
```


7.3.1 指向一维数组元素的指针

数组可以看作是一个占用更大存储空间的对象，它也有地址。C++规定，**数组名既代表数组本身，又代表整个数组的地址**，还是是数组首元素的地址值，即a与第0个元素的地址&a[0]相同。例如：

```
①p=a;  
②p=&a[0];
```

是等价的。

7.3.1 指向一维数组元素的指针

数组名是一个指针常量，因而它不能出现在左值和某些算术运算中，例如：

```
int a[10], b[10], c[10];
```

```
a=b; //错误，a是常量不能出现在左值的位置
```

```
c=a+b; //错误，a、b是地址值，不允许加法运算
```

```
a++; //错误，a是常量不能使用++运算
```

```
a>b //正确，表示两个地址的比较，而非两个数组内容的比较
```

7.3.1 指向一维数组元素的指针

- ▶ 2. 指向一维数组元素的指针变量
- ▶ 定义指向一维数组元素的指针变量时，指向类型应该与数组元素类型一致，例如：

```
int a[10], *p1;  
double f[10], *p2;  
p1=a; //正确  
p2=f; //正确  
p1=f; //错误，指向类型不同不能赋值
```

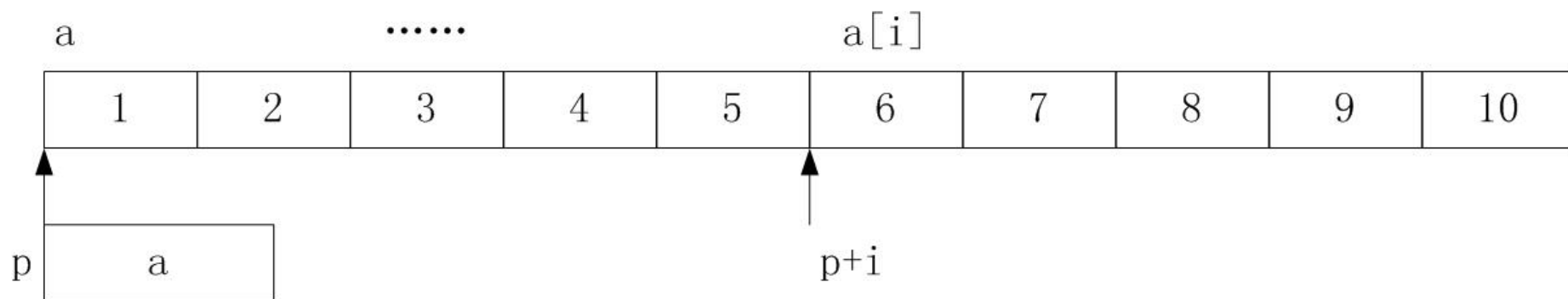
7.3.1 指向一维数组元素的指针

- ▶ 3. 通过指针访问一维数组元素
- ▶ 由于数组元素的地址是规律性增加的，根据指针算术运算规则，可以利用指针及其运算来访问数组元素。
- ▶ 设有如下定义：

```
int *p, a[10]={1,2,3,4,5,6,7,8,9,10};  
p=a; //p指向数组a  
p++;
```

7.3.1 指向一维数组元素的指针

图7.11 指向一维数组的指针



设： a 是一维数组名， p 是指针变量且 $p=a$ 。

根据以上叙述，访问一个数组元素 $a[i]$ ，可以用：

- ①数组下标法： $a[i]$;
- ②指针下标法： $p[i]$;
- ③地址引用法： $*(a+i)$;
- ④指针引用法： $*(p+i)$ 。

7.3.1 指向一维数组元素的指针

例7.8 用多种方法遍历一维数组元素

①下标法。

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10], i;
6      for (i=0;i<10;i++) cin>>a[i];
7      for (i=0;i<10;i++) cout<<a[i]<<" ";
8      return 0;
9  }
```

7.3.1 指向一维数组元素的指针

②通过地址间接访问数组元素。

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10], i;
6      for (i=0;i<10;i++) cin>>*(a+i);
7      for (i=0;i<10;i++) cout<<*(a+i)<<" ";
8      return 0;
9  }
```

7.3.1 指向一维数组元素的指针

③通过指向数组的指针变量间接访问元素。

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10], *p;
6      for (p=a;p<a+10;p++) cin>>*p;
7      for (p=a;p<a+10;p++) cout<<*p<<" ";
8      return 0;
9  }
```


- ▶ 4. 数组元素访问方法的比较
 - ▶ （1）使用下标法访问数组元素，程序写法比较直观，能直接知道访问的是第几个元素。
 - ▶ （2）下标法与地址引用法运行效率相同。而使用指针引用法，指针变量直接指向元素，不必每次都重新计算地址，能提高运行效率。

7.3.1 指向一维数组元素的指针

- ▶ (3) 需要注意指针变量各种运算形式的含义。
- ▶ ① $*p++$ 。
- ▶ ② $*(p++)$ 和 $*(++p)$ 不同。
- ▶ ③ $(*p)++$ 表示p所指向的元素加1。
- ▶ a. $*(p++)$ 等价于 $a[i++]$;
- ▶ b. $*(++p)$ 等价于 $a[++i]$;
- ▶ c. $*(p--)$ 等价于 $a[i--]$;
- ▶ d. $*(--p)$ 等价于 $a[--i]$ 。

- ▶ 可以利用一个字符型的指针处理字符数组和字符串，其过程与通过指针访问数组元素相同。使用指针可以简化字符串的处理，是程序员处理字符串常用的编程方法。

7.4.1 指向字符串的指针

- ▶ 可以定义一个字符数组，用字符串常量初始化它，例如：

```
char str[]="C Language";
```

- ▶ 系统会在内存中创建一个字符数组str，且将字符串常量的内容复制到数组中，并在字符串末尾自动增加一个结束符'\0'。

str

C	␣	L	a	n	g	u	a	g	e	\0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

7.4.1 指向字符串的指针

C++ 允许定义一个字符指针，初始化时指向一个字符串常量，一般形式为：

```
char *字符指针变量=字符串常量, . . . . .
```

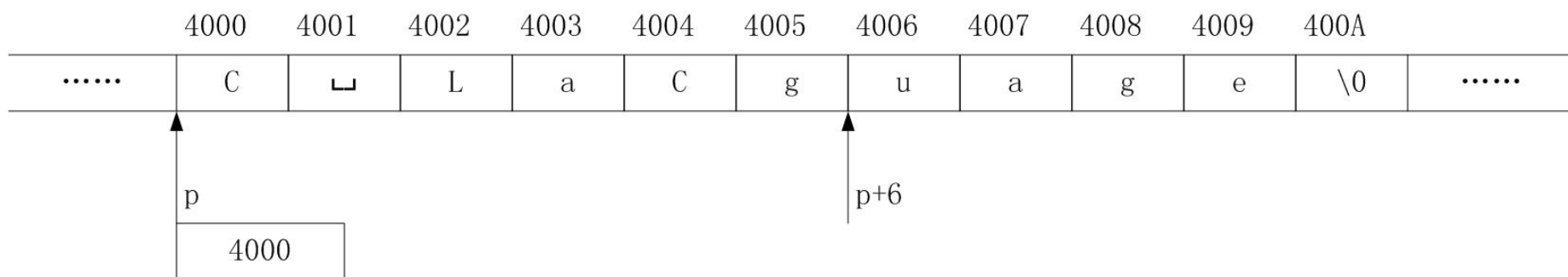
```
char *p="C Language";
```

```
char *p;  
p="C Language";
```

7.4.1 指向字符串的指针

初始化时，p存储了这个字符串首字符地址4000，而不是字符串常量本身，称p指向字符串。

图7.19 指向字符串的指针



7.4.1 指向字符串的指针

通过字符指针可以访问字符串。例如：

```
char str[]="C Language", *p=str; //p指向字符串的  
指针
```

```
cout<<p<<endl; //输出: C Language
```

```
cout<<p+2<<endl; //输出: Language
```

```
cout<<&str[7]<<endl; //输出: age
```

7.4.1 指向字符串的指针

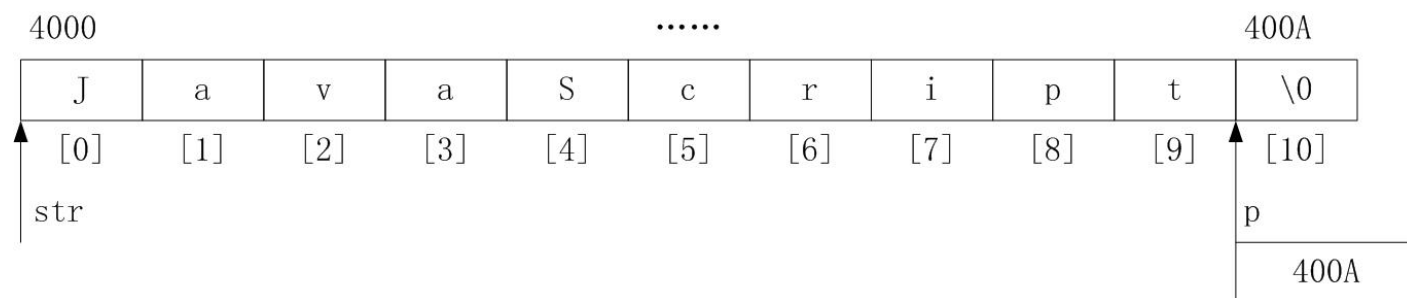
通过字符指针遍历字符串。

```
int n;  
char str[]="C Language", *p=str; //p指向字符串的  
指针  
while (*p!='\0')  
{   if(*p=='a') n++;  
    p++;  
}
```


7.4.1 指向字符串的指针

例7.15

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char str[100], *p=str;
6      cin>>str; //输入字符串
7      while (*p) p++; //指针p指向到字符串结束符
8      cout<<"strlen="<<p-str<<endl;
9      return 0;
10 }
```



7.4.1 指向字符串的指针

例7.15

```
1  #include <iostream>
2  using namespace std;
3  int main()
```

程序运行屏幕

strlen=10

JavaScript ↙

7.4.2 指针与字符数组的比较

```
char s[100]="Computer";  
char *p="Computer";
```

- ▶ 1. 存储内容不同
- ▶ 2. 运算方式不同
- ▶ 3. 赋值操作不同

```
s++; //错误
```

```
p++; //正确
```

```
s="C++"; //错误
```

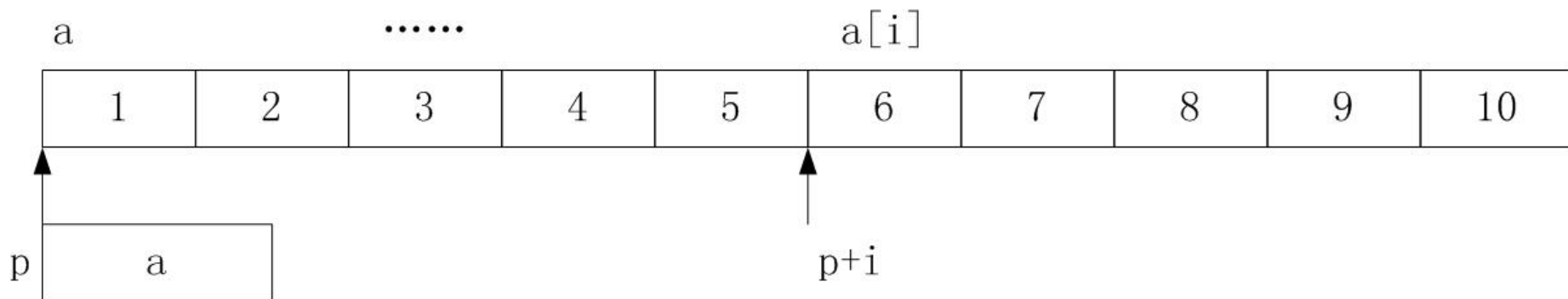
```
p="C++"; //正确
```

复习：

- ▶ 指针的定义：专门用来存放地址的变量
- ▶ 指针的类型：`int *p;` `double *q;` `char *t;`
- ▶ 间接引用：`int a, *p=&a; *p=100;`

复习:

- ▶ C++程序员更偏爱使用指针来访问数组元素，这样做的好处是运行效率高、写法简洁。



设：`a`是一维数组名，`p`是指针变量且`p=a`。

数组元素`a[i]` 可以表示为：`p[i]`、`*(a+i)`、`*(p+i)`。

`a[i]`的地址可以表示为：`&a[i]`、`a+i`、`p+i`

复习:

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10], i;
6      for (i=0;i<10;i++) cin>>a[i];
7      for (i=0;i<10;i++) cin>>*(a+i);
8      return 0;          cout<<*(a+i)<<" ";
9  }
```

复习:

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10], *p;
6      for (p=a;p<a+10;p++) cin>>*p;
7      for (p=a;p<a+10;p++) cout<<*p<<" ";
8      return 0;
9  }
```

复习:

- ▶ 使用字符指针处理字符串可以简化字符串的处理，是程序员处理字符串常用的编程方法。

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char *p="VisualBasic";
6      int i=0;
7      while (p[i]) cout<<p[i++];
8      return 0;
9  }
```


7.4.3 指向字符串数组的指针

字符串数组是一个二维字符数组，例如：

```
char sa[6][7]={"C++","Java","C","PHP",  
               "CSharp","Basic"};
```

sa

sa[0]	C	+	+	\0			
sa[1]	J	a	v	a	\0		
sa[2]	C	\0					
sa[3]	P	H	P	\0			
sa[4]	C	S	h	a	r	p	\0
sa[5]	B	a	s	i	c	\0	

7.4.3 指向字符串数组的指针

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char sa[6][7]={"C++","Java","C","PHP",
6                    "CSharp","Basic"};
7      int i;
8      for(i=0;i<6;i++) cout<<sa[i];
9      return 0;
10 }
```

- ▶ 指针最重要的应用是作为函数参数，它使得被调函数除了返回值之外，能够将更多的运算结果返回到主调函数中
- ▶ 指针是函数参数传递的重要工具。

7.5.1 指针作为函数参数

- ▶ 1. 指针变量作为函数形参
- ▶ 函数形参可以是指针类型，一般形式为：

```
返回类型 函数名(指向类型 *指针变量名, ...)  
{  
    函数体  
}
```

7.5.1 指针作为函数参数

例7.18

```
1  #include <iostream>
2  using namespace std;
3  void swap(int *p1, int *p2)
4  {
5      int t;
6      t=*p1 , *p1=*p2, *p2=t;
7  }
8  int main()
9  {
10     int a, b;
11     cin>>a>>b;
12     swap(&a, &b);
13     cout<<"min="<<a<<" ,max="<<b;
14     return 0;
15 }
```

7.5.1 指针作为函数参数

例7.18 以下方式无法交换两个变量的值

```
3  void swap(int *p1, int *p2)
4  {
5  int *t;
6  t=p1 , p1=p2, p2=t;
7  }
8  int main()
9  {
10     int a, b;
11     cin>>a>>b;
12     swap(&a, &b);
13     return 0;
14 }
```

- ▶ 通过将指针作为函数参数的方法，既可以返回多个运算结果，又避免了使用全局变量。

7.5.1 指针作为函数参数

例7.19 计算a和b的平方和、自然对数和、几何平均数、和的平方根

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  double fun(double a,double b,double *sqab,
              double *lnab, double *avg)
5  {
6      *sqab=a*a+b*b;  *lnab=log(a)+log(b);
7      *avg=(a+b)/2;
8      return (sqrt(a+b)); //函数返回和的平方根
9  }
10
11 int main()
12 {
13     double x=10,y=12,fsq,fln,favg,fsqr;
14     fsqr=fun(x, y, &fsq, &fln, &favg);
15     cout<<x<<" "<<y<<" "<<fsq<<" "<<fln
16         <<" "<<favg<<" "<<fsqr<<endl;
17     return 0;
18 }
```


7.5.1 指针作为函数参数

- ▶ 2. 数组作为函数形参
- ▶ （一维或多维）数组作为函数的形参，例如：

```
double average(double *a, int n)
{
    ... //函数体
}
```

- ▶ 函数调用形式如下：

```
double X[100], f;
f = average(X, 100);
```

7.5.1 指针作为函数参数

例7.20 编写函数average，返回数组n个元素的平均值。

```
1  #include <iostream>
2  using namespace std;
3  double average(double *a, int n)
4  {
5      double avg=0.0, *p=0;
6      for (p=a;p<a+n;p++) avg=avg + *p;
7      return n<=0 ? 0 : avg/n ;
8  }
10 int main()
11 {
12     double x[10]={66,76.5,89,100,71.5,
13                  86,92,90.5,78,88};
14     cout<<"average="<<average(x,10)<<endl;
15     return 0;
16 }
```

- ▶ 4. 字符指针变量作为函数形参
- ▶ 将一个字符串传递到函数中，传递的是地址，则函数形参既可以用字符数组，又可以用指针变量，两种形式完全等价。在子函数中可以修改字符串的内容，主调函数得到的是变化后的字符串。

7.5.1 指针作为函数参数

例7.21 自定义函数实现strcpy函数的字符串复制功能

```
1  #include <iostream>
2  using namespace std;
3  char *stringcpy(char *strDest,
                   const char *strSrc)
4  {
5      char *p1=strDest;
6      const char *p2=strSrc;
7      while (*p2!='\0')
8          *p1=*p2, p1++, p2++;
9      *p1='\0';
10     return strDest; //返回实参指针
11 }
```

7.5.1 指针作为函数参数

例7.21

```
11  int main()
12  {
13      char s1[80], s2[80], s3[80]="string=";
14      cin>>s1; //输入字符串
15      strcpy(s2, s1); //复制s1到s2
16      cout<<"s2:"<<s2<<endl;
17      strcpy(&s3[7], s1); //复制s1到s3的后面
18      cout<<"s3:"<<s3<<endl;
19      return 0;
20  }
```

7.5.3 函数指针

- ▶ C++允许定义指向函数的指针变量，定义形式为：

返回类型 (*函数指针变量名)(形式参数列表), ...;

- ▶ 它可以指向形如

```
返回类型 函数名(形式参数列表)
{
    函数体
}
```

```
int (*p)(int a, int b); //定义函数指针变量
```

- ▶ 1. 指向函数
- ▶ 可以将函数的地址赋值给函数指针变量，形式为

函数指针变量=函数名；

- ▶ 它要求函数指针变量与指向函数必须有相同的返回类型、参数个数、参数类型。

7.5.3 函数指针

例如假设：

```
int max(int a, int b); //max函数原型  
int min(int a, int b); //min函数原型  
int (*p)(int a, int b); //定义函数指针变量
```

则

```
p=max;
```

称p指向函数max。它也可以指向函数min，即可以指向所有与它有相同的返回类型、参数个数、参数类型的函数。

- ▶ 2. 通过函数指针调用函数
- ▶ 对函数指针间接引用即是通过函数指针调用函数，一般形式为：

- ①(*函数指针)(实参列表)
- ②函数指针(实参列表)

- ▶ 两种形式是完全相同的。通常，程序员偏爱用第②种形式。
- ▶ 通过函数指针调用函数，在实参、参数传递、返回值等方面与函数名调用相同。例如：

```
c=p(a,b); //等价于c=max(a,b);
```

7.5.3 函数指针

例7.23

```
1  #include <iostream>
2  using namespace std;
3  int max(int a, int b) //求最大值
4  {
5      return a>b ? a:b ;
6  }
7  int min(int a, int b) //求最小值
8  {
9      return a<b ? a:b ;
10 }
```

7.5.3 函数指针

例7.23

```
11  int main()
12  {
13      int (*p)(int a,int b); //定义函数指针变量
14      p=max; //p指向max函数
15      cout<<p(3,4)<<" "; //通过p调用函数
16      p=min; //p指向min函数
17      cout<<p(3,4)<<" "; //通过p调用函数
18      return 0;
19  }
```

从中看出，函数调用p(3,4)究竟调用max或者min，取决于调用前p指向哪个函数。

▶ 3. 函数指针的用途

- ▶ 指向函数的指针多用于指向不同的函数，从而可以利用指针变量调用不同函数，相当于将函数调用由静态方式（固定地调用指定函数）变为动态方式（调用哪个函数是由指针值来确定）。

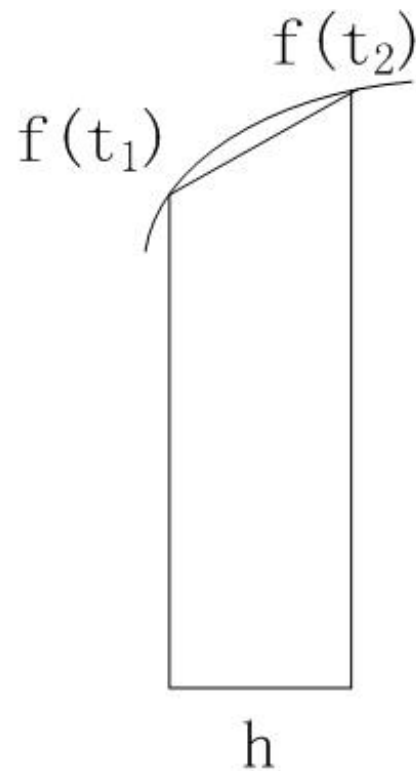
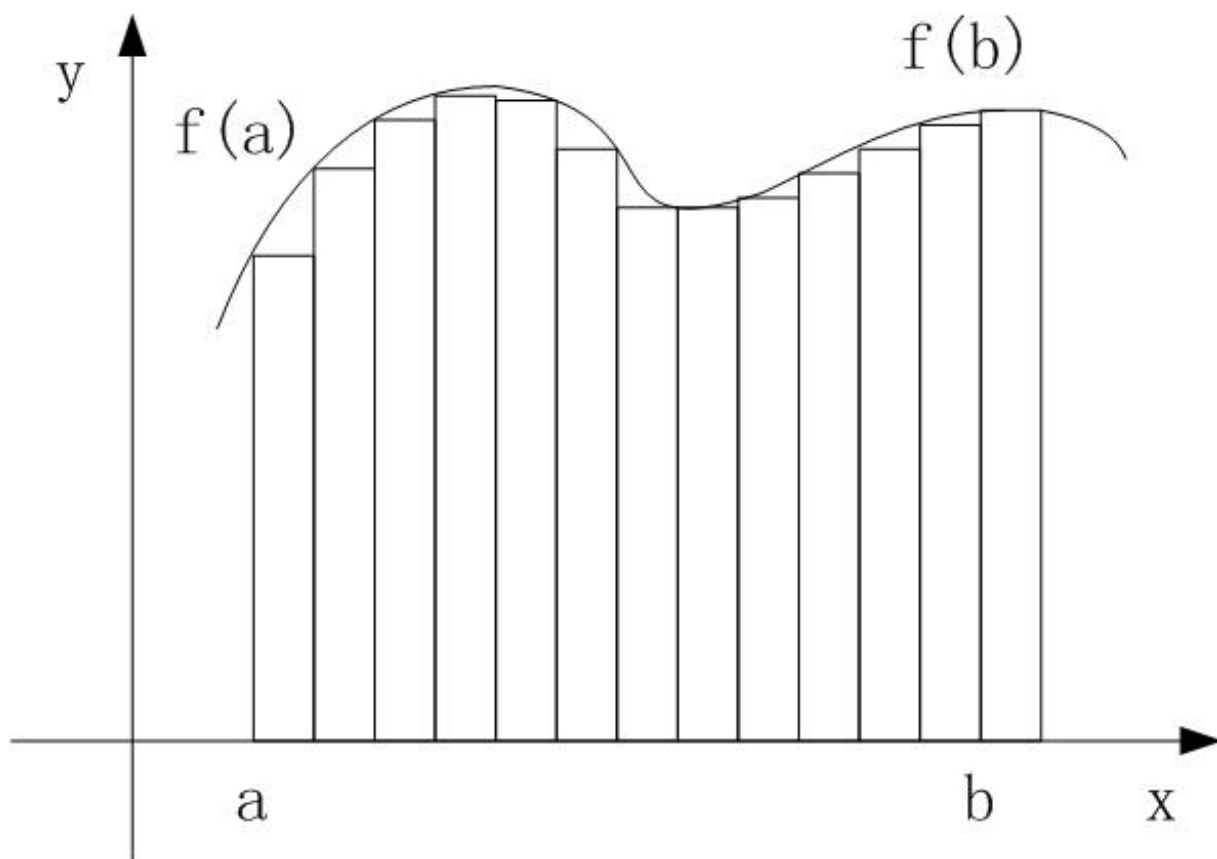
▶ 【例7. 24】 编写程序计算如下公式。

▶ 说明：

▶ 这里用梯形法求定积分 $\int_a^b f(x)dx$ 的近似值。
如图所示，求 $f(x)$ 的定积分就是求 $f(x)$ 曲线与 x 轴包围图形的面积，梯形法是把所要求的面积垂直分成 n 个小梯形，然后面积求和。

7.5.3 函数指针

图7.24 梯形法求定积分示意



7.5.3 函数指针

根据上述思想编写函数integral，由于需要计算多个不同 $f(x)$ 的值，因此向integral传递 $f(x)$ 的函数指针，由integral回调具体的 $f(x)$ 求值，其函数原型为：

```
double integral(double a, double b,  
                double (*f)(double x)) //求定积分
```

在integral中通过函数指针f调用传进去的具体函数，即称为回调。

7.5.3 函数指针

例7.24

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  double integral(double a, double b,
5                  double (*f)(double x)) //求定积分
6  {
7      int n=1000, i;
8      double h, x, s=0.0;
9      h=(b-a)/n;
10     for(i=1; i<=n; i++) {
11         x=a+(i-1)*h;
12         s=s+(f(x)+f(x+h))*h/2;
13     }
14     return s;
```


7.5.3 函数指针

例7.24

```
14  }
15  double f1(double x)
16  { return 1+x;
17  }
18  double f2(double x)
19  { return exp(-x*x/2);
20  }
21  double f3(double x)
22  { return x*x*x;
23  }
```

7.5.3 函数指针

例7.24

```
24  int main()
25  {
26      double a,b;
27      cin>>a>>b;
28      cout<<(integral(a,b,f1)+
                integral(a,b,f2)+integral(a,b,f3))
                <<endl;
29      return 0;
30  }
```

- ▶ 通过对象名称直接访问对象，优点是直观，操作哪个对象一目了然，缺点是由于对象名存在作用域的限制，某些情况下是不能按名称访问对象的，例如一个函数内部不能使用另一个函数的局部变量；
- ▶ 通过指针（或地址）间接访问对象，优点是无所不能，缺点是程序中大量出现的间接访问，实在分不清具体是哪个对象，需要通过上下文去分析。
- ▶ C++扩充了C语言对象访问方式，提供了引用访问。通过引用访问对象，结合了按名访问和按地址访问各自的优点，非常适合作为函数参数。

7.8.1 引用的概念与定义

- ▶ 简单地说，引用（reference）就是一个对象的别名（alias name），其声明形式为：

引用类型 **&引用名称=对象名称** , ;

```
int x; //定义整型变量x  
int &r=x; //声明r是x的引用
```

- ▶ 在C++中，引用全部是const类型，声明之后不可更改（即不能再是别的对象的引用）。

- ▶ 1. 引用的规则
- ▶ (1) 声明一个引用类型变量时，必须同时初始化它，声明它是哪个对象的别名，即绑定对象。例如：

```
int &r; //错误，引用是const类型，必须在声明时初始化  
int x, &r=x; //正确 声明r是x的引用
```

- ▶ (2) 不能有空引用，引用必须与有效对象的内存单元关联。

- ▶ (3) 引用一旦被初始化，就不能改变引用关系，不能再作为其他对象的引用。例如：

```
int x, y; //定义整型变量x,y
int &r=x; //正确 声明r是x的引用
int &r=y; //错误 r不能再是别的对象的引用
```

- ▶ (4) 指定类型的引用不能初始化到其他类型的对象上，例如：

```
double f; //定义浮点型变量f
int &r=f; //错误 r值整型的引用，不能绑定到浮点型的对象上
```

- ▶ (5) 引用初始化与对引用赋值含义完全不同，例如：

```
int x; //定义整型变量x
int &r=x; //初始化 指明r是x的引用，即将r绑定到x
r=100; //引用赋值 100赋值到r绑定的内存单元中（即x）
```

- ▶ (6) 取一个引用的地址和取一个对象的地址完全一样，都是用取地址运算。例如：

```
int x; //定义整型变量x,y
int &r=x; //声明r是x的引用
int *p1=&x; //p1指向x
int *p2=&r; //p2指向r，本质上指向x
```


▶ 2. 引用作为函数形参

```
1  #include <iostream>
2  using namespace std;
3  //引用作为函数形参
4  void swap(int &a,int
5  {    int t;
6      t=a, a=b, b=t;
7  }
8  int main()
9  {    int x=10, y=20;
10     swap(x,y);
11     cout<<x<<" "<<y;
12     return 0;
13 }
```

```
1  #include <iostream>
2  using namespace std;
3  //引用作为函数形参
4  void swap(int &x,int
&y)
5  {    int t;
6      t=x, x=y, y=t;
7  }
8  int main()
9  {    int x=10, y=20;
10     swap(x,y);
11     cout<<x<<" "<<y;
12     return 0;
13 }
```

7.8.2 引用的使用

显然，函数引用传递方式也可以实现多个数据结果返回到主调函数中，其功能与指针方式相同。但指针方式返回数据结果必须：

- ①实参为地址，即进行“&”取地址运算；
- ②形参分配指针变量接受实参地址；
- ③函数内部使用指针间接访问，即进行“*”间接访问运算。而引用传递方式把这个过程简化了。

使用引用作为函数形参，比使用指针变量简单、直观、方便，特别是避免了在被调函数中出现大量指针间接访问时，所指对象究竟是哪个具体对象伤脑筋的问题，从而降低了编程的难度。

- ▶ 3. 引用作为函数返回值
- ▶ 函数的返回值可以是引用类型，即函数返回引用，其定义形式为：

引用类型& 函数名(形式参数列表)

{

函数体

}

7.8.2 引用的使用

例7.29

```
1  //函数返回引用
2  #include <iostream>
3  using namespace std;
4  int& max(int &a,int &b)
5  { return (a>b? a:b); }
6  int main()
7  {   int x=10,y=20,z;
8      z = max(x,y);
9      cout << z;
10     return 0;
11 }
```

7.8.2 引用的使用

可以看出，函数返回引用与函数返回值有重大区别，它不是返回一个临时对象，而是相当于返回实体对象本身。正因为如此，函数返回引用可以作为左值。例如：

```
int& fun(int &a,int &b)
{ return (a>b? a:b); }
int x=10,y=20,z=5;
fun(x,y)=z; //调用fun函数后相当于y=z;
cout << y;
```

CP[®]程序设计