C++语言培训教材

（中册）提高篇

● 刘峻桐　刘　豫　著

URL:http://www.phei.co.cn

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
電子工業出版社

C++语言培训教材（中册）提高篇
電子

C++语言培训教材

（中册） 提高篇

刘峻桐 刘 豫 编著
高永泉 贾宝金 审校

電子工業出版社

内容简介

全书分上、中、下三册．上册为基础篇，讲述了C＋＋语言的特点及优越性和C＋＋程序设计的风格，C＋＋的操作符、语句、函数、数组、变量作用域、模块化程序设计及指针等。读者通过上册的学习，基本上可以掌握C＋＋语言的基础。

中册为提高篇。进一步对指针和数组进行深入的研究，进而提出类、对象以及继承性，并围绕类和对象进一步阐述了C＋＋语言面向对象的特点。随后又对构造函数、析构函数、虚函数以及函数重载等作了详细的讲述，培养读者用类模拟解决实际问题的能力，掌握面向对象的程序设计方法。

下册为应用篇。重点讲述C＋＋语言的输入输出(I/O)及介绍4个应用例程，通过解剖分析这4个C＋＋实用程序，培养读者运用所学的C＋＋语言编写程序时解决实际问题的能力。

本书适用于初中级教材，也可作为有关的训练和自学教材。

C＋＋语言培训教材
（中册） 提高篇
刘峻桐 刘 豫 编著
高永泉 贾宝金 审校
责任编辑：毛兆余
特约编辑：张成全
*
电子工业出版社出版
北京市海淀区万寿路173号信箱(100036)
电子工业出版社发行 各地新华书店经销
北京科技大学印刷厂印刷
*
开本：787×1092毫米 1/16 印张：10.5 字数：253千字
1996年12月第一版 1996年12月第一次印刷
印数：5000册 定价：13.00元
ISBN 7-5053-3921-4/TP・1697

前　言

C++代表着程序设计语言领域的一个重要进步。目前，学习C++程序设计语言的人越来越多，许多以前用C语言编程的人员现在开始转向用C++。C++代表了C语言的发展趋势。C++采用了许多重要技术，这使它替代C语言只是一个时间问题了。

C++是C语言的增强型版本。在C语言的基础上，C++做了功能扩充以支持面向对象的程序设计。这些扩充极大地增强了C语言的能力。C++语言的强大功能及其通用性，决定了它必将成为继C之后的主要程序设计语言之一。

事实上，C语言在应用上所形成的广泛基础已经注定了C++语言在未来程序设计语言中的主导地位。从本质上讲，C++与C有着不同的编程风格。学习C++语言，不仅要学习语言要素本身，而更重要的是学会如何用C++的思维方式进行程序设计。为了使读者能够尽早地养成用C++思维方式来考虑问题的习惯，本书力求一开始即给读者展现C++的编程风格，而不是传统的C语言风格。

在语言学习中，单调的语法讲解不仅无助于有经验的程序设计人员提高编程技巧和能力，而且往往会使初学者感到迷惑。因此，本书在讲解C++语言要素的同时，结合大量精短的程序，以帮助读者理解和掌握C++语言。在阅读本书时读者将会发现，用这种方法讲解语言，不仅可以使你非常容易地掌握C++语言要素，与此同时还能学到许多编程技巧。

本书分上、中、下三册。上册为基础篇，基础篇共十七章，首先讲述了由C语言演变为C++语言的发展过程、C++语言的特点以及与C语言的主要区别，阐述了C++语言的优越性。进而讲述了C++程序设计的风格及C++的操作符、语句、函数、数组、变量作用域、模块化程序设计及指针等。因此可以说上册是学习C++语言的基础。读者通过上册的学习，基本上可以掌握C++语言的基础。

中册为提高篇，共九章（第十八章至第二十六章），是在上册的基础上进一步对指针和数组进行了深入的研究讨论，进而提出类、对象以及其继承性，并围绕类和对象进一步阐述了C++语言面向对象的特点，随后又对构造函数、析构函数、虚函数以及函数重载等作了详细的讲述，以培养读者用类模拟解决实际问题的能力，进而掌握面向对象的程序设计方法。

下册为应用篇，由本书的第二十七章至第三十一章以及两个附录组成。重点讲述C++语言的输入输出(I/O)及介绍4个应用例程，并通过举例解剖分析4个C++实用程序，来培养读者运用所学的C++语言解决实际编程问题的能力。

本书在编写过程中，高永泉、贾宝金、王桂兰、李桂萍等给予了大力支持，并做了大量工作，在此表示衷心的感谢。但因作者水平有限，虽然从事编程多年，亦恐书中仍有错误不妥之处，欢迎读者批评指正。

目　录

第十八章　指针和数组的深入讨论

在C++语言中，指针和数组有很密切的关系，密切到使指针和数组确实应该同时加以处理。任何可通过数组下标完成的操作也可通过指针完成，同样，可以象处理数组那样对指针进行寻址。指针操作在C++编程语言中非常重要，本章介绍的内容将广泛地用于C++编程中。

本章主要介绍如下内容：

- 数组名和指针
- 指针的优越性
- 地址运算
- 使用字符指针
- 字符型指针数组
- 指针的指针
- 指向函数的指针

18.1　数组名和指针

数组名即是指针。为了证实这一点，我们首先定义一个数组和一个指针：

```
int a[5]={20,30,40,50,60};
int * pa;
```

下列语句把数组a的第一个元素的地址赋给指针pa：

```
pa=& a[0];
```

这样做是很容易理解的。下面我们通过引用数据操作符*和指针打印出数组的第一个元素的值：

```
cout<< * pa;
```

很显然，打印的结果是20。

前面我们讲过，数组在映射到内存中时，数组名就是一个指针，它指向数组的第一个元素。既然是这样，能否直接用*a打印出第一个元素的值呢？答案是肯定的。下面语句同样能打印出数组a中的第一个元素的值：

```
cout<< *a;
```

由此进一步往前推，我们可以不使用下标，打印出数组各元素的值：

```
cout<< *(a+1);        //打印出第二个元素值;
```

```
cout<<*(a+2);        //打印出第三个元素值;
cout<<*(a+3);        //打印出第四个元素值;
cout<<*(a+4);        //打印出第五个元素值;
```

由于数组名是指向数组第一个元素的指针，因此，可以用引用来得到数组各个元素的值。

18.2 指针的优越性

必须记住，尽管数组名实际上就是指针，是特殊类型的指针，但它是指针常量，而不是指针变量。

常量通常是不能改变的，比如，下述语句是不正确的，因为不能给一个常量 6 赋另一个值：

```
6=7+8;    //错误的赋值
```

由于数组名是一个指针常量，因此，在使用时也不能试图改变存在其中的内容。这也就是为什么在程序执行时不能为数组名赋值的原因。

下面声明了一个字符数组：

```
char str[30];          //声明一个数组。
```

那么，下述语句是非法的：

```
str="chinese";         //非法赋值。
```

当声明一个数组后，C++便知道数组名是一个指针常量，因此再给一个常量赋值，会导致错误。不能试图改变数组名中存放的内容。

然而，除了数组名引伸过来的指针外，指针是变量，因而指针在使用时，可以改变其中的值，这就是指针和数组名相比较所具有的主要不同点；同时也是为什么在数组之外还必须学习指针的主要原因。也正是因为指针的这一优点，而实际上可用它来处理包括数组在内的所有数据。

可以给同一个指针赋不同的值，从而使它指向不同的变量。下面程序中定义了两个字符型数组和一个字符型指针。随后通过给指针赋不同的值，使它先后指向第一个数组的第 2 个元素和第二个数组的第 2 个元素。

```
#include<iostream.h>
void main()
{
    char str1[ ]={A,B,C,D,E};
    char str2[ ]={F,G,H,I};
    char * ptr;
    ptr=& str1[1];
    cout<<"The first value is"<<*ptr<<"\n";
    ptr=&str2[1];
    cout<<"The second value is "<<*ptr<<"\n";
```

```
    rerurn;
}
```

必须记住数组名和指针之间的区别。指针是变量,因此下述对指针的操作都是有意义的:

```
ptr=str1;
ptr++;
```

而由于数组名是常量,不是变量,因此,下述对数组名的操作都是非法的:

```
str1=ptr;
str1++;
str1= &str2[1];
```

在实际使用中,由于数组容易声明,而指针可被改变,因此许多C++程序员都喜欢结合使用指针和数组。通常是先声明一个数组,然后用指针来引用该数组的值,如果数组数据改变,指针则可帮助改变它。

下面程序首先声明并初始化一个整型数组,然后声明一个指针指向该数组,紧接着用两个 for 循环,分别用下标和指针打印出数组中的内容。

```
#include <iostream.h>
void main()
{
    int ctr;
    int iara[5] = { 10, 20, 30, 40, 50};
    int *iptr;
    iptr = iara;
    cout << "Using array subscripts:\n";
    cout << "iara\tiptr\n";
    for (ctr = 0; ctr < 5; ctr++)
    {
      cout << iara[ctr] << "\t" << iptr[ctr] << "\n";
    }
    cout << "\nUsing pointer notation:\n";
    cout << "iara\tiptr\n";
    for (ctr = 0; ctr < 5; ctr++)
    {
      cout << *(iara) << "\t" << *(iptr+ctr) << "\n";
      iara++;
    }
  return;
}
```

18.3 地址运算

若 P 是一个指针,则 P++使 P 指向下一个元素,P+=i 使其指向 P 当前所指向元素

后面的第 i 个元素。这些操作是指针运算最简单和最普遍的形式。

然而，当给指针做增量或减量运算时，指针中存放的地址每次并非一定是增加或减少 1。

假如 f_ptr 是一个浮点型指针，用下述语句初始化该指针：

```
float ara[]={11.5,21.45,329.79,423.5,698.45};
f_ptr=ara;
```

此时 f_ptr 指向数组 ara 的第一个元素，该元素中存放的数据是 11.5。当我们给该指针作增量运算或加 1 时：

```
f_ptr++;
或 f_ptr+=1;
```

经过增量运算或加 1 后的指针 f_ptr 指向数组的第 2 个元素。

而我们知道，浮点数在机器中所占内存单元的个数因机器而异，多数情况下是 4 个字节的内存。如果假定浮点数在机器上占 4 个字节的内存，在这种情况下，要使指针 f_ptr 从指向浮点数组的第一个元素变为指向浮点数组的第二个元素，指针 f_ptr 中的地址需增加 4。

由此可见，指针增加或减少操作时，指针中存放的地址并不是简单地加减操作，而是根据指针类型(所指向变量的类型)，以及这种类型的数据在机器中所占的字节数作相应尺寸的加减操作。这正是为什么 C++要求在声明指针时，必须采用正确的数据类型，以及一种类型的指针只能指向同种类型的变量的原因。

上面初始化的指针 f_ptr，当给它增加 1 后，打印 * f_ptr，则结果将是数组 ara 的第二个元素中的值 21.45。

因此我们说，当给指针加 1 时，C++给该指针增加一个相应的数据类型在内存中所占的字节的个数值，而不是一个字节。同样，当给指针减 1 时，C++把指针中存放的地址，减去相应的数据类型所占内存的字节数。

注意，对指针的操作仅限于指针与整数相加减，两个同类型指针相减或比较，除此之外，其它任何指针运算都是非法的。不允许两个指针相加、相乘、相除、移位或屏蔽它们，也不允许把指针和 float 或 double 型数相加。

下述程序中定义了一个浮点数组和一个浮点型指针。认真分析程序中每个语句的作用，写出程序的运行结果

```
#include <iostream.h>
void main()
{
    float ara[] = { 100.0, 200.0, 300.0, 400.0, 500.0};
     float *fptr;
    fptr = &ara[0];
    cout << *fptr << "\n";
    fptr++;
    cout << *fptr << "\n";
    fptr++;
```

```
    cout << *fptr << "\n";
    fptr++;
    cout << *fptr << "\n";
    fptr++;
    cout << *fptr << "\n";
    fptr = ara;
    cout << *(fptr+2) << "\n";
    cout << (fptr+0)[0] << " " << (ara+0)[0] << "\n";
    cout << (fptr+1)[0] << " " << (ara+1)[0] << "\n";
    cout << (fptr+4)[0] << " " << (ara+4)[0] << "\n";
    return;
}
```

指针的减法也是合法的，如果 p 和 q 指向同一数组，则 p－q 是 p 和 q 之间元素的数目。下述计算字符串长度的函数中利用了指针减法运算：

```
int strlen(char * s)
{
char * p=s;
while (*p! ='\0')
p++;
return(p-s);
}
```

C++处理指针(地址)运算的方法是连贯和有规律的。指针、数组和地址的综合处理，是 C++语言的主要优点之一。下面我们再用两个函数的定义，说明指针(地址)运算的特性。

下面程序段中包含两个函数，其中函数 alloc()返回一个指向 n 个连续字符单元的指针。alloc()函数的调用者可以使用该指针存储字符。函数 free()释放已请求到的存储空间以备后面再用。

实现 alloc()函数最简单的方法是让 alloc 从一个大的字符数组中分配出一部分。我们称这个数组为 allocbuf，是 alloc()和 free()的局部变量。

因为函数是以指针而不是以下标处理内存分配和释放的，所以其它子程序不需要知道这个数组的名字。这个数组可以被说明成外部静态变量，即对含有 alloc()和 free()的源文件来说是局部的，而在该文件之外是不可见的。实际实现时，数组甚至可以没有名字，而是通过向操作系统请求一个指向某个未被命名的存贮块的指针来得到。

图 18.1 显示了字符数组 allocbuf 在调用 allloc()函数分配出一段内存前和分配后的情况：

程序中指针 allocp 指向下一个可用元素。当用 alloc()请求分配 n 个字符单元时，它要检查 allocbuf 中是否有足够的空间。若有，alloc()返回 allocp 的当前位置(即可用块的开始位置)然后使 allocp 加 n 指向下一可用区。

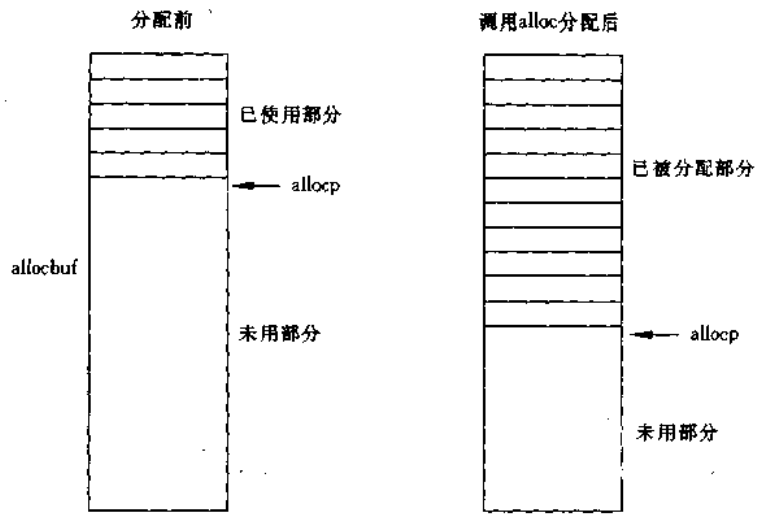

图 18.1

```
# define ALLOCSIZE 10000
static char allocbuf[ALLOCSIZE];
static char * allocp=allocbuf;
char * alloc(int n)
{
    if (allocbuf+ALLOCSIZE-allocp>=n)
    {
      allocp+=n;
      return allocp-n;
    }
    else cout <<"unencagh space";
      return 0;
}

void afree (char * p)
{
    if (p>=allocbuf && p<allocbuf+ALLOCSIZE)
    allocp=p;
}
```

从上述程序中可以看出，首先用数组名初始化指针 allocp，使它指向数组第一个元素的位置：

其实，也可用下述语句初始化指针 allocp：

```
static char * allocp=& allocbuf[0];
```

对于前面我们讲的指针更容易变化，等看完下述内容后，我们会对此有更深刻的理解。

上述两个函数的代码虽然很短，但其中用到了几乎所有允许的指针运算：

①其中包括指针与整数相加减：

```
allocp+=n;
return (allocp-n);
```

②两指针比较：

```
p>=allocbuf && p<allocbuf+ALLOCSIZE
allocbuf+ALLOCSIZE-allocp>=n
```

③把数组名用作指针常量：

```
allocp=allocbuf;
allocp+ALLOCSIZE>=n+allocp
p>=allocbuf && p<allocbuf+ALLOCSIZE
```

④两指针相减：

```
allocbuf-allocp>=n-ALLCOSIZE
```

对于指针和整数相加减，比较容易理解。而对于两指针相减和进行比较，需作进一步说明。

在特定环境中，指针可以比较。若 p 和 q 是指向同一数组的元素的指针，则关系运算<，>=等都可正常使用。

```
如:p>q;
```

当 p 所指的元素位于 q 所指的元素后面时为真。在这种情况下，关系运算符==和!=也可以使用。另外，指针都可以和 NULL 进行有意义的相等或不等比较。

但是，如果两个指针是指向不同数组的，那么这两个指针间的运算和比较就是没有意义的，也是行不通的。因此，必须牢记，千万不能拿两个指向不同数组的指针进行比较和运算。这种做法如果走运，所用编译器会给你指出错误；如果不走运，这样生成的代码可能在一个机器上能够运行，而在另一个机器上则出现莫明其妙的失败。

18.4 使用字符指针

可以用字符数组处理内存中的字符串，然而用指针处理内存中的字符串时，指针的可改变性将会更加充分地显示出来。下面语句分别声明和初始化一个字符数组和一个字符指针：

```
char str[ ]="I like C++";
char * ptr="C++ is fun";
```

C++在处理上述两个声明和初始化语句时，基本上是以相同的方法存贮两个串。

当把一个字符串赋给一个字符指针时，C++首先寻找足够的空间来存放该字符串，然后把该字符串的第一字符的地址赋给指针。上述两个串定义语句在多数情况下所做的工作几乎是相同的。

例如，可以用打印语句从串的第一个字符开始打印，直到遇到空零(null zero)：

```
cout<<"str is" <<str<<"\n";
cout<<"ptr is" <<ptr<<"\n";
```

这时，用数组和用指针存放字符串之间几乎没有什么区别。然而我们知道，数组名是指针常量，因此不能改变它的值。下面赋值语句是非法的：

```
str="Chinese";                    //非法。
```

要改变数组中存放的内容的唯一途径是每次一个字符地把新串的内容赋给数组的各个元素，或使用象 strcpy()这种内部函数给字符数组赋值。

可以方便地使字符指针指向新的字符串：

```
prt="Chinese";                    //改变指针使它指向新串。
```

在实际应用中，当用户需要把输入存放在一个指针指向的字符串中时，为了给字符串保留足够的存贮空间，往往首先声明一个字符数组，然后把数组的第一元素的地址赋给字符指针：

```
char str[1000];
char *ptr=str;
```

注意下述几个语句的区别：

```
cout<<str;                        //从第一个元素开始打印字符串直到遇到'\n'
cout<<ptr;                        //从第一个元素开始打印字符串直到遇到'\n'
cout<<*ptr;                       //打印第一个字符。
```

下述程序中首先声明一个字符数组和字符指针，然后把数组第一个元素的地址赋给指针。用指针打印出整个串和串中的各个字符。

```
#include "iostream.h"
void main()
{
    char str[ ]="abcdefghi";
    char * p =str;
    cout<<p<<"\n";                //打印出整个串。
    for(int i=0;i<9;i++)
    {
      cout<< *p<<'\n';            //每循环一次打印一个字符
      P++;
    }
}
```

这里必须再次强调指出，用 cout 从数组或指针名开始打印字符串，直到遇到空零结束(null aero)。因此下述语句打印出整个字符串，而不是串中第一个字符的地址：

```
cout<<p;
cout<<str;
```

如果这里定义的不是字符数组和字符指针，而是整型数组和指向整型数组的指针，那么，情况就不一样了。请看下面一个例子。

下述程序中定义并初始化了一个整型数组，随后声明了一个整型指针，然后把数组的第一个元素地址赋给指针，并用指针操作数组并打印出各元素。注意把本例与上述程序进行比较。

```
#include "iostream.h"
void main()
{
    int ara[ ]={1,2,3,4,5,6,7};
    int * para=ara;
    cout<<para<<"\n";          //打印第一个元素的地址!
    cout<<*para<<"\n";         //打印 1;
    for(int i=0,i<7;i++)
    {
      cout<< *para;            //打印所有元素值
      para++;
    }
    return;
}
```

18.5 字符型指针数组

前面讲过指针数组，其实最有用的指针数组是字符型指针数组。由于这种数组能够非常经济地存放字符串，因此，在这里我们有时也称其为字符串数组。

我们学习过多维数组。用一个二维表也能够存放若干字符串，但这样做通常要浪费掉一些存贮单元。因为在定义二维表时，不论表中每行存放多少个字符，都必须以最大字符数分配存贮单元。下面我们定义一个二维数组，用它存放十二个月的名字：

```
char name[5][10]={
                  "January",
                  "February",
                  "March",
                  "April",
                  "May",
}
```

图 18.2 显示了这个二维数组在内存中的情况：

J	a	n	u	a	r	y	\0		
F	e	b	r	u	a	r	y	\0	
M	a	r	c	h	\0				
A	p	r	i	l	\0				
M	a	y	\0						

图 18.2

从图中可以看见，当遇到上述二维数组声明和初始化语句时，C++根据二维数组的维数计算出总元素的个数(本例为 50 个元素)，并在内存中为之分配内存，把对应的字符串，放到各行中。分配给数组的内存是受保护的，其它变量不允许侵占这些分配给数组的存贮单元。

然而可以看出，由于为二维数组分配内存时，每一行的元素个数都必须相等，但存在每行中字符串中字符的个数却不一定相等，这样一来必会造成一些存贮单元的浪费。怎样才能避免这种浪费呢？

前面我们提到的字符型指针数组可以很好地解决这个问题。对于上述问题，可以声明一个一维字符型指针数组。

字符型指针数组的每个元素为一个字符指针，它指向内存中的一个字符串。这样一来，这些字符串就不一定要求必须具有相同的长度。这种数组，有时也称为边缘不齐的数组。下面是这种数组的一个定义，

```
char *name[ ]={
                "January",
                "February",
                "March",
                "April",
                "May",
                "June",
}
```

尽管过去从未见过这种定义的数组，它仍然是很容易理解的，因为我们对字符串和字符指针的使用，以及字符数组都已经有了比较深刻的了解。数组名 name 前面的 * 使该数组成为一个指针数组，类型为字符型。各字符串并不是赋给该数组的各个元素。而是数组的各个元素作为指针指向各字符串。

因为每个字符串都有一个指针元素指向它的开始位置，因此它们在内存中的具体位置并不重要。而且，每个字符串只占它们自身以及结束标志 '\0' 所需的内存空间，因此这种数组不浪费任何存贮空间。

图 18.3 显示了这种数组在内存中的情况：

由图中可以看出，这种数组的边缘呈不齐状态。边缘不齐(ragged_adge)正是来源于此。

下述语句分别打印各字符串的值：

```
cout<<*(name);      //打印出 January
cout<<*(name+1);    //打印出 February
cout<<*(name+2);    //打印出 March
cout<<*(name+3);    //打印出 April
cout<<*(name+4);    //打印出 May
cout<<*(name+5);    //打印出 June
```