II

第二篇　高级 C++程序语言

本篇介绍指针，字符串，函数的高级应用，预处理指令，数据流与文件的存取，输出格式，程序计时，struct 与数据结构，名称空间和异常处理等主题。学习完这个部分，就可以拥有使用文件存取数据，自由设定数据格式，并将大型程序区分为许多小文件，以解决实际问题的能力。

在第九章“字符串”中，我们有一个关于编码的有趣程序，读者可以用来把电子邮件转成只有拥有破译码才能理解的文字，便于机密文件的传递。

第 8 章　指针

第 9 章　字符串

第 10 章　函数的高级应用

第 11 章　预处理指令

第 12 章　数据流与文件的存取

第 13 章　输出格式

第 14 章　程序计时

第 15 章　struct 与数据结构

第 16 章　名称空间

第 17 章　异常处理

第8章

指针（pointer）

基本上，指针（pointer）是一种特别的数据类型，用来储存某一数据在内存中的地址。虽然对于简单的应用程序而言，函数调用时使用参数的引用（reference）已经能够应付大部分的需要，但指针的使用在复杂程序的开发上常能大幅提高处理效率。事实上，要完全发挥 C++ 的功能，以及有效率地使用窗口开发环境，譬如调用微软公司的 Microsoft Foundation Classes (MFC)，充分了解指针是必要的。

8.1 内存地址与指针
8.2 指针与引用
8.3 数组与指针的代数计算
8.4 指针参数
8.5 函数指针
8.6 动态内存分配
8.7 常犯的错误
8.8 本章重点
8.9 本章练习

8.1 内存地址与指针

在内存内的数据

所有的数据都必须存在内存内才能处理，每个数据在内存内都有下列四个相关的特性：

1. 数据名称；
2. 数据类型，用来规范该数据的储存方式和占用的内存大小；
3. 数据内容；
4. 该数据所在的内存地址。

例如：

```
float A = 2.5;        // 定义 float 变量 A 并存入 2.5
```

所对应的是一个名叫 A 的 float 变量，其占用的内存大小可以用 sizeof(float) 或是 sizeof(A) 求得 (通常为 4 bytes)，目前内存内所储存内容为 2.5，而其所在地址则可以用取址运算符 (address operator) "&" 来得到。图标如图 8.1.1 所示：

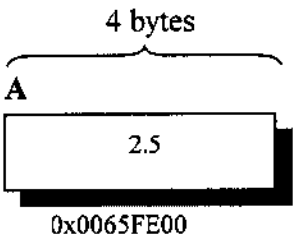


图 8.1.1 变量 A 的内容和地址

这个推论可以用程序 Variable.cpp 来证实。

范例程序　文件 Variable.cpp

```
// Variable.cpp

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

// ----- 主程序 ------------------------
int main()
{
   float A = 2.5;

   cout << "A 的值为: "    << A << endl;
   cout << "A 所占用的内存大小为: "
        << sizeof(A)      << " 或是 "
        << sizeof(float) << endl;
   cout << "A 所在地址为: "
        << "0x"  <<   &A  << endl;
}
```

操作结果

```
A 的值为: 2.5
A 所占用的内存大小为: 4 或是 4
A 所在地址为: 0x0065FE00
```

C++ 程序对于地址的预设表示法为十六进制（例如图 8.1.1 中的 0x0065FE00），这个表示法只是便于我们了解，与计算机内部的实际运作无关.

指针

为了把数据的地址储存下来，我们必须定义指针（pointer）。指针是用来储存其它数据的地址的特殊变量。例如，我们可以使用以下语句来定义一个名叫 pF 的指针：

```
float* pF;      // 定义指针 pF
pF = &A;        // 将数据 A 的地址存入指针 pF
```

具体表示如图 8.1.2 所示：

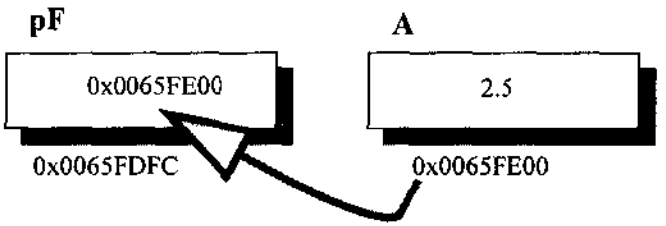


图 8.1.2　指针 pF 的内容和变量 A 的地址之间的关系

由于指针本身也是数据的一种，因此它也具有一般数据所有的“名称”、“数据类型”、“数据内容”和“内存地址”等四项特性。对于指针 pF 而言，它的

名称：　pF
数据类型：　float *
数据内容：　0x0065FE00
内存地址：　0x0065FDFC。

尤其是，虽然指针的数据内容都是 unsigned int (没有正负号的整数)，但声明时所指定的目标数据的数据类型 (例如上述的 **float ***) 无法改变。也就是说，它只能储存声明时所指定的数据类型的地址。譬如， pF 已声明为

```
float* pF;          // 定义指针 pF
```

则 pF 储存的地址所标志的内存空间都必须用来存放类型为 float 的数据.

假如　　int X = 12;

则语句　　pF = &X;　// 错误! 指针 pF 必须存放 float 数据的地址

是错误的语法，将在编译时发生问题，必须配合目标数据的数据类型改成

```
int* pF;
```

才可以.

在一般的情况下，指针是用来间接存取数据的工具，使用者不需要确实知道它所储存的地址数值. 因此，我们通常将上述指针 pF 和变量 A 之间的关系说成“指针 pF 指向变量 A”，并以图 8.1.3 的示意图具体表达:

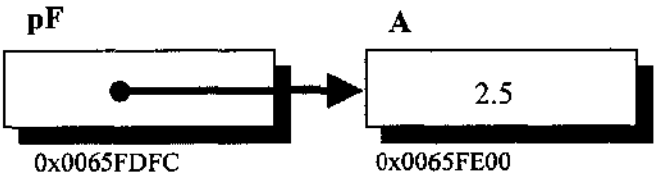


图 8.1.3　指针 pF 和变量 A 之间的关系

指针的定义

指针的定义语句有两种写法:

```
float*  pF;      // 定义指针 pF (较常用)
```

或　`float  *pF;`　　// 定义指针 pF (较少用)

这两种写法所定义的都是 pF，而不是 *pF.

如果要同时定义两个指针时，必须将语句写成

```
float *p1, *p2; // 同时定义 p1, p2 两个指针
```

如果写成

```
float* p1, p2;
```

则只有 p1 被定义成指向 float 变量的指针，而 p2 被定义成一个 float 变量.

我们可以在定义 pF 的同时给它初值，例如:

```
float*  pF = &A;   // 定义指针 pF，同时存入数据 A 的地址
```

表示“pF 这个指针变量储存了 A 的地址”。这个讲法常常简洁地说成“指针 pF 指向变量 A”。“*”是所谓的“取值运算符”(indirection operator)，当取值运算符和指针一起使用时，譬如

```
float x;          // 定义 float 变量 x
x = *pF;          // 将指针 pF 所指处的数据存到变量 x
```

表示的是“将存在指针 pF 所指之处的内容储存到 x 里面”，也就是说，x 被存进了数值 2.5。

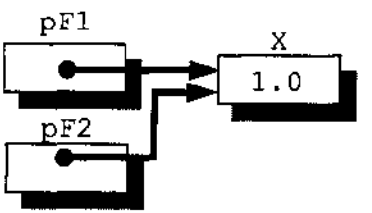
变量指针与常量指针

由于上述的指针本身是变量，因此，可以改变指针的内容，让它指向不同的数据。例如:

```
float* pF;  // 定义指针 pF
float X, Y;
pF = &X;    // pF 指向 X
pF = &Y;    // pF 改指向 Y
```

而且可以有好几个指针同时指向一个数据。例如:

```
float *pF1, *pF2;
float X = 1.0;
pF1 = &X;
pF2 = &X;
```



这个时候，从 pF1 通过取值运算符 (pointer dereference operator)“*”对内容的更动，会作用在同一个数据上。例如，下列两个语句:

```
*pF1 = 56;       // 在 pF1 所指向的位置存入 56
cout << *pF2;  // 将 pF2 所指向位置的内容输出
```

的输出为 56。

但是，并不是所有的指针都是变量。例如，我们在上一章介绍数组时，就发现数组名的值是数组开头元素的地址；也就是说，数组名即是指针。例如，具有 5 个元素的数组 V 的定义和声明语句是：

```
double V[5];          // 定义数组 V
```

数组的名称 V 就是这个数组的指针（见图 8.1.4）：

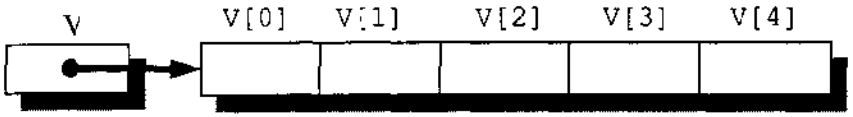


图 8.1.4 一维数组名和指针之间的关系

指针 V 内部所存的是这个数组第一个元素 V[0] 的地址。这个内容是不允许变动的，因此，V 是一个常量指针 (constant pointer)。

8.2 指针与引用

取址运算符 (address operator) “&” 和取值运算符 (pointer dereference operator) “*” 是两个相反操作的运算符。假如我们已定义了一个变量 x 和一个可以指向该变量数据类型的指针 p（见图 8.2.1）：

```
float  X = 3.8;          // 声明和初始化 X
float* p = &X;           // 声明和初始化 p
```

则其关系相当于

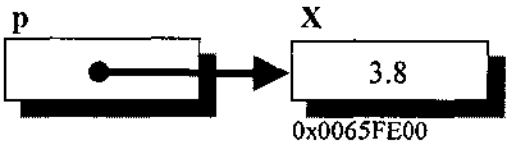


图 8.2.1 指针 p 和变量 X 之间的关系

这时候

*p 是 X 的内容 （亦即上图中的 3.8）

&X 是 p 的内容 （亦即 X 的地址 0x0065FE00） 。

我们曾在 6.2 节“以引用的方式调用”中介绍过变量引用的定义：

```
float& y = x;    // y 是 x 的引用
```

表示“y 是 x 的一个别名” ，对于 y 的任何存取动作，都相当于直接作用在 x 上面.

指针和引用的声明有一个很大的区别：指针声明时不需要初始化，并可以随后再予以设定。而引用在声明时就必须指定其引用的对象，而且此关系不可改变。例如

```
float &x;          // 错误！必须给 x 引用的对象
```

是错误的声明。而且

```
float y, z;          // 定义 y 和 z 两个 float 变量
float &x = y;        // 声明 x 是 y 的引用
float &x = z;        // 错误！不能更改引用的对象
```

是错误的指令，不能为编译器接受。

程序 Ref.cpp 是一个使用引用和指针的简单程序，用来验证上述特性。

范例程序　文件 Ref.cpp

```
// Ref.cpp
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;

// ------ 主程序 ----------------------
int main()
{
float x  = 1.0;
float &y = x;     // 定义 x 的引用 y
float *p = &x;    // 定义和初始化指针 p

 cout << "x 原来的值是 " << x << endl;
 *p = 5.0;
 cout << "执行 *p = 5.0; 后\n";
 cout << "x 的值是     " << x << endl;
 y = 7.3;
 cout << "执行 y = 7.3; 后\n";
 cout << "x 的值是     " << x << endl;
}
```

操作结果

```
x 原来的值是 1
执行 *p = 5.0; 后
x 的值是     5
执行 y = 7.3; 后
x 的值是     7.3
```

提示

语句(1)　　float *p;　　// 定义指针 p

固然可以解读为："指针 p 的内容的数据类型为 float" 而

语句(2)　　float &y = x　　// 定义 y 是 x 的引用

可以解读为"变量 y 的地址与数据类型为 float 的变量 x 一样"，但是 (1) 和 (2) 这两个语句却是分别用来定义指针 p 和引用 y。基于此认识，正确的想法应该是："将 * 和 & 两个运算符在上面 (1) 和 (2) 这两个定义语句中视为具有特殊意义，用来定义指针和引用"，而不是原来的取址运算符 (address operator) "&" 和取值运算符 (pointer dereference operator) "*"。

8.3 数组与指针的代数计算

使用下标来计算数组元素的地址

一维数组就是向量 (vector)，我们在 7.1 节和 7.3 节已分别学过如何以下标来计算数组元素的地址。例如，有一个长度为 5 数据类型为 double 的向量 A：

```
const int m = 5;
double A[m];
```

则第 k+1 个元素，也就是 A[k] 的地址为

```
&A[0] + k × sizeof(double)
```

对于一个 20 ×60 的 double 矩阵 B：

```
const int m = 20;
const int n = 60;
double B[m][n];
```

元素 B[i][j] 的地址为

```
&B[0][0] + (i × n + j) × sizeof(double)
```

对于一个 20 ×60 ×80 的 double 张量 C：

```
const int m = 20;
const int n = 60;
const int p = 80;
double C[m][n][p];
```

元素 C[i][j][k] 的地址为

```
&C[0][0][0] + (i × n × p + j × p + k) ×sizeof(double)
```

使用指针来存取数组元素：指针代数

除了上述获得元素地址的方式外，使用指针可以获得更高的效率。由于指针的内容储存的是地址，因此也可以接受加减乘除四则运算以及关系运算 (==, !=, <, <=, >, >=) 进行大小的比较。

考虑上述向量 A，如果我们进一步定义一个指针变量 p：

```
float *pV;
```

则下面两式都可以将 A[0]的地址存到 pV 里面：

```
pV = &A[0];
pV = A;
```

如图 8.3.1 所示。

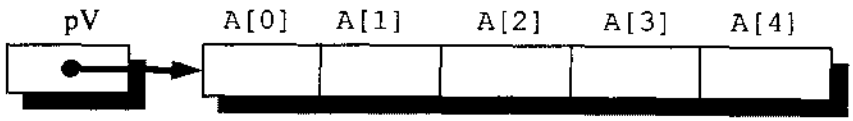


图 8.3.1　一维数组(向量)和指针变量 pV 之间的关系

通过指针变量 pV，一维数组的元素就可以使用取值运算符 * 获得：

*pV	相当于 A[0]
*(pV+i)	相当于 A[i]

我们可以写成通式：

A[i] = *(pV+i)　　这里 i 的范围为 0～(m-1)。

我们注意到，以指针计算元素地址时，由于声明指针时已经包括元素数据类型（此例为 double）的信息，每个元素所占用的内存大小不需要再另外指明。也就是说，我们不用写成

```
pV + i × sizeof(double)
```

以获得 A[i] 的地址，也不需要用 *(pV + i × sizeof(double)) 去求 A[i] 的内容。这个特殊的计算方式称为指针代数（pointer algebra）或指针算术（pointer arithmetic），可以简化语句。

提示

指针与取值运算符、增减运算符常结合在一起，写成很精简的语句。我们将这些操作的语法归纳成下表：

*++p	相当于	*(++p)	先增加指针再取用元素
*p++	相当于	*(p++)	先取用元素再增加指针
*--p	相当于	*(--p)	先减少指针再取用元素
*p--	相当于	*(p--)	先取用元素再减少指针

我们在程序 Apointer.cpp 中使用 *pV++ 去逐一获得向量的各个元素，以求得向量的总和：

范例程序　文件 Apointer.cpp

```
// Apointer.cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

// ----- 主程序 ---------------------------------
int main()
{
  const int Size =  5;
  float V[Size]  = {48.4, 39.8, 40.5, 42.6, 41.2};
  float Sum = 0.0, Average;
  float *pV = V;

    for (int i=0; i<Size; i++)
        Sum += *pV++;
    cout  << "数组 V 各元素的总和是  : "
          << Sum << endl;
    cout  << "数组 V 各元素的平均值是: "
          << Sum / float(Size) << endl;
}
```

操作结果

```
数组 V 各元素的总和是  : 212.5
数组 V 各元素的平均值是: 42.5
```

讨 论

由于指针可以比较大小，所以程序 Apointer.cpp 中

```
for (int i=0; i<Size; i++)
    Sum += *pV++;
```

可以进一步改写为

```
while (pV < V + Size )
Sum += *pV++;
```

二维数组的指针

首先，我们考虑一个 2 ×3 的 double 矩阵 B：

```
const int m = 2;
const int n = 3;
double B[m][n];
```

此声明产生了一个二维数组，亦即矩阵 (matrix)，以及一组常量指针：B，B[0] 和 B[1]。在内存内，他们的相对关系如图 8.3.2 所示：

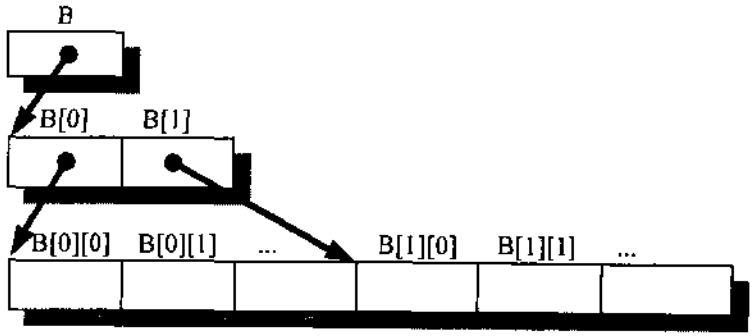


图 8.3.2 二维数组 B 的储存方式及相关的指针

也就是说，二维数组可以看成是由“行”为单位元素所组合而成的向量，其中各“行”本身又是由很多元素组成的向量。指针 B[0] 指向 B[0][0] 的起始地址，指针 B[1] 指向 B[1][0] 的起始地址；而指针 B 则指向 B[0]，是“指针 B[0] 的指针”。因此，表 8.3.1 中的三种表示法是完全相等的，可以互换：

表 8.3.1　二维数组的元素表示法

下标表示法	指针表示法（1）	指针表示法（2）
B[0][0]	*B[0]	*(*B)
B[0][1]	*(B[0]+1)	*(*B+1)
B[0][2]	*(B[0]+2)	*(*B+2)
B[1][0]	*B[1]	*(*(B+1))
B[1][1]	*(B[1]+1)	*(*(B+1)+1)
B[1][2]	*(B[1]+2)	*(*(B+1)+2)

其中指针表示法（2）能够成立的原因是由于 *B 和 B[0] 是一样的，而 *(B+1) 又等于 B[1]。

我们可以进一步写成通式，将上表整理成：

```
B[i][j] ≡ *(B[i] + j) ≡ *(*(B+i) + j)
```

下列程序 ShowMatrix.cpp 将上面三种二维数组的元素表示法写成完整的程序，分别用来显示同一个矩阵，以证实指针表示法的正确性。

范例程序　文件 ShowMatrix.cpp

```
// ShowMatrix.cpp
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;
using std::setw;
const int Row = 2;
```

```
const int Col = 3;
// ----- 主程序 ----------------------
int main()
{
  float B[Row][Col]={ 1.8, 4.9, 6.8,
                      6.2, 2.1, 3.4};
  // ---------------------------------
  cout  << "(1)数组 B 是: " << endl;
  for (int i=0; i<Row; i++)
    {
      for (int j=0; j<Col; j++)
        cout << setw(5) << B[i][j];
      cout << endl;
    }
  cout << endl;
  // ---------------------------------
  cout  << "(2)数组 B 是: " << endl;
  for (int i=0; i<Row; i++)
    {
      for (int j=0; j<Col; j++)
        cout << setw(5) << *(B[i] + j);
      cout << endl;
    }
  cout << endl;
  // ---------------------------------
  cout  << "(3)数组 B 是: " << endl;
  for (int i=0; i<Row; i++)
    {
      for (int j=0; j<Col; j++)
        cout << setw(5) << *(*(B+i) + j);
      cout << endl;
    }
  return 0;
}
```