第15章

struct 与数据结构

有别于 int，float，double 与 char 等基本数据类型，以及由相同数据类型元素组成的“数组”，本章介绍的 struct 可以在同一个名称下拥有多种数据类型。使用 struct 能让数据的存取和处理更为灵活。

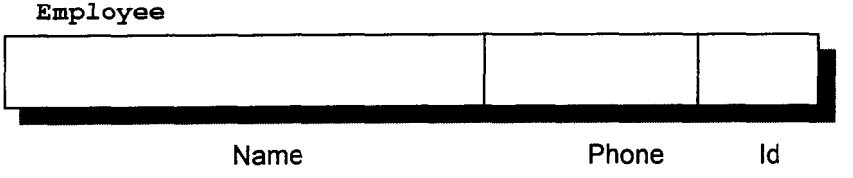
15.1 struct 的声明和使用

使用 struct 可以由使用者自行声明合乎自己需要的数据类型．其组成成分称为成员 (member) 或数据成员 (data field)．而各成员本身又可以是各种不同的数据类型．例如，我们可以声明一个新的复合式数据类型，取名为 Employee，包括的数据成员有 Name (姓名)，Phone（电话号码）以及 Id（编号）三种：

```
struct Employee
{
    char  Name[20];
    char  Phone[10];
    int  Id;
};                          //   注意这要用到“;”!
```

在上述名叫 Employee 的 struct 数据类型声明中，关键词 struct 是英文 structure （结构）的缩写，此种数据结构又称为记录（record)．

由于各成员的储存位置是连续的，我们可以将 Employee 在内存中的储存方式图示如下：



在以下的说明中，我们将“由 struct 声明的数据类型”简称为“struct 数据类型”．上述的 Employee 就是一种“struct 数据类型”。

一旦声明了自定的 struct 数据类型后，就可以使用标准的定义语句来定义新的变量．例如：

```
Employee Ea, Eb;
```

就定义了两个名称分别为 Ea 和 Eb 的 Employee 变量．由某一数据类型定义的变量称为该数据类型的实例 (instance)．Ea 和 Eb 都是 Employee 的实例．

我们也可以在声明 Ea 和 Eb 时一并给予初始值．例如，上面的语句可进一步写成

```
Employee Ea = {"Ann",    "02384125", 105};
```

```
Employee Eb = {"Joanne", "03544132", 106};
```

要存取 Employee 变量的个别数据成员时，必须同时给定变量名称和数据成员名称，中间用一个成员运算符号（member operator）“.”隔开。这个成员运算符号实际上就是小数点符号。例如：

```
Ea.Name
Eb.Phone
Ea.Id
```

分别用来代表 Ea 这个 Employee 变量的三个成员，其值目前分别为 "Ann"，"02384125" 和 105。这个语法基本上和我们在 10.1 节介绍的成员函数的语法是一致的。

讨论

就内存的使用而言，语句

```
Struct Employee
{
   char  Name[20];
   char  Phone[10];
   int  Id;
};
```

只完成了 struct 数据类型的声明，它并没有在内存中安排内存空间。一直到执行

```
Employee Ea, Eb;
```

的语句后，才真正规划出需要的内存空间以供各成员内的数据储存。

在下面的范例程序中，我们可以仔细观察如何使用 struct 声明自定义的数据类型，以及各字段内的数据如何存取。

范例程序 文件 Pgm15-1.cpp

```
// Pgm15-1.cpp
#include <iostream>
using std::cout;
using std::endl;

struct Employee
{
    char  Name[20];
    char  Phone[10];
    int  Id;
};

// ----- 主程序 ----------------------------
int main()
{
  Employee Ea = {"Ann",    "02384125", 105};
  Employee Eb = {"Joanne", "03544132", 106};

  cout << "Ea 的数据是:\n"
       << "姓名    : " << Ea.Name  << '\n'
       << "电话号码: " << Ea.Phone << '\n'
       << "编号    : " << Ea.Id    << endl;
  cout << "Eb 的数据是:\n"
       << "姓名    : " << Eb.Name  << '\n'
       << "电话号码: " << Eb.Phone << '\n'
       << "编号    : " << Eb.Id    << endl;
  return 0;
}
```

此为试读,需要完整PDF请访问: www.ertongbook.com

执行结果

```
Ea 的数据是:
姓名    : Ann
电话号码: 02384125
编号    : 105
Eb 的数据是:
姓名    : Joanne
电话号码: 03544132
编号    : 106
```

除了上述的语法外，“struct 数据类型”的声明和变量定义也可以写成单一语句。例如，本节 Employee 的声明可以合并 Ma 和 Mb 的定义而写成以下的样子：

```
struct
{
    char Name[20];
    char  Phone[10];int  Id;
} Ea, Eb;
```

在这个语法中，数据类型名称 Employee 由于不需要而去除了。这是因为上式粗体字的部分

```
struct {char Name[20];  char Phone[10]; int Id;}
```

本身就已经是新定义的数据类型之具体内容，不用再取个名称来代表它了。仔细比较以下与基本数据类型 int 声明变量的语法就可以更清楚了：

```
int          x, y ;
```
数据类型　　变量名称

```
struct {char Name[20];char Phone[10]; int Id;}   Ea, Eb ;
```
数据类型　　变量名称

(实例)

15.2 由 struct 构成的数组

数组的每一个成员都具有相同的数据类型，其成员由共同的数组名加上下标(subscripts)来存取．结合数组和 struct，可以一次完成很多具有相同结构的 struct 变量的定义．

例如，承续 15.1 节 Employee 的声明，我们可以使用

```
Employee Officer[50];
```

同时定义从 Officer[0]到 Officer[49]，共 50 个 Employee 变量．假如我们想要知道 Officer[12]的各字段的数据，可以直接写成：

```
cout << Officer[12].Name  << endl;
cout << Officer[12].Phone << endl;
cout << Officer[12].Id    << endl;
```

由于程序通常用来处理大量的数据，因此，使用 struct 写成数组的形式才是最常看到的用法．

以下 Pgm15-2.cpp 是一个允许使用者逐一输入各数组元素的各成员值的范例程序 (每输入一个项目后，要按两次 Enter 键)．

范例程序 文件 Pgm15-2.cpp

```
// Pgm15-2.cpp
#include <iostream>
using std::cin;    using std::cout;
const int Size       = 2;
const int NameSize   = 20;
const int PhoneSize  = 10;
struct Employee
{
    char  Name[NameSize];
    char  Phone[PhoneSize];
};
// ----- 主程序 ----------------------------
```

```
int main()
{
  Employee Officer[Size];
  cout << "共 " << Size << " 个 Officers:\n";
  for (int i=0; i<Size; i++)
  {
     cout << "请输入 Officer[" << i
          << "] 的姓名: ";
     cin.getline(Officer[i].Name, NameSize, '\n');
     cout << "电话号码: ";
     cin.getline(Officer[i].Phone,PhoneSize, '\n');
  }
  for (int i=0; i<Size; i++)
  {
     cout << "Officer["     << i << "] 的数据是:\n"
          << "姓名    : "   << Officer[i].Name  << '\n'
          << "电话号码: "   << Officer[i].Phone << '\n';
  }
  return 0;
}
```

执行结果

```
共 2 个 Officers:
请输入 Officer[0] 的姓名: Alan John
电话号码: 03-4521234
请输入 Officer[1] 的姓名: Peter Pan
电话号码: 02-4354512

Officer[0] 的数据是:
姓名    : Alan John
电话号码: 03-4521234
Officer[1] 的数据是:
姓名    : Peter Pan
电话号码: 02-4354512
```

15.3 struct 数据类型与函数参数的传递

由 struct 所声明的数据类型与基本数据类型，如 int，float，double 与 char 等，在使用上具有相同的特性和语法。在 15.1 节中，我们看到了自定的数据类型 Employee 如何被用来定义其实例（instances） Ea 和 Eb。当这些实例被用来作为参数传递时，其预设的语义也是传值（pass-by-value）。也就是说，在“被调用函数”内部将另外产生一个复制数据，而不会影响“调用函数”内的数据。

例如，在“调用函数”内，调用语句可以写成：

```
ShowMember(Ea);
```

而“被调用函数”则可以定义成：

```
void ShowMember(Employee A)
{
  cout << "数据的详细内容是:\n"
       << "姓名    : " << A.Name  << '\n'
       << "电话号码: " << A.Phone << '\n'
       << "编号    : " << A.Id    << endl;
  return;
}
```

如果要改变变量内容，可以使用传引用（pass by reference）的语法。例如，在“调用函数”内，将变量与所要输入的字符串当成参数，而写成：

```
ChangeName(Ea, "Jackson");
```

对应的“被调用函数”则定义成：

```
void ChangeName (Employee& A, char NewName[])
{
  strcpy(A.Name, NewName);
  return;
}
```

如此，可以改变原有实例 Ea 中 Name 栏内的字符串内容。详见范例程序

Pgm15-3.cpp.

范例程序　文件 Pgm15-3.cpp

```
// Pgm15-3.cpp
#include <iostream>
using namespace std;

struct Employee
{
    char  Name[20];
    char  Phone[10];
    int  Id;
};

void ShowMember(Employee A)
{
  cout << "数据的详细内容是:\n"
       << "姓名    : " << A.Name  << '\n'
       << "电话号码: " << A.Phone << '\n'
       << "编号    : " << A.Id    << endl; return;
}

void ChangeName (Employee& A, char NewName[])
   { strcpy(A.Name, NewName); return; }

// ============= 主程序 ========================
int main()
{
  Employee Ea = {"Ann",    "02384125", 105};
  Employee Eb = {"Joanne", "03544132", 106};
  ShowMember(Ea);
  ShowMember(Eb);
  ChangeName(Ea, "Jackson");
  cout << "执行 ChangeName() 后:\n";
  ShowMember(Ea);
```

```
  return 0;
}
```

执行结果

```
数据的详细内容是:
姓名    : Ann
电话号码: 02384125
编号    : 105
数据的详细内容是:
姓名    : Joanne
电话号码: 03544132
编号    : 106
执行 ChangeName() 后:
数据的详细内容是:
姓名    : Jackson
电话号码: 02384125
编号    : 105
```

使用指针

为了能够在使用函数时存取同一个变量，除了使用引用外，我们也可使用传址（pass-by-address）来达到同样的目的．以本节引用的 Employee 数据类型为例，我们可以在“调用函数”内使用以下的语句来调用：

```
ChangeId(&Ea,  00128);
```

而“被调用函数”则定义为

```
void ChangeId(Employee* pE, int NewId)
{
  (*pE).Id=NewId;
  return;
}
```

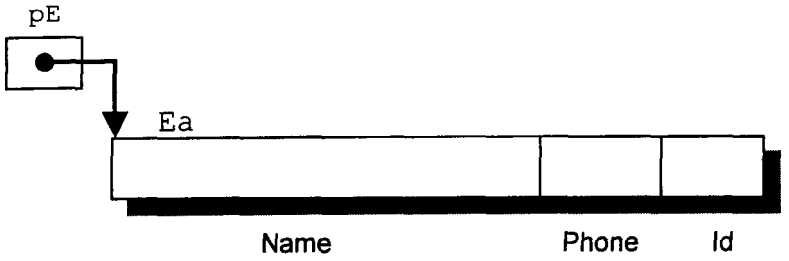
此为试读，需要完整PDF请访问：www.ertongbook.com

当 ChangeId() 被调用时，指针 pE 获得 Ea 的起始地址。而 (*pE).Id 表示由指针 pE 所指向的 Employee 变量内的字段 Id。

提示

这里*pE 两边的小括号 () 是必须的，因为成员运算子“.”的优先权高于取值运算子“*”。如果把小括号省略，则其语意相当于 *(pE.Id)，亦即“pE.Id 这个指针所指向位置内的数据”，显然会造成错误的结果。

pE 与 Ea 之间的关系如下图所示：



由于指针的这种用法经常遇到，因此 C++ 有一个更形象的符号，将 (*pE).Name 写成

```
pE->Name
```

其中“->”是由“负号”和“大于”两个符号构成，以代表一个“箭头”，表示“由 pE 指向的 struct 变量内的成员 Name”的意思。

因此，　ChangeId()可进一步改写为

```
void ChangeId(Employee* pE,  int NewId)
{
  pE->Id = NewId;
  return;
```

```
}
```

范例程序 文件 Pgm15-4.cpp

```
// Pgm15-4.cpp
#include <iostream>
using std::cout;    using std::endl;
struct Employee
{
    char  Name[20];
    char  Phone[10];
    int  Id;
};

void ShowMember(Employee A)
{
  cout << "数据的详细内容是:\n"
       << "姓名    : " << A.Name   << '\n'
       << "电话号码: " << A.Phone  << '\n'
       << "编号    : " << A.Id     << endl; return;
}
void ChangeName (Employee& A, char NewName[])
   { strcpy(A.Name, NewName); return; }
void ChangeId(Employee* pE,  int NewId)
   {  pE->Id = NewId;           return;}

// ========= 主程序 ========================
int main()
{
  Employee Ea = {"Ann",    "02384125", 105};
  Employee Eb = {"Joanne", "03544132", 106};
  ShowMember(Ea);
  ShowMember(Eb);
  ChangeId(&Ea,  208);
  cout << "执行 ChangeId() 后:\n";
  ShowMember(Ea);
```

```
    return 0;
}
```

执行结果

```
数据的详细内容是:
姓名    : Ann
电话号码: 02384125
编号    : 105
数据的详细内容是:
姓名    : Joanne
电话号码: 03544132
编号    : 106
执行 ChangeId() 后:
数据的详细内容是:
姓名    : Ann
电话号码: 02384125
编号    : 208
```

15.4 struct 实例的动态声明

我们在 8.6 节曾经介绍过“动态地定义数组”的语法。同样地，我们也可以动态地产生由 struct 实例所构成的数组，亦即 struct 实例的动态内存分配(dynamic memory allocation)。

以下的语句：

```
Employee Labor[50];
```

所定义的陈列 Labor 在编译时就已决定其大小为 50；而下列的语句则可以在执行时才临时决定数组的大小：

```
int Size;
```

```
cin >> Size;
Employee* pE = new Employee[Size];
```

此段程序执行后会依 Size 指定的大小在内存的特殊区域，称为内存堆 (heap) 的地方，规划出需要的内存空间，并把第一个变量的开头地址存入指针 pE 内．如果此数组不再需要，可以执行下列的语句回收内存空间：

```
delete [] pE;
```

提示

回收 struct 实例的语法必须包括中括号 []，而由基本数据类型构成的数组在回收时则可以省略语句里的中括号．

要存取上述 Employee 或 pE 内部的成员 Id 时，可以使用数组下标 (subscripts) 或指针算术 (pointer arithmetic) 的语法．例如，要取用第 k 个数组元素内的成员 Id，下述语法都是正确的：

```
Labor[k].Id
(*(Labor + k)).Id
(Labor + k)->Id

pE[k].Id
(*(pE + k)).Id
(pE + k)->Id
```

我们在范例程序 Pgm15-5.cpp 中示范动态产生由 struct 实例所构成的数组，称为 Employee 之完整语法，并在程序结束时回收内存空间．

范例程序 文件 Pgm15-5.cpp

```
// Pgm15-5.cpp
#include <iostream>
using std::cin;
using std::cout;
struct Employee
{
    char  Name[20];
    char  Phone[10];
    int  Id;
};
// --------- 主程序 ------------------------
int main()
{
 . int Size;
      cout << "请输入 Employee 的数目:\n";
      cin >> Size;
      Employee* pE = new Employee[Size];
      delete [] pE;
  return 0;
}
```

15.5 指针成员与数据结构

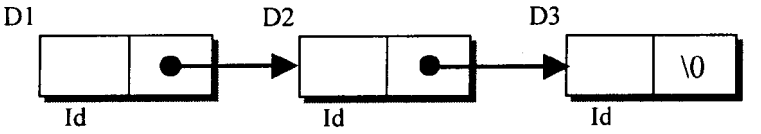
struct 所声明的数据类型可以使用“指针”作为成员．最特别的是指针成员可以指向自己所在的 struct 数据类型．例如：

```
Struct Data
{
  int  Id;
  Data* pD;
};
```

我们可根据这一特性定义一串 Data 变量:

```
Data D1, D2, D3;
D1.pD = &D2;
D2.pD = &D3;
```

图标如下:



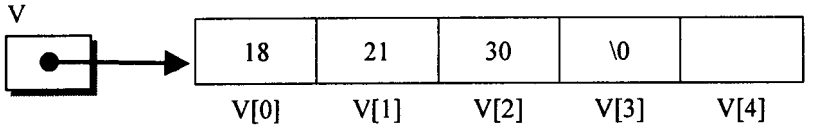
指针成员具有的这种特性称为“自我引用”(auto-reference)。“自我引用”让很多具有同样结构的数据以“指针”串联在一起，产生很多有用的数据结构，譬如列表（lists），堆栈（stacks），二叉树（binary trees）等。像上图前后相连的数据结构就是典型的列表。

此外，我们注意到列表最后一个元素内的指针值为 NULL（亦即 '\0'），表示不指向任何数据，可以用来作为检查列表是否“到此为止”的根据。

以下我们将检讨一下“列表”可以带来的便利。首先回顾一个由数组形成的 int 向量 V:

```
int* V = new  int [Size];
```

假设原先 Size 的值为 5，而且我们已将前三个元素分别储存了 18，21 和 30 三个值:



此为试读，需要完整PDF请访问：www.ertongbook.com