Data structures

Data structures

A data structure is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths. Data structures can be declared in C++ using the following syntax:

数据结构是在一个名称下组合在一起的一组数据元素。这些数据元素称为成员，可以具有不同的类型和不同的长度。数据结构可以在c++中使用以下语法声明:

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

Where `type_name` is a name for the structure type, `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{}`, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

其中type_name是结构类型的名称，object_name可以是具有该结构类型的对象的一组有效标识符。在大括号{}中，有一个包含数据成员的列表，每个成员都指定了一个类型和一个有效的标识符作为其名称。

For example:

```
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```

This declares a structure type, called `product`, and defines it having two members: `weight` and `price`, each of a different fundamental type. This declaration creates a new type (`product`), which is then used to declare three objects (variables) of this type: `apple`, `banana`, and `melon`. Note how once `product` is declared, it is used just like any other type.

它声明了一个称为product的结构类型，并定义它有两个成员:weight和price，每个成员都是不同的基本类型。这个声明创建了一个新类型(product)，然后用它来声明这个类型的三个对象(变量):apple、banana和melon。请注意，一旦product被声明，它的使用就像任何其他类型一样。

Right at the end of the `struct` definition, and before the ending semicolon (`;`), the optional field `object_names` can be used to directly declare objects of the structure type. For example, the structure objects `apple`, `banana`, and `melon` can be declared at the moment the data structure type is defined:

在结构定义的末尾，在结束分号(;)之前，可选字段object_names可用于直接声明结构类型的对象。例如，结构对象apple、banana和melon可以在定义数据结构类型时声明:

```
struct product
{
  int weight;
  double price;
} apple, banana, melon;
```

In this case, where `object_names` are specified, the type name (`product`) becomes optional: `struct` requires either a `type_name` or at least one name in `object_names`, but not necessarily both.

在这种情况下，在指定了object_names的情况下，类型名称(product)变成可选的:struct需要type_name或至少object_names中的一个名称，但不一定两者都需要。

It is important to clearly differentiate between what is the structure type name (`product`), and what is an object of this type (`apple`, `banana`, and `melon`). Many objects (such as `apple`, `banana`, and `melon`) can be declared from a single structure type (`product`).

明确区分什么是结构类型名称(产品)，什么是这种类型的对象(苹果、香蕉和甜瓜)是很重要的。可以从单个结构类型(产品)声明许多对象(如apple、banana和melon)。

Once the three objects of a determined structure type are declared (`apple`, `banana`, and `melon`) its members can be accessed directly. The syntax for that is simply to insert a dot (`.`) between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

一旦声明了确定的结构类型的三个对象(apple、banana和melon)，就可以直接访问它们的成员。它的语法只是在对象名和成员名之间插入一个点(.)。例如，我们可以操作这些元素中的任何一个，就好像它们是各自类型的标准变量:

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Each one of these has the data type corresponding to the member they refer to: `apple.weight`, `banana.weight`, and `melon.weight` are of type `int`, while `apple.price`, `banana.price`, and `melon.price` are of type `double`.

它们中的每一个都有对应于它们所引用的成员的数据类型:apple。重量,香蕉。体重,和西瓜。Weight的类型是int,而apple. Weight的类型是int。价格,香蕉。价格,和西瓜。价格是标准的两倍。

Here is a real example with structure types in action:

下面是一个实际使用结构类型的例子:

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

  mine.title = "2001 A Space Odyssey";
  mine.year = 1968;

  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;

  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;
```

```
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

The example shows how the members of an object act just as regular variables. For example, the member `yours.year` is a valid variable of type `int`, and `mine.title` is a valid variable of type `string`.

这个例子展示了对象的成员如何像常规变量一样工作。例如，你的会员。Year是int类型的有效变量，mine是。Title是字符串类型的有效变量。

But the objects `mine` and `yours` are also variables with a type (of type `movies_t`). For example, both have been passed to function `printmovie` just as if they were simple variables. Therefore, one of the features of data structures is the ability to refer to both their members individually or to the entire structure as a whole. In both cases using the same identifier: the name of the structure.

但是mine和你的对象也是类型为movies_t的变量。例如，两者都被传递给printmovie函数，就像它们是简单变量一样。因此，数据结构的特性之一是能够单独引用它们的成员或将整个结构作为一个整体引用。在这两种情况下使用相同的标识符:结构的名称。

Because structures are types, they can also be used as the type of arrays to construct tables or databases of them:

因为结构是类型，它们也可以用作数组的类型来构造表或数据库:

```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} films [3];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<3; n++)
```

```
  {
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
  }

  cout << "\nYou have entered these movies:\n";
  for (n=0; n<3; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

Pointers to structures

Like any other type, structures can be pointed to by its own type of pointers:

与任何其他类型一样，结构体也可以由其自身类型的指针指向:

```
struct movies_t {
  string title;  int year;
};
movies_t amovie;
movies_t * pmovie;
```

Here `amovie` is an object of structure type `movies_t`, and `pmovie` is a pointer to point to objects of structure type `movies_t`. Therefore, the following code would also be valid:

这里的movie是一个结构类型为movies_t的对象，而pmovie是一个指向结构类型为movies_t的对象的指针。因此，以下代码也是有效的:

```
pmovie = &amovie;
```

The value of the pointer `pmovie` would be assigned the address of object `amovie`.

指针pmovie的值将被赋给对象amovie的地址。

Now, let's see another example that mixes pointers and structures, and will serve to introduce a new operator: the arrow operator (`->`):

现在，让我们看看另一个混合了指针和结构的例子，并将用于引入一个新的操作符:箭头操作符(->):

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

The arrow operator (`->`) is a dereference operator that is used exclusively with pointers to objects that have members. This operator serves to access the member of an object directly from its address. For example, in the example above:

箭头操作符(->)是一个解引用操作符，仅用于指向具有成员的对象的指针。该操作符用于直接从对象的地址访问对象的成员。例如，在上面的例子中:

`pmovie->title`

is, for all purposes, equivalent to:

在所有情况下，等于:

`(*pmovie).title`

Both expressions, `pmovie->title` and `(*pmovie).title` are valid, and both access the member `title` of the data structure pointed by a pointer called `pmovie`. It is definitely something different than:

两个表达式，pmovie->title和(*pmovie)。Title是有效的，它们都访问由名为pmovie的指针所指向的数据结构的成员Title。它绝对不同于:

`*pmovie.title`

which is rather equivalent to:

`*(pmovie.title)`

This would access the value pointed by a hypothetical pointer member called `title` of the structure object `pmovie` (which is not the case, since `title` is not a pointer type). The following panel summarizes possible combinations of the operators for pointers and for structure members:

这将访问结构对象pmovie的一个名为title的指针成员所指向的值(事实并非如此，因为title不是指针类型)。下面的面板总结了用于指针和结构成员的操作符的可能组合:

Expression	What is evaluated	Equivalent
`a.b`	Member `b` of object `a`	
`a->b`	Member `b` of object pointed to by `a`	`(*a).b`
`*a.b`	Value pointed to by member `b` of object `a`	`*(a.b)`

Nesting structures

Structures can also be nested in such a way that an element of a structure is itself another structure:

结构也可以嵌套，这样结构的一个元素本身就是另一个结构:

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;
```

After the previous declarations, all of the following expressions would be valid:

在前面的声明之后，以下所有表达式都是有效的:

```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

(where, by the way, the last two expressions refer to the same member).

(顺便说一下，这里的最后两个表达式指的是同一个成员)。