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:

```
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.

For example:

```
1 struct product {
2   int weight;
3   double price;
4 } ;
5
6 product apple;
7 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.

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:

```
1 struct product {
2   int weight;
3   double price;
4 } 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.

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`).

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:

```
1 apple.weight
2 apple.price
3 banana.weight
4 banana.price
5 melon.weight
6 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`.

Here is a real example with structure types in action:

```
1  // example about structures
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  using namespace std;
6
7  struct movies_t {
8    string title;
9    int year;
10 } mine, yours;
11
12 void printmovie (movies_t movie);
13
14 int main ()
15 {
16   string mystr;
17
```

```
Enter title: Alien
Enter year: 1979

My favorite movie is:
 2001 A Space Odyssey (1968)
And yours is:
 Alien (1979)
```

```
  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`.

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.

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

```
// 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";
}
```

```
Enter title: Blade Runner
Enter year: 1982
Enter title: The Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these movies:
Blade Runner (1982)
The Matrix (1999)
Taxi Driver (1976)
```

Pointers to structures

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

```
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:

```
pmovie = &amovie;
```

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

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

```
// pointers to structures
#include <iostream>
```

```
Enter title: Invasion of the body snatchers
Enter year: 1978
```

```
#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;
}
```

```
You have entered:
Invasion of the body snatchers (1978)
```

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:

```
pmovie->title
```

is, for all purposes, equivalent to:

```
(*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:

```
*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:

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:

```
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:

```
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).

Previous: **Dynamic memory** Index Next: **Other data types**