Data structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not mere sequences of elements all of the same data type, but sets of different elements with different data types.

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 are declared in C++ using the following syntax:

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

where `structure_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.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as `structure_name` is created and can be used in the rest of the program as if it was any other type. For example:

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

product apple;
product banana, melon;
```

We have first declared a structure type called `product` with two members: `weight` and `price`, each of a different fundamental type. We have then used this name of the structure type (`product`) to declare three objects of that type: `apple`, `banana` and `melon` as we would have done with any fundamental data type.

Once declared, `product` has become a new valid type name like the fundamental ones `int`, `char` or `short` and from that point on we are able to declare objects (variables) of this compound new type, like we have done with `apple`, `banana` and `melon`.

Right at the end of the `struct` declaration, and before the ending semicolon, we can use the optional field `object_name` to directly declare objects of the structure type. For example, we can also declare the structure objects `apple`, `banana` and `melon` at the moment we define the data structure type this way:

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

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like `apple`, `banana` and `melon`) from a single structure type (`product`).

Once we have declared our three objects of a determined structure type (`apple`, `banana` and `melon`) we can operate directly with their members. To do that we use a dot (`.`) inserted 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:

```
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 `float`.

Let's see a real example where you can see how a structure type can be used in the same way as fundamental types:

```
// 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";
}
```

```
Enter title: Alien
Enter year: 1979

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

The example shows how we can use the members of an object 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`.

The objects `mine` and `yours` can also be treated as valid variables of type `movies_t`, for example we have passed them to the function `printmovie` as we would have done with regular variables. Therefore, one of the most important advantages of data structures is that we can either refer to their members individually or to the entire structure as a block with only one identifier.

Data structures are a feature that can be used to represent databases, especially if we consider the possibility of building arrays of them:

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

#define N_MOVIES 3

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

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<N_MOVIES; 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<N_MOVIES; 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: Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

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

Pointers to structures

Like any other type, structures can be pointed 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`. So, the following code would also be valid:

```
pmovie = &amovie;
```

The value of the pointer `pmovie` would be assigned to a reference to the object `amovie` (its memory address).

We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator (->):

```
// 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;
}
```

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

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

The previous code includes an important introduction: the arrow operator (->). This is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference. In the example we used:

```
pmovie->title
```

Which is for all purposes equivalent to:

```
(*pmovie).title
```

Both expressions `pmovie->title` and `(*pmovie).title` are valid and both mean that we are evaluating the member `title` of the data structure pointed by a pointer called `pmovie`. It must be clearly differentiated from:

```
*pmovie.title
```

which is equivalent to:

```
*(pmovie.title)
```

And that would access the value pointed by a hypothetical pointer member called `title` of the structure object `pmovie` (which in this case would not be a pointer). The following panel summarizes possible combinations of pointers and structure members:

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

Nesting structures

Structures can also be nested so that a valid element of a structure can also be in its turn 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 declaration we could use any of the following expressions:

```
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).