13. 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:

```
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.

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:

```
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.

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:

```
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.

Here is a real example with structure types in action:

```
// 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 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)
```

13.1 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>
#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 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)`

13.2 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
```

13.3 Type aliases (typedef / using)

A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

In C++, there are two syntaxes for creating such type aliases: The first, inherited from the C language, uses the typedef keyword:

```
typedef existing_type new_type_name ;
```

where existing_type is any type, either fundamental or compound, and new_type_name is an identifier with the new name given to the type.

For example:

```
typedef char C;
```

```
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

This defines four type aliases: C, WORD, pChar, and field as char, unsigned int, char* and char[50], respectively. Once these aliases are defined, they can be used in any declaration just like any other valid type:

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

More recently, a second syntax to define type aliases was introduced in the C++ language:

```
using new_type_name = existing_type ;
```

For example, the same type aliases as above could be defined as:

```
using C = char;
using WORD = unsigned int;
using pChar = char *;
using field = char [50];
```

Both aliases defined with typedef and aliases defined with using are semantically equivalent. The only difference being that typedef has certain limitations in the realm of templates that using has not. Therefore, using is more generic, although typedef has a longer history and is probably more common in existing code.

Note that neither typedef nor using create new distinct data types. They only create synonyms of existing types. That means that the type of myword above, declared with type WORD, can as well be considered of type unsigned int; it does not really matter, since both are actually referring to the same type.

Type aliases can be used to reduce the length of long or confusing type names, but they are most useful as tools to abstract programs from the underlying types they use. For example, by using an alias of int to refer to a particular kind of parameter instead of using int directly, it allows for the type to be easily replaced by long (or some other type) in a later version, without having to change every instance where it is used.

13.4 Unions

Unions allow one portion of memory to be accessed as different data types. Its declaration and use is similar to the one of structures, but its functionality is totally different:

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

This creates a new union type, identified by type_name, in which all its member elements occupy the same physical space in memory. The size of this type is the one of the largest member element. For example:

```
union mytypes_t {
  char c;
  int i;
  float f;
} mytypes;
```

declares an object (mytypes) with three members:

```
mytypes.c
mytypes.i
mytypes.f
```

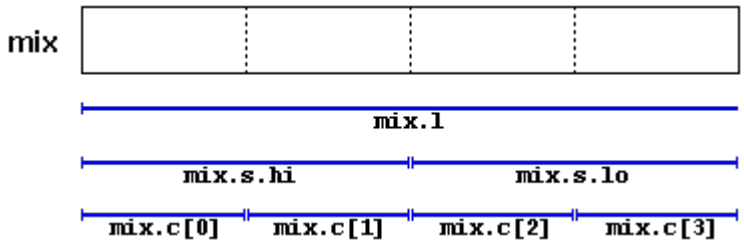
Each of these members is of a different data type. But since all of them are referring to the same location in memory, the modification of one of the members will affect the value of all of them. It is not possible to store different values in them in a way that each is independent of the others.

One of the uses of a union is to be able to access a value either in its entirety or as an array or structure of smaller elements. For example:

```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
```

```
    } s;
  char c[4];
} mix;
```

If we assume that the system where this program runs has an int type with a size of 4 bytes, and a short type of 2 bytes, the union defined above allows the access to the same group of 4 bytes: mix.l, mix.s and mix.c, and which we can use according to how we want to access these bytes: as if they were a single value of type int, or as if they were two values of type short, or as an array of char elements, respectively. The example mixes types, arrays, and structures in the union to demonstrate different ways to access the data. For a little-endian system, this union could be represented as:



The exact alignment and order of the members of a union in memory depends on the system, with the possibility of creating portability issues.

13.5 Anonymous unions

When unions are members of a class (or structure), they can be declared with no name. In this case, they become anonymous unions, and its members are directly accessible from objects by their member names. For example, see the differences between these two structure declarations:

structure with regular union	structure with anonymous union

`struct book1_t {`
`  char title[50];`
`  char author[50];`
`  union {`
`  float dollars;`
`  int yen;`
`  } price;`
`  } book1;`
	`struct book2_t {`
	`  char title[50];`
	`  char author[50];`
	`  union {`
	`  float dollars;`
	`  int yen;`
	`  };`
	`} book2;`

The only difference between the two types is that in the first one, the member union has a name (price), while in the second it has not. This affects the way to access members dollars and yen of an object of this type. For an object of the first type (with a regular union), it would be:

```
book1.price.dollars
book1.price.yen
```

whereas for an object of the second type (which has an anonymous union), it would be:

```
book2.dollars
book2.yen
```

Again, remember that because it is a member union (not a member structure), the members dollars and yen actually share the same memory location, so they cannot be used to store two different values simultaneously. The price can be set in dollars or in yen, but not in both simultaneously.

13.6 Enumerated types (enum)

Enumerated types are types that are defined with a set of custom identifiers, known as enumerators, as possible values. Objects of these enumerated types can take any of these enumerators as value.

Their syntax is:

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

This creates the type type_name, which can take any of value1, value2, value3, ... as value. Objects (variables) of this type can directly be instantiated as object_names.

For example, a new type of variable called colors_t could be defined to store colors with the following declaration:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Notice that this declaration includes no other type, neither fundamental nor compound, in its definition. To say it another way, somehow, this creates a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type color_t may take are the enumerators listed within braces. For example, once the colors_t enumerated type is declared, the following expressions will be valid:

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

Values of enumerated types declared with enum are implicitly convertible to an integer type, and vice versa. In fact, the elements of such an enum are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to or from. If it is not specified otherwise, the integer value equivalent to the first possible value is 0, the equivalent to the second is 1, to the third is 2, and so on... Therefore, in the data type colors_t defined above, black would be equivalent to 0, blue would be equivalent to 1, green to 2, and so on...

A specific integer value can be specified for any of the possible values in the enumerated type. And if the constant value that follows it is itself not given its own value, it is automatically assumed to be the same value plus one. For example:

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

In this case, the variable y2k of the enumerated type months_t can contain any of the 12 possible values that go from january to december and that are equivalent to the values between 1 and 12 (not between 0 and 11, since january has been made equal to 1).

The implicit conversion works both ways: a value of type months_t can be assigned a value of 1 (which would be equivalent to january), or an integer variable can be assigned a value of january (equivalent to 1).

13.7 Enumerated types with enum class

But, in C++, it is possible to create real enum types that are neither implicitly convertible to int and that neither have enumerator values of type int, but of the enum type itself, thus preserving type safety. They are declared with enum class (or enum struct) instead of just enum:

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Each of the enumerator values of an enum class type needs to be scoped into its type (this is actually also possible with enum types, but it is only optional). For example:

```
Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```

Enumerated types declared with enum class also have more control over their underlying type; it may be any integral data type, such as char, short or unsigned int, which essentially serves to determine the size of the type. This is specified by a colon and the underlying type following the enumerated type. For example:

```
enum class EyeColor : char {blue, green, brown};
```

Here, Eyecolor is a distinct type with the same size of a char (1 byte).