Chapter 6: Structures

Introduction

Thus far you have worked with variable's whose data types are very simple: they are a numbers of either integer or floating-point format with a specific range. These types of variables, who only have a single value to their name, are known as *basic variables* or *primitives*. Everything in computers is built on numbers, but not necessarily singular types. Sometimes it's advantageous to group common variables into a single collection. For example a date would require a day, month, and year. From what I've currently explained, you could create three separate variables for each, like so:

```
int day, month, year;
```

This isn't so bad, but what happens if you want to store two dates and not one? You'd have to create three more variables and give them unique names:

```
int day1, month1, year1;
int day2, month1, year2;
```

This begins to become a hassle. Not only do you have to create many variables, but you have to keep giving them unique names. C++ provides a way to collect similar variables into a single *structure*.

Structures

The term *structure* in C++ is a bit ambiguous; it means both a user-defined type which is a grouping of variables as well as meaning a variable based on a user-defined structure type. For the purpose of distinction I will refer to the user-defined type side as *structure definition* and the variable side as *structure variable*.

A structure definition is a user-defined variable type which is a grouping of one or more variables. The type itself has a name, just like '`int`', '`double`', or '`char`' but it is defined by the user and follows the normal rules of identifiers. Once the type has been defined through the C++ '`struc`t' keyword, you can create variables from it just like you would any other type.

Since a structure definition is a grouping of several types: it is a group of one or more variables. These are known as *elements* or *member variables* as they are *members* of the structure definition they are part of. It is like you, your friend 'Bob', and your brother 'Mike' being members of a Girl Watching Club. That single *group* name would be the structure definition that refers to *all* of you, whom are members. Following through with

our hinted example, a structure definition could be a ‘`date`’ which might be made up of three ‘`int`’ member variables: ‘`day`’, ‘`month`’, and ‘`year`’.

Before creating a structure variable you must create a structure definition. This is a blue print for the compiler that is used each time you create a structure variable of this type. The structure definition is a listing of all member variables with their types and names.

When you create a structure variable based on a structure definition, all of the member variables names are retained. The only name you have to give is that of the new structure variable. The element names within that variable will be the same as in the structure type. If you create two structure variables from ‘data’, both will have all three member variables with the same name in both: ‘day’, ‘month’, and ‘year’.

Member variables are distinguished by the structure variable they are part of. You wouldn’t simply be able to use ‘day’ by itself; instead you’d have to refer to both the structure variable’s name (that you gave when you created it) as well as ‘day’.

Defining Structures

Defining a structure is giving the compiler a blue print for creating your type. Think of it as a list of people invited to a party. When you create a variable based on the structure definition, all of the member variables are created automatically and grouped under the name you gave. Note that when you create a variable of any kind, you must give it a unique name that is different than its type. Below is the syntax for a structure definition:

```
struct tag
{
    members;
};
```

Writing a structure definition begins with the word ‘struct’ followed by the type-to-be, and ended with a structure block that is ultimately terminated with a semi-colon. Do not forget this trailing semi-colon when defining a structure!

The name of a structure definition is known as the structure *tag*. This will be the name of the type that you create, like ‘int’ or ‘float’. It is the type that you will specify when creating a structure variable.

This structure block is remarkably similar to a statement block since it starts and ends with curly braces. But don’t forget that it *ultimately* ends with a semi-colon. Within the structure block you declare all the member variables you want associated with that type. Declare them as you would normal variables, but *do not try to initialize them*. This is simply a data blue print, it is not logic or instructions and the compiler does not *execute* it.

Remember the party member list I told you about earlier. The people on the list aren't actually *in* the list; it is simply mentioning that they *will* be there when the party happens. Likewise the *data members* (synonym for member variables) of a structure won't actually be created until a variable *based* on the structure is created. Technically an '`int`' is just this as well. It's a description of a storage unit. That storage unit isn't reserved until you create a variable with it.

Date Definition

The follow defines a structure called '`date`' which contains three '`int`' member variables: '`day`', '`month`', and '`year`':

```
struct date
{
   int day;
   int month;
   int year;
};
```

If you've read the variables chapter, you'd know we can shorten this definition to the following:

```
struct date
{
    int day, month, year;
};
```

Remember that you *cannot* initialize member variables in a structure definition. The following is *wrong* and will *not* compile:

```
struct date
{
    int day = 24, month = 10, year = 2001;
};
```

Definition Scope

Now we know *how* to define a structure, but *where* should you do it? A structure definition has the same type of scoping as a variable. If you define a structure in a function, you will only be able to use it there. If you define it in a nested statement block, you will only be able to use it inside there and any statement blocks nested within it. But the most common place is defining it globally, as in outside of any functions or blocks. Typically, you'll want to be able to create variables of the defined structure anywhere in your program, so the definition will go at the top. The following is an empty program that defines a '`date`' structure globally:

```
01  #include <iostream.h>
02
03  struct date
04  {
05      int day, month, year;
06  };
07
08  int main()
09  {
10      return 0;
11  }
```

Both structure types and variables follow the same scope as normal variables, as do *all identifiers*. If you define a structure within a function, then you can only use it within that function. Likewise if you define a structure outside of any function then it can be used in any place throughout your program.

Creating Structure Variables

Once you have defined a structure you can create a variable from it just as you would any other variable. In the case of '`date`' we could create a structure variable of this type by specifying that followed by a name:

```
date birthday;
```

C Note: In C you would have to specify '`struct`' before a type that was based on a structure. Luckily, they got rid of this nuisance in C++.

The above would create a variable called '`birthday`' whose type is the structure '`date`'. The variable contains three parts: '`day`', '`month`', and '`year`'. What this actually does is set aside a whole block of memory that can contain all of the member variables. Each member variable then occupies a chunk of it for their individual storage units. The member variables are not actually created one at a time.

Storage for member variables exist at some *offset* from the beginning of the glob of memory reserved by the entire structure. For example, in our 'date' structure variable the first member variable is 'day' so it exists at offset 0. The next member variable, 'month' in this case', will exist at the next available offset. If 'day' uses 4 bytes (32 bits), then 'month' will be at offset 4:

<picture of date broken up into three pieces>

This is an important concept to remember as you will see later on.

Initializing Structure Variables

You cannot initialize member variables in the structure definition. This is because that definition is only a map, or plan, of what a variable based on this type will be made of. You can, however, initialize the member variables of a structure *variable*. That is, when you create a variable based on your structure definition you can pass each member variable an initializer.

To initialize a structure variable's members, you follow the original declaration, minus the semi-colon, with the assignment operator (=). Next you define an initialization block which is a list of initializers separated by commas and enclosed in curly braces. Lastly, you end it with a semi-colon. These values are assigned to member variables in the order that they occur. Let's look at an example:

```
date nco_birthday = { 19, 8, 1979 };
```

This creates a variable called '`nco_birthday`' and initializes it to a list of values. The values are assigned to the member variables in the order they are declared in the structure definition. Remember what I mentioned about each member variable having an offset. The same order in which each member is given an offset is the order in which each is assigned a value in an initialization. So the first initializer is used to initialize the first member variable in the structure, next is the second, and so on and so forth. This order of initialization continues until the values run out.

So, in our structure '`date`', the member variable '`day`' is first. Therefore the left-most value, '19', will be assigned to '`day`'. Likewise '8' is assigned to '`month`' and '1979' is assigned to '`year`'.

If you try to assign more values than are member variables, you will get a compiler error. However, you can assign *fewer* values than there are member variables. If there are no more values to be assigned, the assignment will simply end. For example, if we had omitted the last value, '1979', then no value would be assigned to '`year`'.

So far I've only used numeric literals as initializers. It is possible to use any expression that you normally would. But remember that the expression must result in a value. Here is an example of initialization with things other than literals:

```
int myday = 19;
int mymonth = 5;
date nco_birthday = { myday, mymonth + 3, 2001 - 22 };
```

Although you can assign a value to a variable in the same way you initialize it, the same is not true with structures. So while this works:

```
int x;
x = 0;
```

This doesn't:

```
date nco_birthday;
nco_birthday = { 19, 8, 1979 };
```

Assigning values to multiple members of a structure variable is only possible when that variable is first created. Once a structure variable has been declared, you must access each member individually.

Accessing Members

There's not much you can do with the structure itself; much of your time with structure variables will be spent using its members. You can use a member variable in any place you'd use a normal variable, but you must specify it by the structure variable's name *as well as* the member variable's name using the member operator.

C Note: The official is *structure* member operator in C, but that part was removed in C++ because structures aren't the only types with members (or that use this operator).

Consider the situation where you have two structure variables created from one definition; like 'nco_birthday' and 'mdp_birthday' both created from 'date'. How do you use the member variable 'day'? What if there is already a local variable called 'day'; how are the two distinguished? The answer is of course by specifying the structure variable as well as its member. You must specify that which possesses what you want and the language assumes nothing.

To specify that you want a member of a specific structure variable, you use the structure member operator which is the period (also known as a "dot"). Simply use the structure's name, follow with the period, and end with the member:

```
structure.member
```

The following creates 'nco_birthday' from 'date' and then assigns each member a value:

```
date nco_birthday;
nco_birthday.day = 19;
nco_birthday.month = 8;
nco_birthday.year = 1979;
```

In this way a member variable can be used in any place that a normal variable is, including mathematic operations, user input, and output. Consider the following program:

```
01  #include <iostream.h>
02
03  struct date
04  {
```

```
05      int day, month, year;
06  };
07
08  int main()
09  {
10      date birth;
11      cout << "Enter your birth date!" << endl;
12      cout << "Year: ";
13      cin >> birth.year;
14      cout << "Month: ";
15      cin >> birth.month;
16      cout << "Day: ";
17      cin >> birth.day;
18
19      cout << "You entered " << birth.month << "/"
20           << birth.day << "/" << birth.year << endl;
21
22      cout << "You were born in the "
23           << ((birth.year / 100) + 1) << "th Century!"
24           << endl;
25      return 0;
26  }
```

This program demonstrates using member variables for user input, output, and mathematical operations. Notices that they are used like you would any other variable, but the structure must be specified each time. Here is the output of this program when I use my birth date as input:

```
Enter your birth date!
Year: 1979
Month: 8
Day: 19
You entered 8/19/1979
You were born in the 20th Century!
```

Likewise you can assign values between member variables and normal variables:

```
date birth = { 19, 8, 0 };
int day = birth.day, year = 1979, month;
birth.year = year;
month = birth.month;
```

Just remember that a member variable by itself is *exactly* like a normal variable of the same type. It can be used in any of the same situations.

Pencil-Box Analogy

Here's an analogy for a structure definition and structure variables. The structure definition would be a mold for a pencil-box. A structure variable based on this definition would be the pencil-box itself. Inside this pencil-box are three compartments: one for pencils, one for erasers, and one for miscellaneous items. In code we might see this as:

```
struct pencilbox_mold
{
    long pencils;
    short erasers;
    float misc;
};

pencilbox_mold pencilbox;
```

The mold itself can't be used for storing anything; it can simply be used to make pencil boxes. Those pencil-boxes, then, *can* store the things they were molded to. In this case the pencil-boxes would store pencils, erasers, and miscellaneous items.

Anytime you need to access one of these members, you must go through the structure variable. Consider the situation where you have two pencil-boxes: one for Jimmy and one for Jamie Sue. You want to take a pencil, but you must decide on one:

```
pencilbox_mold Jimmys;
pencilbox_mold JamieSues;
```

You could take a pencil from Jimmy's pencil box:

```
Jimmys.pencils--;
```

Or from Jamie Sue's:

```
JamieSues.pencils--;
```

You could even give all of Jimmy's pencils to Jamie Sue:

```
JamieSues.pencils = Jimmys.pencils;
Jimmys.pencils = 0;
```

You can't do anything with the pencil-box mold as far as pencils, because it doesn't have any. It is only used for creating the pencil-boxes; it isn't used to contain anything of its own. The following wouldn't work:

```
pencilbox_mold.pencils--;
```

Blarg.

Variables with Definition

The syntax of '`struct`' also allows you to create variables based on a structure definition without using two separate statements:

```
struct tag
```

```
{
    member(s);
} variable;
```

The structure variables created in this way will have the same scope as their structure definition. This is a nice thing to use when you want to group some variables in one place in your program without it affecting other things. You can create a structure definition as well as variables from it in that one local place and not have to ever use it again. The following creates a 'point' variable right after the 'pointtag' structure is defined:

```
struct pointtag
{
    int x, y;
} point;
```

In this, 'point' is a variable just as if we had declared it separately. In fact, this statement is identical to:

```
struct pointtag
{
    int x, y;
};
pointtag point;
```

Rather than defining a structure under a name and then creating variables which refer to the named definition, the definition becomes part of the variable declaration. It is even possible to omit the structure tag which may make more sense in this situation:

```
struct
{
    int x, y;
} point;
```

The above creates a structure variable called 'point' which has two member variables. Because the structure definition is not named, it cannot be used elsewhere to create variables of the same structure type. However, like in any variable declaration, you can create multiple variables of the same type by separating them with commas:

```
struct
{
    int x, y;
} point1, point2;
```

Now we have two structure variables of the same nameless structure type. Even more fun we can initialize these structure variables as we normally would:

```
struct
{
    int x, y;
} point1 = { 0, 0}, point2 = {0, 0};
```

This creates the two structure variables, ‘point1’ and ‘point2’, and initializes both members, ‘x’ and ‘y’, on each to zero (0). You don’t need to make a name-less definition to do this. You could still have a structure tag *and* initialize the variables created after the definition.

I won’t require you to remember all the strange things in this section. It is mainly here as a reference should you want to know if certain things are possible. But before I go on, there’s some positively useless crazy stuff you can do. Not only can you write name-less structures, but you can write them without declaring any variables. This will cause a warning on smarter compilers, but is perfectly legal:

```
struct
{
    int x, y;
};
```

The above structure definition can’t be used to create any variables because it has no name. Likewise, no variables are declared after it either. So this is a nice way to fill up your programs with useless source code. But why stop there? You could make an empty, name-less definition:

```
struct { };
```

Definition vs Declaration

When you place a member variable inside a structure definition, it is known as declaring that member variable. When you create a variable of any kind it is known as variable declaration, but in creating variables you are also defining them. The difference between the words “declare” and “define”, as they pertain to C++ and programming in general, are often misunderstood.

If you were at a meeting and there was a roll call, all of the members of the meeting would be required to declare themselves. However, all you are hearing is their names and that they are present. For you to know more, they would have to be *defined* or described. All declared things must eventually be defined if they are to be used. A declaration is an *acknowledgement* of existence, whereas a definition is a *complete description* of an entity. In programming terms, a definition of an entity is linked to something.

The entity can be anything from a variable to a type. Now that you have seen structures, you have dealt with both definition and declaration. The member variables of a structure definition are *declared* not defined. They cannot be used until they have been defined, which is after a structure variable is created, based on the definition.

The structure definition itself *is* truly a definition. It can be used to create variables. Since you can use it, it has been defined. Even though a structure definition is not linked to any memory, as a variable is, it is still a definition because it completely defines a type. A separate, but similar statement for acknowledging the existence of a structure is a structure declaration. This includes only the 'struct' keyword, followed by the name of the structure type, and ended with a semi-colon:

```
struct tag;
```

Before a structure type can be used, though, it must be defined. This means, if you declare a structure, as above, you must also define it before it can be used:

```
struct tag;     // declare structure
struct tag      // define structure
{
    members;
};
```

A structure declaration can *not* be used to create variables. A structure *delaration* can not be used at all except to acknowledge the presence of the structure. This will become useful to you in more complex programs. You won't see any plain structure declarations in this part of the book however, because I don't want to confuse you. The important point of this section, that I will reiterate one more time, is that to use something it must be defined, not just declared.

When you write a variable declaration statement, the variable is usually defined as well and thus created. However, it is possible to write a variable declaration statement that only *declares* the variable, and does not define it. We have not touched on this yet, though. For a variable to be fully defined, it must be linked to a place in memory where its value can be stored; thus making it usable. References are defined when they are declared, because they are initialized to be aliases of already-defined variables. Since they are shades of already defined things, they too are defined.

Structure References

References to structure variables, work the same way as normal references. To create one you would write out the name of the structure type, followed by the reference name which is preceded by the reference operator (`&`):

```
struct_name &reference;
```

The following creates a structure variable based on 'date' and a reference to it:

```
date birth;
date &mybirth = birth;
```

Both of these, ‘birth’ and ‘mybirth’, would have access to the same member variables and their values. Thus they can be used interchangeably:

```
birth.year = 1981;
mybirth.year -= 2;
```

Can you guess what the value of ‘birth.year’ would be from the above? It would be ‘1979’. The reference ‘mybirth’ is just an alias to ‘birth’ and its group of member variables. Remember that a reference is not a real variable (it has no value), but simply a nickname for another.

Structure Pointers

Utilizing pointers with structures, unfortunately, adds a previously unseen complexity. A pointer to a variable cannot be used to modify the variable’s value until it has been dereferenced. This case is no different from structures. But it affects the way you access the structure variable’s members. Recall that a pointer simply contains a memory address. The pointer knows nothing of what this memory address is used for, which is why you have to dereference a pointer to a specific type. Thus, you cannot access a structure variable’s members until you dereference a pointer to the structure type:

```
date birth;
date *p = &birth;
(*p).year = 1979;
```

The pointer ‘p’ above, had to be dereferenced before the member variable ‘year’ could be accessed through it. It was surrounded in parenthesis because the indirection operator (asterisk ‘*’) has a lower precedence than the member operator. Thus the following would not work:

```
*p.year = 1979;
```

This would be seen as “get the value pointed to by ‘`p.year`’” and ‘`p.year`’ is an invalid identifier; hence the parenthesis around the indirection operation. This method of accessing a structure variable’s members is cumbersome and requires two operations simply to get at the member variable: indirection and then member. As you progress in your C++ programming, you will undoubtedly encounter pointers more and more often. For this purpose, there is an operator specifically for accessing a structure variable’s members through a pointer. This is a member operator known specifically as a pointer-to-member operator or a dash followed by a greater than sign ‘`->`’ (also known as an “arrow”):

```
p->year = 1979;
```

This operator only works on pointers to structure variables. You cannot use it on normal structure variables for members. The following would not work:

```
birth->year = 1979;
```

The left operand *must* be a pointer to a variable with members, and the right operand must be the name of a member variable within that.

Structure Members

A structure definition contains multiple variables, but not necessarily just primitives. You can define a structure to have *structure member variables*. Consider the following:

```
struct moment
{
    date theDate;
    time theTime;
};
```

This assumes that ‘date’ and ‘time’ are structures that have already been *declared*. These structure members can then be accessed as normal member variables, but then *their* variables must be accessed as well. If I had a variable based on ‘moment’ called ‘birth’, then I would need to write the following to assign ‘19’ to “birth’s date’s day”:

```
birth.theDate.day = 19;
```

The member operator is like possession in that way. Remember how C++ syntax is like English syntax with different rules. To indicate possession in English you write an apostrophe followed by an “s”. In C++ you simply have to write a period (dot).

A structure only has to have been declared before it can be used in another structure’s definition. The following is perfectly acceptable:

```
struct date;
struct time;
struct moment
{
    date theDate;
    time theTime;
};
struct date
{
    int day, month, year;
};
struct time
{
    int sec, min, hour;
};
```

To be able to define a structure you only must know the types and the names of the member variables declared inside. With the above we declare the structures ‘date’ and

‘time’ but do not define them until later. This simply acknowledges that they exist and they can therefore be used within ‘moment’.

What if I hadn’t defined ‘date’ and ‘time’? It would still be legal, but then I would not be able to *use* ‘moment’ at all. Why? Since ‘date’ or ‘time’ have not been defined, the compiler does not know how big they are supposed to be or what kind of data they contain. You couldn’t then create a variable based on ‘moment’ because the compiler doesn’t know how big of a memory block to allocate. Likewise if you try to use a structure that has been declared *before* it has been defined, you will encounter the same problem. Consider the following:

```
01  #include <iostream.h>
02
03  struct date;
04
05  int main()
06  {
07      date birth;
08      struct date { int day, month, year; };
09      return 0;
10  }
```

The above would not work because ‘date’ is being used prior to its definition. To fix this program, we would simply have to swap lines 7 and 8.

Defining Structure in Structure

It is possible to define a structure inside a structure definition and create variables from it at the same time. For example:

```
struct moment
{
    struct date
    {
        int day, month, year;
    } theDate;

    struct time
    {
        int second, minute, hour;
    } theTime;
};
```

The drawback of the above is that the ‘date’ and ‘time’ definitions cannot be used elsewhere without also referring to the parent structure. This is a sticky situation that we won’t explore more until *classes*. For now you can think of the tags ‘date’ and ‘time’ existing soley for the purpose of this definition; i.e. they could not be used elsewhere to create new variables.

If the ‘date’ definition isn’t going to be used elsewhere anyway, the structure tag can simply be omitted. Thus we could remove the ‘date’ and ‘time’ structure type identifiers and it would work fine. You can write any number of variables in a structure definition and a valid variable declaration statement (minus initialization of course) is valid inside a structure definition.

Let’s say we want to be able to use ‘date’ elsewhere, but not ‘time’. The following program demonstrates how the structure definitions would be written as well as uses the defined structure types in an extended “birth date” sample:

```
01  #include <iostream.h>
02
03  struct date { int day, month, year; } ;
04
05  struct moment
06  {
07      date theDate;
08      struct
09      {
10          int sec, min, hour;
11      } theTime;
12  };
13
14  int main()
15  {
16      moment birth;
17      cout << "Enter your birth moment!" << endl;
18      cout << "Year: ";
19      cin >> birth.theDate.year;
20      cout << "Month: ";
21      cin >> birth.theDate.month;
22      cout << "Day: ";
23      cin >> birth.theDate.day;
24      cout << "Hour (military): ";
25      cin >> birth.theTime.hour;
26      cout << "Minute: ";
27      cin >> birth.theTime.min;
28      cout << "Second: ";
29      cin >> birth.theTime.sec;
30
31      cout << "You entered " << birth.theDate.month << "/"
32           << birth.theDate.day << "/" << birth.theDate.year
33           << " @ " << birth.theDate.hour << ":"
34           << birth.theDate.min << ":" << birth.theDate.sec
35           << endl;
36
37      if (birth.theTime.hour > 20 || birth.theTime.hour < 8)
38          cout << "You were born early in the morning!"
39               << endl;
40
41      return 0;
42  }
```

Any number of structure definitions can be nested; you can get extremely complex with structures, which is why they are sometimes known as *complex types*.

Bit-Fields

Structures can have member variables of any data type … and beyond. They have the unique ability to have member variables that are precise *bit-fields*. A bit field is a specific number of bits that stores an unsigned or signed integer number. Unlike variable data types which use a specific number of bits, you can actually define how many. A bit-field acts like a normal integer variable, but has a storage capacity limited to the number of bits you assign to it.

To create a bit-field member variable you designate either 'signed' or 'unsigned', followed by the name, then a colon, and lastly the number of bits:

signed/unsigned name : number_of_bits;

For example, our date structure has 'day', 'month', and 'year' variables. Each of those takes up at least two (2) bytes, probably four (4). The 'day' member only needs to store up to the number value thirty-one (31); thus it will only ever use five (5) bits (five bits can store the values 0 to 31). The 'month' member only needs four (4) bits (0 to 15) to store the values one (1) to twelve (12). We could define these as bit-fields like so:

unsigned day : 5;
unsigned month : 4;

Example date structure using bit-fields:

```
struct date
{
    unsigned day : 5;
    unsigned month : 4;
    unsigned year : 15;
};
```

This is yet to be done.

Unions

Another type of complex type that looks like a structure is a 'union'. You declare, define, and use unions exactly the same as structures: simply replace 'struct' with 'union' and follow all the same usage rules. The difference is not in the usage, but how they are stored in memory. A structure is a big glob of memory that is partitioned out to each member. A union, on the other hand, is a small glob of memory and each member shares

the same memory space! Basically a union is a name for a single memory cell (one or more bytes) that is accessed via one or more variables of any type:

```
union
{
   int i;
   double n;
} x;
```

The above is a union variable named ‘x’ with two members. Both of these occupy the same memory, but may not use the same amount of bytes. The size of a union variable is determined by the largest member in the union. If you use one of the smaller members, some of the memory occupied by the union variable will not get touched.

Getting back to ‘x’, we might use it like so:

```
x.i = 10;
x.n = 155.678;
```

Since both members use the same memory space, changing the ‘n’ member also mucks with the memory used by ‘i’ (thus changing it as well). Members in a union can be any kind of variable and their memory usage might differ. An integer (such as ‘i’) is not stored in memory the same as a floating point (such as ‘n’). The results of utilizing both members at the same time might be quite unexpected!

Author’s Preference: The members of a union all use the same space which means once you choose which one to use; you should use that one exclusively until you clear the memory. Certain variable types don’t mix with one another!

Unions are most useful in more advanced programs where you want a variable, but you’re not sure what kind until later. You use less space by declaring one instead of two. And this works out when you only end up using one. However, one immediately useful trait in unions is for a bit-field that can be accessed by individual bits or through a single variable.

Getting back to our date structure that is scrunched into bit-fields, you could put that structure in part of a union:

```
union
{
    long i;
    date d;
} mydate;
```

Now, you can access the individual bit-fields through the ‘`date`’ member of ‘`mydate`’, or the entire variable value through ‘`i`’. Thus you can reset all the bit-fields like this:

```
mydate.i = 0;
```

And then set them individually as well:

```
mydate.d.day = 19;
mydate.d.month = 8;
mydate.d.year = 1979;
```

The size of '`mydate`' will be four (4) bytes because of the member '`i`' which needs that many. This means that one byte will be wasted because the 'date' structure only uses three (3).

Structure Padding

If you use the 'sizeof' keyword on a structure variable, it may yield a size different than the sum of the members' sizes. This is due to structure *padding* based on *alignment*. The term *alignment* refers to aligning a structure on certain memory boundaries: like every 4, 8, 16 or other amount of bytes. This means that if a structure uses less memory than a multiple of the current alignment, it will be padded with unused bytes.

The exact structure alignment depends on your compiler and can usually be set when building your programs. A smaller alignment means less memory used but a sacrificing a hit in speed (due to how memory is accessed); the opposite is then true with a bigger alignment.