

Structures

Objectives:

This section contains some very advanced but important features of the C programming language.

Having read this section you should be able to:

1. program using a structure rather than several arrays.
2. how pointer can be used in combination with structures to form linked list.

Structures:

The array is an example of a data structure. It takes simple data types like int, char or double and organises them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of the same type. At first this seems perfectly reasonable. After all why would you want an array to be composed of twenty chars and two ints? Well this sort of mixture of data types working together is one of the most familiar of data structures. Consider for a moment a record card which records name, age and salary. The name would have to be stored as a string, i.e. an array of chars terminated with an ASCII null character, and the age and salary could be ints.

At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C language provides struct. At first it is easier to think of this as a record - although it's a little more versatile than this suggests.

Defining A New Type:

Declaring a struct is a two-stage process. The first stage defines a new data type that has the required structure which can then be used to declare as many variables with the same structure as required. This two-stage process is often confusing at first - especially as it results in the need to think up multiple names with the same general meaning - but it really is quite simple. For example, suppose we need to store a name, age and salary as a single structure. You would first define the new data type using:

```
struct emprec
{
    char name[25];
    int age;
    int pay;
};
```

and then you would declare a new variable:

```
struct emprec employee
```

Notice that the new variable is called employee and it is of type emprec which has been defined earlier. You see what we mean about duplicating names - emprec is the name of the general employee record structure and employee is a particular example of this general type. It might help to compare the situation with that of a general int type and a particular int variable such as count - emprec is a type like int and employee is a variable like count. You can see that in general you can define a structure using:

```
struct name
{
    list of component variables
};
```

and you can have as long a list of component variables as you need. Once defined you can declare as many examples of the new type as you like using:

```
struct name list of variables;
```

For example:

```
struct emprec employee, oldemploy, newemploy;
```

and so on. If you want you can also declare a structure variable within the type definition by writing its name before the final semi-colon. For example:

```
struct emprec
{
    char name[25];
    int age;
    int pay;
} employee;
```

defines the structure and declares a structure variable called employee. The only trouble with this form is that not many C programmers use it and many will even think that it is an error! So how do we use a struct?

When you first start working with arrays it seems obvious that you access the individual elements of the array using an index as in `a[i]` for the *i*th element of the array, but how to get at the individual components of a structure? The answer is that you have to use qualified names. You first give the name of the structure variable and then the name of the component separated by a dot. For example, given:

```
struct emprec employee
```

then:

```
employee.age
```

is an int and:

```
employee.name
```

is a char array. Once you have used a qualified name to get down to the level of a component then it behaves like a normal variable of the type. For example:

```
employee.age=32;
```

is a valid assignment to an int and:

```
employee.name[2] = 'X';
```

is a valid assignment to an element of the char array. Notice that the qualified name uses the structure variable name and not the structure type name. You can also define a structure that includes another structure as a component and of course that structure can contain another structure and so on. In this case you simply use the name of each structure in turn, separated by dots, until you reach a final component that isn't a structure. For example, if you declare a struct firm which includes a component employee which is an emprec then:

```
firm.employee.age
```

is an int. You may be feeling a little disappointed at the way in which structures are used. When you first meet arrays it is obvious how useful they are because the array index is an integer which can be used within a loop to process vast amounts of data in a few lines of code. When you first meet the struct it just doesn't have the same obvious advantages. Because you have to write out a full qualified name to get at each of the components of the struct you can't automate the processing in the same way. However this is reasonable enough when you remember that each component of a struct can be a different data type! The point is that the value of a struct is different to that of an array. A struct can be used to wrap up a group of variables which form a coherent entity.

For example, C has no facilities for manipulating complex numbers but this is easy enough to put right using a struct and a few functions. A complex number is composed of two parts - a real and imaginary part - which can be implemented as single or double precision values. This suggests defining a new struct type:

```
struct comp
{
    float real;
    float imag;
};
```

After this you can declare new complex variables using something like:

```
struct comp a,b;
```

The new complex variables cannot be used as if they were simple variables - because they are not. Most versions, of the C language do allow you to assign structures so you could write:

```
a=b;
```

as shorthand for

```
a.real=b.real;
a.imag=b.imag;
```

Being able to assign structures is even more useful when they are bigger. However you can't expect C to sort out what you mean by $c = a + b$ - for this you have to write out the rule for addition as:

```
c.real=a.real+b.real;
c.imag=a.imag+b.imag;
```

Structures and Functions:

Of course a sensible alternative to writing out the addition each time is to define a function to do the same job - but this raises the question of passing structures as parameters. Fortunately this isn't a big problem. Most C compilers, will allow you to pass entire structures as parameters and return entire structures. As with all C parameters structures are passed by value and so if you want to allow a function to alter a parameter you have to remember to pass a pointer to a struct. Given that you can pass and return structs the function is fairly easy:

```
struct comp add(struct comp a , struct comp b)
{
    struct comp c;
    c.real=a.real+b.real;
    c.imag=a.imag+ b.imag;
    return c;
}
```

After you have defined the add function you can write a complex addition as:

```
x=add(y,z)
```

which isn't too far from the $x=y+z$ that you would really like to use. Finally notice that passing a struct by value might use up rather a lot of memory as a complete copy of the structure is made for the function.

Pointers to Structures:

You can define a pointer to a structure in the same way as any pointer to any type. For example:

```
struct emprec *ptr
```

defines a pointer to an emprec. You can use a pointer to a struct in more or less the same way as any pointer but the use of qualified names makes it look slightly different For example:

```
(*ptr).age
```

is the age component of the emprec structure that ptr points at - i.e. an int. You need the brackets because '!' has a higher priority than '*'. The use of a pointer to a struct is so common, and the pointer notation so ugly, that there is an equivalent and more elegant way of writing the same thing. You can use:

```
prt->age
```

to mean the same thing as (*ptr).age. The notation gives a clearer idea of what is going on - prt points (i.e. ->) to the structure and .age picks out which component of the structure we want. Interestingly until C++ became popular the -> notation was relatively rare and given that many C text books hardly mentioned it this confused many experienced C programmers!

There are many reasons for using a pointer to a struct but one is to make two way communication possible within functions. For example, an alternative way of writing the complex number addition function is:

```
void comp add(struct comp *a , struct comp *b , struct comp *c)
{
  c->real=a->real+b->real;
  c->imag=a->imag+b->imag;
}
```

In this case c is now a pointer to a comp struct and the function would be used as:

```
add(&x,&y,&z);
```

Notice that in this case the address of each of the structures is passed rather than a complete copy of the structure - hence the saving in space. Also notice that the function can now change the values of x, y and z if it wants to. It's up to you to decide if this is a good thing or not!

Malloc:

Now we come to a topic that is perhaps potentially the most confusing. So far we have allowed the C compiler to work out how to allocate storage. For example when you declare a variable:

```
int a;
```

the compiler sorts out how to set aside some memory to store the integer. More impressive is the way that

```
int a[50]
```

sets aside enough storage for 50 ints and sets the name a to point to the first element. Clever though this may be it is just static storage. That is the storage is allocated by the compiler before the program is run - but what can you do if you need or want to create new variables as your program is running? The answer is to use pointers and the malloc function. The statement:

```
ptr=malloc(size);
```

reserves size bytes of storage and sets the pointer ptr to point to the start of it. This sounds excessively primitive - who wants a few bytes of storage and a pointer to it? You can make malloc look a little more appealing with a few cosmetic changes. The first is that you can use the sizeof function to allocate storage in multiples of a given type. For example:

```
sizeof(int)
```

returns a number that specifies the number of bytes needed to store an int. Using sizeof you can allocate storage using malloc as:

```
ptr= malloc(sizeof(int)*N)
```

where N is the number of ints you want to create. The only problem is what does ptr point at? The compiler needs to know what the pointer points at so that it can do pointer arithmetic correctly. In other words, the compiler can only interpret ptr++ or ptr=ptr+1 as an instruction to move on to the next int if it knows that the ptr is a pointer to an int. This works as long as you define the ptr to be a pointer to the type of variable that you want to work with. Unfortunately this raises the question of

how malloc knows what the type of the pointer variable is - unfortunately it doesn't.

To solve this problem you can use a TYPE cast. This C play on words is a mechanism to force a value to a specific type. All you have to do is write the TYPE specifier in brackets before the value. So:

```
ptr = (*int) malloc(sizeof(int)*N)
```

forces the value returned by malloc to be a pointer to int. Now you can see how a simple idea ends up looking complicated. OK, so now we can acquire some memory while the program is running, but how can we use it? There are some simple ways of using it and some very subtle mistakes that you can make in trying to use it! For example, suppose during a program you suddenly decide that you need an int array with 50 elements. You didn't know this before the program started, perhaps because the information has just been typed in by the user. The easiest solution is to use:

```
int *ptr;
```

and then later on:

```
ptr = (*int) malloc(sizeof(int)*N)
```

where N is the number of elements that you need. After this definition you can use ptr as if it was a conventional array. For example:

```
ptr[i]
```

is the ith element of the array. The trap waiting for you to make a mistake is when you need a few more elements of the array. You can't simply use malloc again to get the extra elements because the block of memory that the next malloc allocates isn't necessarily next to the last lot. In other words, it might not simply tag on to the end of the first array and any assumption that it does might end in the program simply overwriting areas of memory that it doesn't own.

Another fun error that you are not protected against is losing an area of memory. If you use malloc to reserve memory it is vital that you don't lose the pointer to it. If you do then that particular chunk of memory isn't available for your program to use until it is restarted.

Structures and Linked Lists:

You may be wondering why malloc has been introduced right after the structure. The answer is that the dynamic allocation of memory and the struct go together a bit like the array and the for loop. The best way to explain how this all fits together is via a simple example. You can use malloc to create as many variables as you want as the program runs, but how do you keep track of them? For every new variable you create you also need an extra pointer to keep track of it. The solution to this otherwise tricky problem is to define a struct which has a pointer as one of its components. For example:

```
struct list
{
  int data;
  struct list *ptr;
};
```

This defines a structure which contains a single int and - something that looks almost paradoxical - a pointer to the structure that is being defined. All you really need to know is that this is reasonable and it works. Now if you use malloc to create a new struct you also automatically get a new pointer to the struct. The final part of the solution is how to make use of the pointers. If you start off with a single 'starter' pointer to the struct you can create the first new struct using malloc as:

```
struct list *star;
start = (*struct list) malloc(sizeof(list))
```

After this start points to the first and only example of the struct. You can store data in the struct using statements like:

```
start->data=value;
```

The next step is to create a second example of the struct:

```
start = (*struct list) malloc(sizeof(list));
```

This does indeed give us a new struct but we have now lost the original because the pointer to it has been overwritten by the pointer to the new struct. To avoid losing the original the simplest solution is to use:

```
struct list *start,newitem;  
newitem = (*struct list) malloc(sizeof(list));  
start->prt=start;  
start=newitem;
```

This stores the location of the new struct in newitem. Then it stores the pointer to the existing struct into the newitem's pointer and sets the start of the list to be the newitem. Finally the start of the list is set to point at the new struct. This procedure is repeated each time a new structure is created with the result that a linked list of structures is created. The pointer start always points to the first struct in the list and the prt component of this struct points to the next and so on. You should be able to see how to write a program that examines or prints the data in each of the structures. For example:

```
thisptr=start;  
while (1==1)  
{  
    printf("%d",thisptr-> data);  
    thisptr=thisptr->prt;  
}
```

This first sets thisptr to the start of the list, prints the data in the first element and then gets the pointer to the next struct in the list and so on. How does the program know it has reached the end of the list? At the moment it just keeps going into the deep and uncharted regions of your machine's memory! To stop it we have to mark the end of the list using a null pointer. Usually a pointer value of 0 is special in that it never occurs in a pointer pointing at a valid area of memory. You can use 0 to initialise a pointer so that you know it isn't pointing at anything real. So all we have to do is set the last pointer in the list to 0 and then test for it That is:

```
thisptr=start;  
while (thisptr!=0)  
{  
    printf("%d",thisptr->data);  
    thisptr=thisptr-> prt;  
}
```

To be completely correct you should TYPE cast 0 to be a pointer to the struct in question. That is:

```
while (thisptr!=(struct list*)0)
```

By generally mucking about with pointers stored in the list you can rearrange it, access it, sort it, delete items and do anything you want to. Notice that the structures in the list can be as complicated as you like and, subject to there being enough memory, you can create as many structures as you like.

You can use the same sort of technique to create even more complicated list structures. For example you can introduce another pointer into each structure and a pointer to the end of the list so that you can work your way along it in the other direction - a doubly linked list. You can create stacks, queues, dequeues, trees and so on. The rest of the story is a matter of either inventing these data structures for yourself or looking them up in a suitable book.

Structures and C++:

The reason why structures are even more important for today's budding C programmer is that they turn into classes in C++. A class is a structure where you can define components that are

functions. In this case the same distinction between a data TYPE and an example of the TYPE, i.e. a variable, is maintained only now the instances of the class include functions as well as data. The same qualified naming system applies to the class and the use of pointers and the -> operator. As this is the basis of C++'s object-oriented features it is important to understand.

Header Files:

The final mystery of C that needs to be discussed is the header file. This started off as a simple idea, a convenience to make programming easier. If you have a standard set of instructions that you want to insert in a lot of programs that you are writing then you can do it using the #include statement.

The # symbol at the start indicates that this isn't a C statement but one for the C pre-processor which looks at the text file before the compiler gets it. The #include tells the pre-processor to read in a text file and treat it as if it was part of the program's text. For example:

```
#include "copy.txt"
```

could be used to include a copyright notice stored in the file copy.txt. However the most common use of the #include is to define constants and macros. The C pre-processor is almost a language in its own right. For example, if you define the identifier NULL as:

```
#define NULL 0
```

then whenever you use NULL in your program the pre-processor substitutes 0. In most cases you want these definitions to be included in all your programs and so the obvious thing to do is to create a separate file that you can #include.

This idea of using standard include files has spiralled out of all proportions. Now such include files are called header files and they are distinguished by ending in the extension .h. A header file is generally used to define all of the functions, variables and constants contained in any function library that you might want to use. The header file stdio.h should be used if you want to use the two standard I/O functions printf and scanf. The standard libraries have been covered in a previous section.

This sort of use of header files is simple enough but over time more and more standard elements of the C environment have been moved into header files. The result is that header files become increasingly mysterious to the beginner. Perhaps they reach their ultimate in complexity as part of the Windows development environment. So many constants and macros are defined in the Windows header files that they amount to hundreds of lines! As another example of how you could use a header file consider the complex structure defined earlier. At the moment it looks messy to declare a new complex variable as:

```
struct comp a,b;
```

If you want to make the complex TYPE look like other data types all you need is a single #define

```
#define COMPLEX struct comp
```

After this you can write:

```
COMPLEX a,b;
```

and the pre-processor will automatically replace COMPLEX by struct comp for you when you compile the program. Put this #define and any others needed to make the complex number type work and you have the makings of a complex.h header file of your very own.