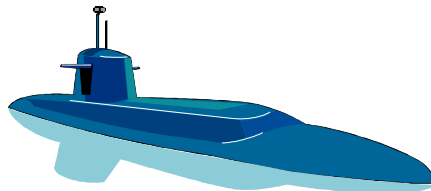


## Structures in C

- **Concepts**
- **Creating a structure template**
- **Using the template to create an instance**
- **Initialising an instance**
- **Accessing an instance's members**
- **Passing instances to functions**
- **Linked lists**



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 1

---

## Structures in C

This chapters investigates structures (records) in C.

## Concepts

- **A structure is a collection of one or more variables grouped together under a single name for convenient handling**
- **The variables in a structure are called *members* and may have any type, including arrays or other structures**
- **The steps are:**
  - **set-up a template (blueprint) to tell the compiler how to build the structure**
  - **Use the template to create as many instances of the structure as desired**
  - **Access the members of an instance as desired**

© Cheltenham Computer Training 1994/1997    sales@ccttrain.demon.co.uk

Slide No. 2

---

## Concepts

Thus far we have examined arrays. The fundamental property of the array is that all of the elements are exactly the same type. Sometimes this is not what is desired. We would like to group things of potentially different types together in a tidy “lump” for convenience.

Whereas the parts of an array are called “elements” the parts of a structure are called “members”.

Just as it is possible to have arrays of any type, so it is possible to have any type within a structure (except void). It is possible to place arrays inside structures, structures inside structures and possible to create arrays of structures.

The first step is to set up a blueprint to tell the compiler how to make the kinds of structures we want. For instance, if you wanted to build a car, you’d need a detailed drawing first. Just because you possess the drawing does not mean you have a car. It would be necessary to take the drawing to a factory and get them to make one. The factory wouldn’t just stop at one, it could make two, three or even three hundred. Each car would be a single individual instance, with its own doors, wheels, mirrors etc.

## Setting up the Template

- Structure templates are created by using the **struct** keyword

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

```
struct Book
{
    char title[80];
    char author[80];
    float price;
    char isbn[20];
};
```

```
struct Library_member
{
    char name[80];
    char address[200];
    long member_number;
    float fines[10];
    struct Date dob;
    struct Date enrolled;
};
```

```
struct Library_book
{
    struct Book b;
    struct Date due;
    struct Library_member *who;
};
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 3

## Setting up the Template

The four examples above show how the template (or blueprint) is specified to the compiler. The keyword **struct** is followed by a name (called a *tag*). The tag helps us to tell the compiler which of the templates we're interested in. Just because we have a structure template does not mean we have any structures. No stack, data segment or heap memory is allocated when we create a structure template. Just because we have a blueprint telling us that a book has a title, author, ISBN number and price does not mean we have a book.

### Structures vs. Arrays

The Date structure, consisting as it does of three integers offers advantages over an array of three integers. With an array the elements would be numbered 0, 1 and 2. This would give no clue as to which one was the day, which the month and which the year. Using a structure gives these members names so there can be no confusion.

The Book structure not only contains members of different types (**char** and **float**) it also contains three arrays.

The Library\_member structure contains two Date structures, a date of birth as well as a date of enrolment within the library.

Finally the Library\_book structure contains a Book structure, a Date structure and a pointer to a Library\_member structure.

## Creating Instances

- Having created the template, an instance (or instances) of the structure may be declared

```
struct Date
{
    int day;
    int month;
    int year;
} today, tomorrow;

struct Date next_monday;

struct Date next_week[7];
```

instances must be declared before the ';' ...

... or "struct Date" has to be repeated

an array of 7 date instances

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 4

## Creating Instances

### Instance?

The template gives the compiler all the information it needs on how to build an instance or instances. An "instance" is defined in the dictionary as "an example, or illustration of". Going back to the car example, the blueprint enables us to make many cars. Each car is different and distinct. If one car is painted blue, it doesn't mean *all* cars are painted blue. Each car is an "instance". Each instance is separate from every other instance and separate from the template. There is only ever one template (unless you want to start building slightly different kinds of car).

Above, the Date template is used to create two date instances, "today" and "tomorrow". Any variable names placed after the closing brace and before the terminating semicolon are structures of the specified type.

After the semicolon of the structure template, "**struct Date**" needs to be repeated.

With the array "next\_week", each element of the array is an individual Date structure. Each element has its own distinct day, month and year members. For instance, the day member of the first (i.e. zeroth) date would be accessed with:

**next\_week[0].day**

the month of the fourth date would be accessed with:

**next\_week[3].month**

and the year of the last date with:

**next\_week[6].year**

## Initialising Instances

- Structure instances may be initialised using braces (as with arrays)

```
int  primes[7] = { 1, 2, 3, 5, 7, 11, 13 };

struct Date   bug_day = { 1, 1, 2000 };

struct Book   k_and_r = {
    "The C Programming Language 2nd edition",
    "Brian W. Kernighan and Dennis M. Ritchie",
    31.95,
    "0-13-110362-8"
};
```

```
struct Book
{
    char    title[80];
    char    author[80];
    float   price;
    char    isbn[20];
};
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 5

## Initializing Instances

In the last chapter we saw how braces were used in the initialization of arrays, as in the “primes” example above. The seven slots in the array are filled with the corresponding value from the braces.

A similar syntax is used in the initialization of structures. With the initialization of “bug\_day” above, the first value 1 is assigned into bug\_day’s first member, “day”. The second value “1” is assigned into bug\_day’s second member, “month”. The 2000 is assigned into bug\_day’s third member “year”. It is just as though we had written:

```
struct Date bug_day;
```

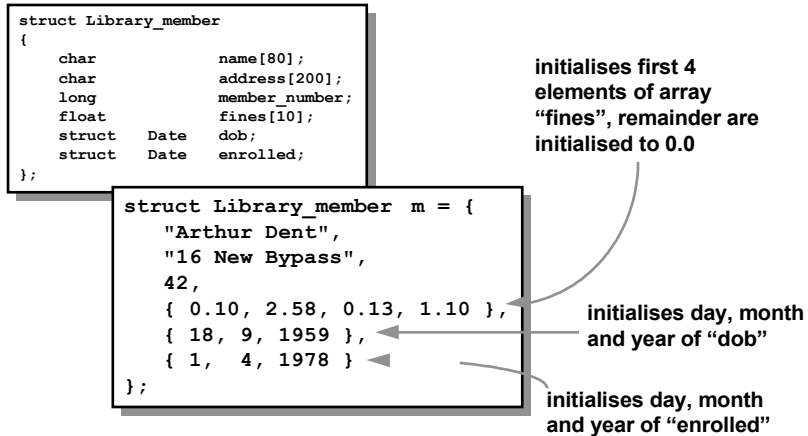
```
bug_day.day = 1;
bug_day.month = 1;
bug_day.year = 2000;
```

With the initialization of “k\_and\_r” the first string is assigned to the member “title”, the second string assigned to the member “author” etc. It is as though we had written:

```
struct Book k_and_r;
```

```
strcpy(k_and_r.title, "The C Programming Language 2nd
edition");
strcpy(k_and_r.author, "Brian W. Kernighan and Dennis
M. Ritchie");
k_and_r.price = 31.95;
strcpy(k_and_r.isbn, "0-13-110362-8");
```

## Structures Within Structures



© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 6

## Structures Within Structures

We have already seen that it is possible to declare structures within structures, here is an example of how to initialize them. To initialize a structure or an array braces are used. To initialize an array within a structure two sets of braces must be used. To initialize a structure within a structure, again, two sets of braces must be used.

It is as though we had written:

```

struct Library_member m;

strcpy(m.name, "Arthur Dent");
strcpy(m.address, "16 New Bypass");
m.member_number = 42;
m.fines[0] = 0.10; m.fines[1] = 2.58; m.fines[2] = 0.13; m.fines[3] = 1.10;
m.fines[4] = 0.00; m.fines[5] = 0.00; m.fines[6] = 0.00; m.fines[7] = 0.00;
m.fines[8] = 0.00; m.fines[9] = 0.00;
m.dob.day = 18; m.dob.month = 9; m.dob.year = 1959;
m.enrolled.day = 1; m.enrolled.month = 4; m.enrolled.year = 1978;
  
```

**Reminder -** Although a small point, notice the date initialization:

**Avoid  
Leading  
Zeros**

```
{ 18, 9, 1959 }
```

above. It is important to resist the temptation to write:

```
{ 18, 09, 1959 }
```

since the leading zero introduces an octal number and "9" is not a valid octal digit.

## Accessing Members

- Members are accessed using the instance name, “.” and the member name

```
struct Library_member
{
    char        name[80];
    char        address[200];
    long        member_number;
    float       fines[10];
    struct Date dob;
    struct Date enrolled;
}

struct Library_member m;

printf("name = %s\n", m.name);
printf("membership number = %li\n", m.member_number);
printf("fines: ");
for(i = 0; i < 10 && m.fines[i] > 0.0; i++)
    printf("£%.2f ", m.fines[i]);
printf("\njoined %i/%i/%i\n", m.enrolled.day,
        m.enrolled.month, m.enrolled.year);
```

© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 7

## Accessing Members

Members of structures are accessed using C's “.” operator. The syntax is:

**structure\_variable.member\_name**

### Accessing Members Which are Arrays

If the member being accessed happens to be an array (as is the case with “fines”), square brackets must be used to access the elements (just as they would with any other array):

**m.fines[0]**

would access the first (i.e. zeroth) element of the array.

### Accessing Members Which are Structures

When a structure is nested inside a structure, two dots must be used as in

**m.enrolled.month**

which literally says “the member of ‘m’ called ‘enrolled’, which has a member called ‘month’”. If “month” were a structure, a third dot would be needed to access one of its members and so on.

## Unusual Properties

- Structures have some very “un-C-like” properties, certainly when considering how arrays are handled

	<u>Arrays</u>	<u>Structures</u>
Name is	pointer to zeroth element	the structure itself
Passed to functions by	pointer	value or pointer
Returned from functions	no way	by value or pointer
May be assigned with “=”	no way	yes

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 8

## Unusual Properties

### Common Features Between Arrays and Structures

Structures and arrays have features in common. Both cause the compiler to group variables together. In the case of arrays, the variables are elements and have the same type. In the case of structures the variables are members and may have differing type.

### Differences Between Arrays and Structures

Despite this, the compiler does not treat arrays and structures in the same way. As seen in the last chapter, in C the name of an array yields the address of the zeroth element of the array. With structures, the name of a structure instance is *just* the name of the structure instance, NOT a pointer to one of the members.

When an array is passed to a function you have no choice as to how the array is passed. As the name of an array is “automatically” a pointer to the start, arrays are passed by pointer. There is no mechanism to request an array to be passed by value. Structures, on the other hand may be passed either by value or by pointer.

An array cannot be returned from a function. The nature of arrays makes it possible to return a pointer to a particular element, however this is not the same as returning the whole array. It could be argued that by returning a pointer to the first element, the whole array is returned, however this is a somewhat weak argument. With structures the programmer may choose to return a structure or a pointer to the structure.

Finally, arrays cannot be assigned with C’s assignment operator. Since the name of an array is a constant pointer to the first element, it may not appear on the left hand side of an assignment (since no constant may be assigned to). Two structures may be assigned to one another. The values stored in the members of the right hand structure are copied over the members of the left hand structure, even if these members are arrays or other structures.



## Instances may be Assigned

- Two structure instances may be assigned to one another via “=”
- All the members of the instance are copied (including arrays or other structures)

```
struct Library_member m = {
    "Arthur Dent",
    . . . . .
};
struct Library_member tmp;
tmp = m;
```

copies array “name”, array “address”, long integer “member\_number”, array “fines”, Date structure “dob” and Date structure “enrolled”

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 9

## Instances May be Assigned

### Cannot Assign Arrays

It is not possible to assign arrays in C, consider:

```
int a[10];
int b[10];

a = b;
```

The name of the array “a” is a constant pointer to the zeroth element of “a”. A constant may not be assigned to, thus the compiler will throw out the assignment “a = b”.

### Can Assign Structures Containing Arrays

Consider:

```
struct A {
    int array[10];
};
struct A a, b;

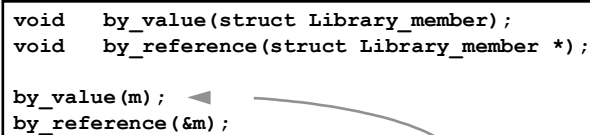
a = b;
```

Now both instances “a” and “b” contain an array of 10 integers. The ten elements contained in “b.array” are copied over the ten elements in “a.array”. Not only does this statement compile, it also works! All the members of a structure are copied, no matter how complicated they are. Members which are arrays are copied, members which are nested structures are also copied.

## Passing Instances to Functions

- An instance of a structure may be passed to a function by value or by pointer
- Pass by value becomes less and less efficient as the structure size increases
- Pass by pointer remains efficient regardless of the structure size

```
void  by_value(struct Library_member);  
void  by_reference(struct Library_member *);  
  
by_value(m);  
by_reference(&m);
```



compiler writes a pointer  
(4 bytes?) onto the stack

compiler writes 300+  
bytes onto the stack

© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 10

## Passing Instances to Functions

### Pass by Value or Pass by Reference?

As a programmer you have a choice of passing a structure instance either by value or by pointer. It is important to consider which of these is better. When passing an array to a function there is no choice. There isn't a choice for one important reason, it is invariably *less* efficient to pass an array by value than it is by pointer. Consider an array of 100 **long int**. Since a **long int** is 4 bytes in size, and C guarantees to allocate an array in contiguous storage, the array would be a total of 400 bytes.

If the compiler used pass by value, it would need to copy 400 bytes onto the stack. This would be time consuming and we may, on a small machine, run out of stack space (remember we would need to maintain two copies - the original and the parameter). Here we are considering a "small" array. Arrays can very quickly become larger and occupy even more storage.

When the compiler uses pass by reference it copies a pointer onto the stack. This pointer may be 2 or 4 bytes, perhaps larger, but there is no way its size will compare unfavorably with 400 bytes.

The same arguments apply to structures. The `Library_member` structure is over 300 bytes in size. The choice between copying over 300 bytes vs. copying around 4 bytes is an easy one to make.

## Pointers to Structures

- Passing pointers to structure instances is more efficient
- Dealing with an instance at the end of a pointer is not so straightforward!

```
void    member_display(struct Library_member *p)
{
    printf("name = %s\n", (*p).name);
    printf("membership number = %li\n", (*p).member_number);

    printf("fines: ");
    for(i = 0; i < 10 && (*p).fines[i] > 0.0; i++)
        printf("£%.2f ", (*p).fines[i]);

    printf("\njoined %i/%i/%i\n", (*p).enrolled.day,
        (*p).enrolled.month, (*p).enrolled.year);
}
```

© Cheltenham Computer Training 1994/1997    sales@ccctrain.demon.co.uk

Slide No. 11

## Pointers to Structures

Passing a pointer to a structure in preference to passing the structure by value will almost invariably be more efficient. Unfortunately when a pointer to a structure is passed, coding the function becomes tricky. The rather messy construct:

**(\*p).name**

is necessary to access the member called "name" (an array of characters) of the structure at the end of the pointer.

## Why (\*p) . name ?

- The messy syntax is needed because “.” has higher precedence than “\*”, thus:

`*p.name`

means “what p.name points to” (a problem because there is no structure instance “p”)

- As Kernighan and Ritchie foresaw pointers and structures being used frequently they invented a new operator

`p->name`      =      `(*p).name`

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 12

## Why (\*p) . name?

The question occurs as to why: `(*p) . name`

is necessary as opposed to: `*p . name`

The two operators “\*” and “.” live at different levels in the precedence table. In fact “.”, the structure member operator, is one of the highest precedence operators there is. The “pointer to” operator, “\*”, although being a high precedence operator is not quite as high up the table.

Thus: `*p . name`

would implicitly mean: `*(p . name)`

For this to compile there would need to be a structure called “p”. However “p” does not have type “structure”, but “pointer to structure”. Things get worse. If “p” were a structure after all, the name member would be accessed. The “\*” operator would find where “p.name” pointed. Far from accessing what we thought (a pointer to the zeroth element of the array) we would access the first character of the name. With `printf`’s fundamental inability to tell when we’ve got things right or wrong, printing the first character with the “%s” format specifier would be a fundamental error (`printf` would take the ASCII value of the character, go to that location in memory and print out all the bytes it found there up until the next byte containing zero).

## A New Operator

Since Kernighan and Ritchie foresaw themselves using pointers to structures frequently, they invented an operator that would be easier to use. This new operator consists of two separate characters “.” and “>” combined together into “->”. This is similar to the combination of divide, “/”, and multiply, “\*”, which gives the open comment sequence.

The messy `(*p) . name` now becomes `p->name` which is both easier to write and easier to read.

## Using p->name

- Now dealing with the instance at the end of the pointer is more straightforward

```
void member_display(struct Library_member *p)
{
    printf("name = %s\n", p->name);
    printf("address = %s\n", p->address);
    printf("membership number = %li\n", p->member_number);

    printf("fines: ");
    for(i = 0; i < 10 && p->fines[i] > 0.0; i++)
        printf("£%.2f ", p->fines[i]);

    printf("\njoined %i/%i/%i\n", p->enrolled.day,
        p->enrolled.month, p->enrolled.year);
}
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 13

## Using p->name

As can be seen from the code above, the notation:

**p->name**

although exactly equivalent to:

**(\*p).name**

is easier to read, easier to write and easier to understand. All that is happening is that the member “name” of the structure at the end of the pointer “p” is being accessed.

Note:

**p->enrolled.day**

and NOT:

**p->enrolled->day**

since “enrolled” is a structure and not a pointer to a structure.

## Pass by Reference - Warning

- Although pass by reference is more efficient, the function can alter the structure (perhaps inadvertently)
- Use a pointer to a constant structure instead

```
void member_display(struct Library_member *p)
{
    printf("fines: ");
    for(i = 0; i < 10 && p->fines[i] = 0.0; i++)
        printf("%f ", p->fines[i]);
}
```

function alters  
the library  
member instance

```
void member_display(const struct Library_member *p)
{
    ....
}
```



© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 14

## Pass by Reference - Warning

We have already seen how passing structure instances by reference is more efficient than pass by value. However, never forget that when a pointer is passed we have the ability to alter the thing at the end of the pointer. This is certainly true with arrays where any element of the array may be altered by a function passed a pointer to the start.

Although we may not intend to alter the structure, we may do so accidentally. Above is one of the most popular mistakes in C, confusing “=” with “==”. The upshot is that instead of testing against 0.0, we assign 0.0 into the zeroth element of the “fines” array. Thus the array, and hence the structure are changed.

### const to the Rescue!

The solution to this problem which lies with the **const** keyword (discussed in the first chapter). In C it is possible to declare a pointer to a constant. So:

```
int *p;
```

declares “p” to be a pointer to an integer, whereas:

```
const int *p;
```

declares “p” to be a pointer to a constant integer. The pointer “p” may change, so

```
p++;
```

would be allowed. However the value at the end of the pointer could not be changed, thus

```
*p = 17;
```

would NOT compile. The parameter “p” to the function **member\_display** has type “pointer to constant structure Library\_member” meaning the structure Library member on the end of the pointer cannot be changed.

## Returning Structure Instances

- Structure instances may be returned by value from functions
- This can be as inefficient as with pass by value
- Sometimes it is convenient!

```
struct Complex add(struct Complex a, struct Complex b)
{
    struct Complex result = a;
    result.real_part += b.real_part;
    result.imag_part += b.imag_part;
    return result;
}

struct Complex c1 = { 1.0, 1.1 };
struct Complex c2 = { 2.0, 2.1 };
struct Complex c3;

c3 = add(c1, c2);    /* c3 = c1 + c2 */
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 15

## Returning Structure Instances

As well as pass by value, it is also possible to return structures by value in C. The same consideration should be given to efficiency. The larger the structure the less efficient return by value becomes as opposed to return by pointer. Sometimes the benefits of return by value outweigh the inefficiencies. Take for example the code above which manipulates complex numbers. The **add** function returns the structure “result” by value. Consider this version which attempts to use return by pointer:

```
struct Complex* add(struct Complex a, struct Complex b)
{
    struct Complex result = a;
    /* as above */
    return &result;
}
```

This function contains a fatal error! The variable “result” is stack based, thus it is allocated on entry into the function and deallocated on exit from the function. When this function returns to the calling function it hands back a pointer to a piece of storage which has been deallocated. Any attempt to use that storage would be very unwise indeed. Here is a working version which attempts to be as efficient as possible:

```
void add(struct Complex *a, struct Complex *b, struct Complex *result)
{
    result->real_part = a->real_part + b->real_part;
    result->imag_part = a->imag_part + b->imag_part;
}
```

Pass by pointer is used for all parameters. There is no inefficient return by value, however consider how this function must be called and whether the resulting code is as obvious as the code above:

```
struct Complex c1 = { 1.0, 1.1 }, c2 = { 2.0, 2.1 }, c3;

add(&c1, &c2, &c3);
```