
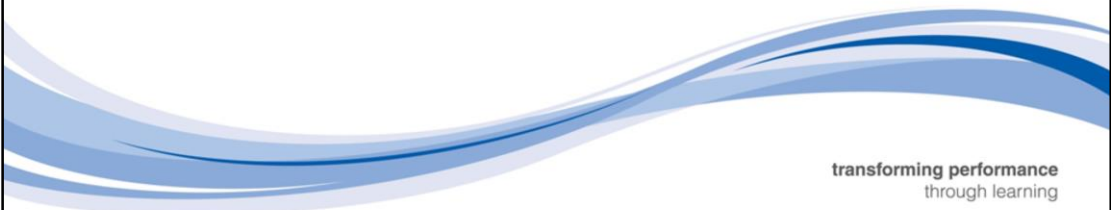


QACADV_v1.0



Advanced C

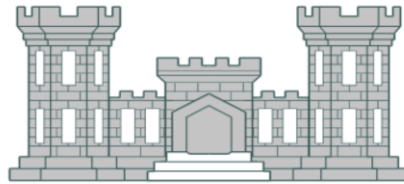
08 Structures



transforming performance
through learning

Structures

- **Objectives**
 - How to create composite data types
- **Contents**
 - Tagged and untagged structures
 - Nested structures
 - Using *typedef*
 - Initialising structures
 - Structure packing and padding
 - Bitfields
 - Structures and files
- **Practical**
- **Summary**



Structures in C are user-defined data types that allow items of related data to be grouped together into a single cohesive unit.

In this chapter, we shall examine a number of advanced structure-handling techniques and issues, starting with the use of untagged structures and nested structures, then moving on to look at the use of `typedefs` for simplifying structure definitions.

Many compilers insert extra "padding bytes" between members in a structure to achieve alignment of members at even addresses, for example. We shall investigate this technique and discuss its advantages and implications. A related topic is the use of bitfields, that is structure members that occupy a specified number of bits. The use of bitfields involves various trade-offs which we shall discuss in due course.

The chapter closes with a look at file handling techniques for structures, including a discussion of indexed file principles.

Tagging structures

- Structure templates can be given names (tags) which must conform to the rules for declaring variables
- Tags live in a different namespace to variables, the "struct" keyword accesses the alternate namespace

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

"i" lives in the variable namespace

"date" lives in the struct/union namespace

*"date" now lives in both namespaces
NOT AN ERROR*

use of "date" in struct/union namespace to declare variable "d"

Structure templates are given names so that if many are declared it is possible to distinguish between them. These names, or *tags*, must conform to the rules for declaring C variables, i.e. begin with a letter or underscore, followed by letters, digits or underscores.

Structure tags (and union tags too) are held in a special *namespace*. A namespace is just a space in which names live.

A variable can be declared with the same name as a structure tag. The example shown above illustrates this with `date`, which is both a structure tag and a variable name. The compiler cannot become confused. The `struct` keyword informs the compiler to switch into the struct/union namespace and think "tag". Its absence means the compiler thinks "variable".

Untagged structures

- It is possible to declare an "unnamed" structure

```
struct
{
    int    a;
    float  b;
} un1, un2, un3;

un1.a = 10;
un2.b = 22.4F;
un1.b = un2.b;
```

*all variables must be
declared by this point*

- No other variables of this type can be declared beyond the terminating ";"
- Variables may not be passed into functions (no way to declare parameter)
- Useful only for variables of a "one off" structure type

Structures may be declared which are not tagged. Not having a tag makes it rather difficult to refer to the structure. In fact, if variables of the structure type are not declared before the terminating semi-colon, they cannot be declared at all.

Consider the following code:

```
struct
{
    int    a;
} s1;
```

```
struct
{
    int    a;
} s2;
```

You may consider that s1 and s2 have the same type. However, the compiler thinks differently; the two structure variables s1 and s2 are considered as having *different* data types, even though the internal layout of the structure variables is the same.

typedef'ing structures

- Remember that `typedef` defines a new name for an existing type, instead of defining a variable

```
struct date_tag
{
    int day;
    int month;
    int year;
};
typedef struct date_tag Date_t;
```



- This may be done in one step as follows:

```
typedef struct date_tag ← optional
{
    int date;
    int month;
    int year;
} Date_t;
```

When `typedef` is used, instead of creating a *variable* `Date_t`, we create a new name for a type. A surprising number of people cannot get this right. For instance, consider the following mess taken from *Fundamentals of Data Structures in C*:

```
typedef struct twoThree * twoThreePtr;
struct twoThree
{
    int data;
    twoThreePtr leftChild, middleChild, rightChild;
};
```

Believe it or not, this actually compiles! Much easier is the following:

```
struct twoThree
{
    int data;
    struct twoThree * leftChild;
    struct twoThree * middleChild;
    struct twoThree * rightChild;
};
typedef struct twoThree * twoThreePtr;
```

Another (although decidedly more fatal) mistake is:

```
typedef struct date_tag
{
    int day;
    int month;
    int year;
};
```

where nothing is `typedef'd`, because covering up the `typedef` keyword, no variable would have been declared.

Initialising structures

- The same rules that apply to initialising arrays also apply to structures
- For arrays, if there are fewer initialisers than elements, the remaining are initialised to zero

```
int a[100] = { 0 };
```

- If there are fewer initialisers than members, the remaining are initialised to zero

```
struct time
{
    int hours;
    int minutes;
    int seconds;
};
struct time midnight = { 0 };
```

As previously seen in the *Arrays* chapter, when fewer initialisers are provided than elements in an array, the compiler initialises the remaining elements to zero.

This is true also of structures. When fewer initialisers are provided than members in a structure, the compiler initialises the remaining members to zero.

The initialisation of "d" above initialises the day, month and year members to zero. This is an alternative to using `memset` as in:

```
memset(&midnight, '\0', sizeof(struct time));
```

Initialising structures - C99

- C99 allows structures to be initialised by member name

```
struct person
{
    int age;
    int pno;
    _Bool Female;
};
```

Note the '.'

Member name

Subsequent members
by position or name

```
struct person fred = {.pno = ++iMaxPno, 0};
```

- Can even be passed to a function

```
void pay_em (const struct person *p);
...
pay_em (&(struct person){.pno = iMaxPno});
```

C99 has made the initialisation of individual members of a struct much easier. Structure members can be initialised by name, as shown, but they must be preceded by a 'dot'. Subsequent members can be initialised, by name in any order, or the next member if no name is given. For example:

```
struct person fred = {.age = 21, 1234};
```

would initialise the member age to 21, and pno (the next member in the struct) to 1234. Any other members would be set to zero.

Structs can be generated and initialised when passed to functions, provided they are defined as const (although the compiler does not make that check).

Alignment of members

- The compiler is not allowed to reorder the members of a structure
- To improve efficiency, members may be aligned on word boundaries
- The compiler introduces *padding*, the net effect being that structures may be larger than expected

```
struct flabby
{
    char a;
    int  b;
    char c;
    int  d;
};
```



The compiler is not allowed to reorder the members of a structure, even if reordering would result in smaller variables.

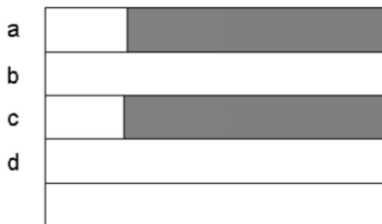
Hardware likes things to be on word boundaries. That way the "things" may be accessed more easily (and more efficiently and therefore more *quickly*). The compiler is allowed to introduce *padding* into structures in order to push members sideways.

In the `flabby` structure above, if the member `b` is to start on a word boundary, three bytes of padding must be added after the member `a` (with a 32 bit word-addressed machine). There is no problem with `c` starting on a word boundary, but a similar byte of padding must be introduced between `c` and `d`.

Although it might look from the structure definition that any variables of type `struct flabby` would be `sizeof(char) + sizeof(int) + sizeof(char) + sizeof(int)` bytes in size, they would in fact be much larger, 16 bytes on a typical 32-bit machine. The `sizeof` operator always returns the true size of a structure (not the size it might have been).

offsetof macro

- Defined in `<stddef.h>`, returns the offset (in bytes) of a member in a structure as a `size_t`



```
struct weird
{
    char    a;
    int     b;
    char    c;
    double  d;
};
```

```
#include <stddef.h>

void eg(const char * name, size_t val)
{
    printf("Offset of %s is %lu bytes\n",
        name, (unsigned long)val);
}

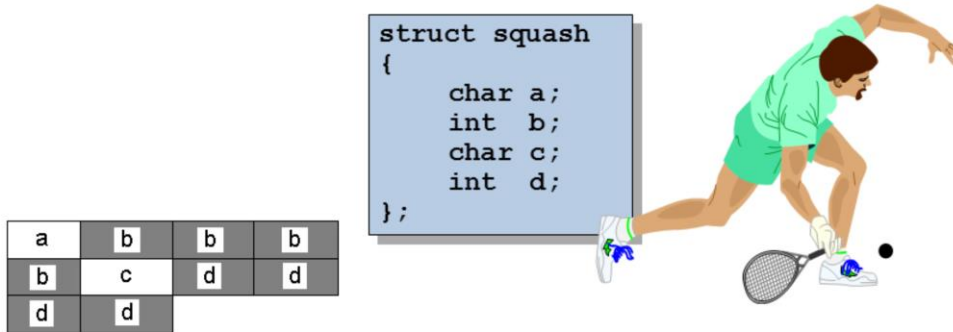
eg("a", offsetof(struct weird, a));
eg("b", offsetof(struct weird, b));
eg("c", offsetof(struct weird, c));
eg("d", offsetof(struct weird, d));
```

As well as the `sizeof` operator giving some insight into how the compiler does things, the `offsetof` macro is available. It "returns", in bytes, the offset of a member from the start of a structure.

In the example shown above, the compiler has inserted padding to achieve word-alignment of members. On a typical 32-bit machine, the member `a` would be at offset 0 bytes, `b` at offset 4 bytes, `c` at offset 8 bytes and `d` at offset 12 bytes. The underlying type of `size_t` is either `unsigned int` or `unsigned long`, so for portability the example prints these values as unsigned longs (`%lu`).

Structure packing

- **Most compilers have some form of structure packing option**
 - Structures will be smaller
 - Accessing some/all members could be slower
- **Not a replacement for considered structure member layout**



Most compilers have a compile-time option for compressing (packing) structures into the smallest space possible, and many support `#pragma pack(1)`. There is no standard command-line flag for achieving this. You should be warned that some compilers have structure packing permanently enabled, which can be a problem (as will be seen below).

Shown above is our flabby structure renamed, which assumes a 32-bit word. The layout of the members in memory is different from before. There are no longer redundant bytes of padding between the members `a` and `b`. However, the members `b` and `d` have been sliced.

Assuming 32-bit addressing, to access the member `b` the compiler must:

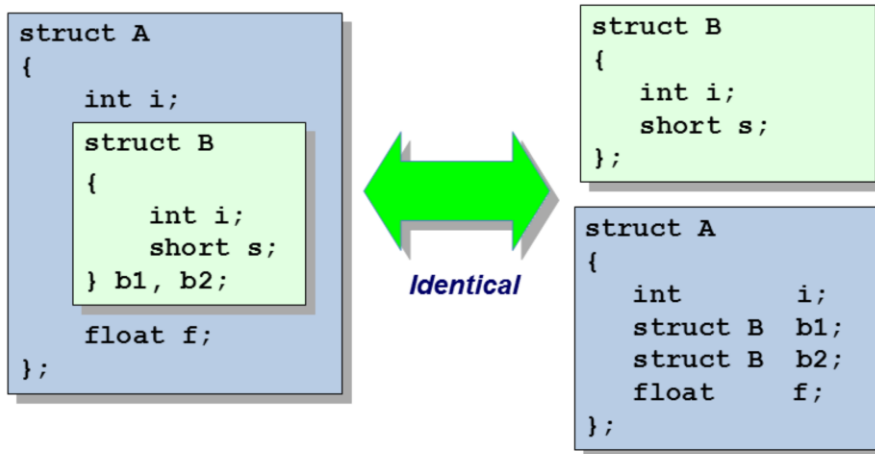
1. Load the word containing `a` and bytes 1-3 of `b` into a register.
2. Shift the register contents left by 8 bits (the value of `a` will be lost).
3. Load the word containing byte 3 of `b` (including `c` and half of `d`) into another register.
4. Right shift by 24 bits (the values of members `c` and `d/2` will be lost).
5. Bit-wise OR the two registers together.

Although the saving in space is advantageous, the increased amount of code necessary on each access to the member `b` is not.

The general rule for the rearranging structures is to place the data into the following order: doubles, floats, longs, ints, shorts, chars. This guarantees that they are as tightly packed as possible independently of the size of an int.

Nesting structures

- Structures can be *defined* within one another
- However, C has extremely poor scoping rules here



The C compiler allows one structure to be *defined* within another one. This is an extension of the simple nesting of structures. Here, the nested structure is being defined "on the fly".

Although you might expect the structure B in the example on the left to be "hidden" somehow by the compiler, this is not the case. The two code fragments shown are identical. This is really rather poor.

Note: C++ compilers support true nesting of structure definitions, so that a structure defined inside another structure is indeed hidden from the "outside world".

Forward declarations

- Nested structures may be "forward declared" as follows:

```
struct inner;

struct outer
{
    int        i;
    float      f;
    struct inner * ps;
    struct inner s;
};

struct outer o;
```



- The nested structure must be fully defined before variables may be declared

A forward declaration is a one-line statement to the compiler introducing the *name* of a new structure type, without giving any more information. This is often referred to as an *incomplete type*, in the sense that the compiler is aware of the type name (such as `struct inner`) without having full details about the structure layout. This allows subsequent structures to contain *pointers* to this structure type, as in the case of `ps` in the `outer` structure above. The compiler is happy with this, because all data pointers in a program will have the same size. It is irrelevant that the compiler doesn't yet know the size of an actual `struct inner`. However, any attempt to include a full `struct inner` member in the `outer` structure is not allowed, since the compiler needs to know the exact size and layout of `struct inner` before this is possible. The ability to "forward declare" structures is useful for mutually recursive structures as follows:

```
struct MyComplexData;
struct LinkedList
{
    struct LinkedList *next;
    struct MyComplexData *data;
};
struct MyComplexData
{
    struct LinkedList *surprise;
};
```

The type of `struct MyComplexData` may be declared later (even *differently* for different data storage requirements) in different applications.

Bitfields

- Members may be defined as bitfields

```
struct bitbag
{
    unsigned int first : 1;
    unsigned int second : 2;
    unsigned int third : 1;
    unsigned int pad : 4;
    unsigned int : 4;
    unsigned int extra : 3;
};
```

Alternatives

- The member "first" is 1 bit wide, "second" is 2 bits wide, "pad" is 4 bits wide, and so on

Declaring bitfields as members of structures has been a feature of C for some time. Since it was "unofficial" (and therefore not generally known about) before the advent of ISO C, it was not used widely.

The member `first` in the structure `bitbag` shown above really will be only 1 bit wide. The type `unsigned` is usually used for bitfields. This avoids the compiler trying to use any of the available bits (i.e. all one of them) as *sign* bits. Don't try using a floating type (`float`, `double` or `long double`), the compiler just won't swallow it.

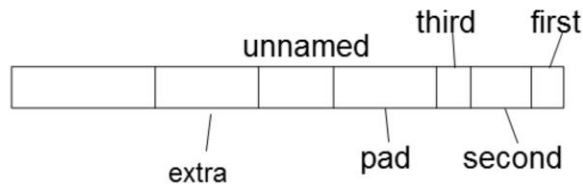
The member `second` is 2 bits wide, and so on. The member `pad` is 4 bits wide. If "padding" is required, however, the alternative syntax can be used to declare the unnamed member following `pad`.

Bitfields implementation

- "Almost everything about bitfields is implementation dependent"

K&R p.150

- The machine allocates a number of words to contain the members



- Members may fill up from either the right or left, depending on compiler

Machines today may have 16, 32, 64 or even 128-bit registers. But if a register is the "least common denominator" of a machine, how can a one or two bit value in a bitfield be loaded into a register? The comment "Almost everything about bitfields..." is almost a cop-out, *everything* is left up to the compiler writer. The ISO standard is not much clearer:

"An implementation may allocate any addressable storage unit large enough to hold a bitfield. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified."

Once a word has been allocated, the members are shoved in somehow. Accessing the member `third` will involve the compiler generating code to perform the following operations:

1. load the (whole) word containing `third` into a register.
2. generate a mask containing zeros, except for the bits corresponding to the member `third`.
3. bit-wise OR the mask and register together.
4. shift left the remaining bits until they appear in the least significant part of the word.

In a manner similar to structure packing, bitfields save storage. However, much more code is required to extract the bitfields and manipulate them.

Accessing bitfields

- Access to a bitfield is the same as for other members
- The address of a bitfield may not be taken

```
struct bitbag bits;

bits.first = 1;
bits.second = 3;

if (bits.extra == 4)
    printf("hello\n");
```

```
struct bitbag bits;

scanf("%d", &bits.extra);
```



- If "too many" bits are assigned, the compiler *should* mask out extra bits, then assign

```
bits.second = 10000;
```

only two least significant bits should be assigned

The compiler does all the work (mentioned previously) and lets the programmer access bitfield members just as though they were "ordinary" members.

One thing the compiler cannot do is take the address of a bitfield. The machine would have to be able to address individual bits for this to work. Since most machines cannot do this, the easiest solution was for C to ban the use of `&` with bitfields.

The member `second` is 2 bits wide and may hold values of 0, 1, 2 or 3. If a value larger than this is assigned to `second` the compiler *should* generate code which masks out all but the two low order bits before assigning:

bit pattern for 10000 (decimal) 10011100010000

bit pattern masked out 00

Thus 0 should be assigned to `second`. However, what happens to all those extra bits, present in 10000, not present in 0? A bad compiler will spatter these extra bits over the other bitfield members `third`, `pad`, `extra`, etc.

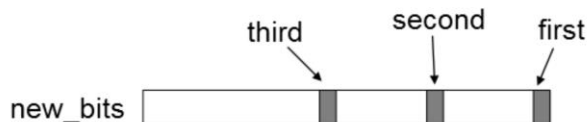
Bitfields on boundaries

- A bitfield can be placed on a word boundary (it *may* be accessed more efficiently)

```
struct new_bitbag
{
    unsigned int first : 1;
    char          : 0;
    unsigned int second : 1;
    unsigned int      : 0;
    unsigned int third  : 1;
};
```

not standard C

- "second" is pushed onto an allocation boundary, so is "third"



The declaration `unsigned:4` seen previously declares an unnamed member which is 4 bits wide.

`char:0` will push the next bitfield member onto a byte boundary.

`unsigned:0` will push the next bitfield member onto a word boundary.

Doing this *may* allow your hardware to access the "pushed" members more efficiently. The member `third` sits on a word boundary and may be accessed more efficiently than might otherwise have been the case.

NOTE: using `char` as a bit field is not ISO C standard.

The use of `char:0` to push onto a byte boundary works on Microsoft Visual C++ version 5.0, and GNU gcc (egcs-2.95.2). It does *not* work on Microsoft Visual C++ version 6.0, where it pushes onto a word boundary instead. Moral: don't rely on non-standard features!

C89 changes

- C89 (ANSI) compilers have additional struct features
- Automatic structures may be initialised
- Structures may be assigned

```
struct time lunch = { 12, 30, 0 };  
struct time liquid;  
liquid = lunch;
```

*each member of lunch is
assigned to each member of liquid*



- Structure comparison is not supported

```
if (liquid == lunch)  
    printf("Hangover!\n");
```



Standard C compilers are slightly more flexible with regard to structures than their K&R counterparts. Firstly, automatic structures (those declared within functions) may be initialised. This feature was adopted by compiler vendors before the C Standard was ratified. It is hard to remember the inconvenience that used to result from the lack of this feature.

Structure variables *of the same type* may be assigned to one another. In K&R days the following code would have been necessary:

```
memcpy(&liquid, &lunch, sizeof(struct time));
```

Each member is copied across, byte by byte regardless of its complexity.

The fact that structures can be assigned so easily leads one to the false conclusion that they may be compared for equality also. The compiler definitely does not support this.

Note there is now an inconsistency between the treatment of the two types of aggregate variable, arrays and structures. Two structures may be assigned, two arrays may not.

Comparing structures

- Comparing structures can give rise to problems. Consider the following:

```
struct weird w1 = { 'a', 26, 'x', 1.71 };
struct weird w2 = { 'a', 26, 'x', 1.71 };
if (memcmp(&w1, &w2, sizeof(w1)) == 0)
    printf("structures are the same\n");
```



- If the structure contains padding, "identical" variables may compare unequal. Better to write:

```
int weird_equal(const struct weird * p1,
               const struct weird * p2)
{
    return p1->a == p2->a
        && p1->b == p2->b
        && p1->c == p2->c
        && p1->d == p2->d;
}
```



Since the compiler will not allow the comparison of structures, we must write structure comparison code ourselves. The most straightforward way of achieving this is via the `memcmp()` function which compares regions of memory, byte by byte. The returned value works in much the same way as `strcmp()` in that zero is returned when the blocks of memory contain the same values.

As we have seen, *padding* may exist within the structure, and the padding cannot be assigned a value. If the variables `w1` and `w2` are automatic (stack based rather than heap based), the memory (i.e. stack) they have been carved out of will be random. This implies that the padding will contain random values, even though the members will have been initialised properly.

Thus, any attempt to regard the structure as a contiguous piece of memory (the `memcmp` approach) will be dangerous.

There are two solutions:

- Write a routine which compares just the members of the structure and ignore the padding
- Initialise the whole structure (including pad bytes) using `memset()`, and use `memcmp()` safely.

Further C89 changes

- Structures may be returned by value from a function

```
struct time noon(void)
{
    static const struct time midday = { 12, 0, 0 };
    return midday;
}
```

- Much less efficient than passing by address, since a structure may be huge, whereas an address will be a fixed size
- Structures may be passed by value into a function as well

```
struct person_details joe_soap;
print_person(joe_soap);
```

A structure may be passed by value into a function, just like any other variable (which isn't an array). This may be convenient for some applications, but it is not necessarily efficient. If the structure is hundreds of bytes in size, all this will need to be copied onto the stack before the function is called. Passing an address will cause only a pointer to be copied onto the stack.

Functions may also return structures, but this again involves copying many bytes of data onto the stack.

There is another inconsistency between the compiler's handling of arrays and structures. A structure may be passed to a function either by value or by address. An array may only be passed by address, although there is a special trick we will show on the next page which allows arrays to be *passed by value* as well.

Arrays and structures

- **Structures can be passed...**
 - By value (default)
 - By address (using the & operator)
- **Arrays can be passed...**
 - By address (array name decays into pointer to initial element)
 - By value (put the array inside a struct!!!)
 - C99 allows the last member of a structure to be a VLA
 - but they cannot be initialised

```
int a[42];
print_array(a);
```

Pass by "reference"

```
struct big
{
    int a[42];
};
struct big mamma;
print_struct(mamma);
```

Pass by value

C89 introduced inconsistencies between the two main aggregate types in C, arrays and structures.

When an array is passed to a function (as seen before), it is the address at which the array starts that is copied across. Structures, on the other hand, are passed by value. By using the & (address of) operator, the address of the structure may be passed to a function.

The choice that exists for structures does not exist with arrays; placing an & before the name of an array will yield the address at which it starts (only the type will change - refer back to "Pointers to Arrays" in the *Arrays of Arrays* chapter). If it is absolutely necessary to pass an array into a function by value, this may be achieved by making the array a member of a structure. Since structures are passed by value, all their members must be copied onto the stack. An array which is a member of a structure will be copied by value into a function.

C99 allows the *last* member of a struct to be an Variable Length Array (VLA). In earlier versions of C this would be emulated by a pointer. Remember that you cannot initialise a VLA, so the normal struct initialiser will not work with a VLA member. Beware also that sizeof does not include the array pointer in the size of the struct. Typically you might use this feature as follows:

```
struct person *p = malloc (sizeof(struct person) +
                           strlen(inName) + 1);
strcpy (p->name, inName);
```

Even after this assignment, sizeof the struct does not change. Trying to get the sizeof the array (sizeof(p->name)) gives a compilation error.

Writing structures to files

- Structures may be written to files via ***fwrite()*** function
 - File should be opened in "binary" mode on MS Windows

```
size_t fwrite
(const void * dest, size_t size, size_t number, FILE * fp);
```

*address of
structs*

*size of each
structure*

*number of
structs to write*

*file to write
to*

- Returns the number of whole structs written
- Writes out from an array of structures

```
enum { limit = 32 };
struct big arr[limit];
size_t nread = fwrite(&arr, sizeof(arr[0]), limit, fp);
```

The function `fwrite` can be used to write structures to files, for example :

```
#include <stdio.h>
int main(void)
{
    enum { max_items = 100 };
    size_t items_w;
    struct weird item, items[max_items];
    FILE *fp = fopen("data.rec", "wb");

    /* Put valid information into "item" and "items" */

    if (!fp)
        ...
    items_w = fwrite(&item, sizeof(item), 1, fp);
    if (items_w != 1)
        ...
    items_w = fwrite(items, sizeof(items[0]), max_items, fp);
    if (items_w != max_items)
        ...
    return 0 ;
}
```

Reading structures from files

- Structures can be read from files via the *fread()* function
 - File should be opened in "binary" mode on MS Windows

```
size_t fread
(void * dest, size_t size, size_t number, FILE * fp);
```

Diagram illustrating the parameters of the `fread` function:

- address to place structs* points to `dest`
- size of each structure* points to `size`
- number of structs to read* points to `number`
- file to read from* points to `fp`

- Returns number of whole structs read
- Reads into an array of structures

```
enum { limit = 32 };
struct big arr[limit];
size_t nread = fread(&arr, sizeof(arr[0]), limit, fp);
```

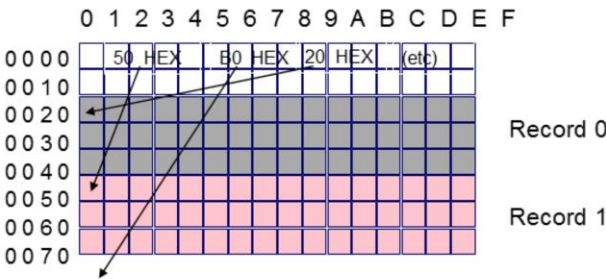
The `fread` function can be used to read structures from files, for example:

```
#include <stdio.h>
int main(void)
{
    enum { max_items = 100 };
    int          items_read;
    struct weird  item, items[max_items];
    FILE         *fp = fopen("data.rec", "rb");

    if (!fp)
        ...
    items_read = fread(&item, sizeof(item), 1, fp);
    if (fp != 1)
        ...
    items_read = fread (items, sizeof(items[0]),
                        max_items, fp);
    if (items_read != max_items)
        ...
    return 0;
}
```

Indexed files - principles

- Files can be designed to contain an "index table" at the start of the file
- The table contains a sorted list of file offsets to records in the file



- Each record may be accessed quickly via the index

This form of indexing requires that the records start at offsets which are stored in a fixed length file header before the actual data. The header would effectively be an array of `long` ints (see note below) which would be accessed sequentially to give offsets into the file. In the illustration shown above, the header is an (unrealistically small) array of 8 offsets with the 'first' record starting at offset 50 (Hex), the second at B0, the third at 20, and so on.

To access records using this file indexing approach, the following algorithm may be applied:

- While not at the end of the header
 - get an offset
 - seek to that position in the file
 - read a record
 - use it

Clearly, the secret to success is building the index.

What variable type should the offset be? The function prototype for `fseek()` in many implementations (including K&R and Microsoft) use `long` (BSD systems use `int`). On most 32-bit platforms this gives a maximum file size of 2GB. Once upon a time that was huge, now it is not - many file systems use 64-bit addressing, giving a maximum file size of 16EB (exabytes).

The Microsoft compiler offers (other than native APIs) the `lseeki64()` function, but the Unix98 standard specifies the type `off_t` (typedef'ed by platform) for `lseek()`. A new type was introduced with the ANSI C standard, called `fpos_t`. It is not used by `fseek()`, but by `fgetpos()` and `fsetpos()`, which are alternatives to `ftell()` and `fseek()`.

Indexed files - details

- Indexed files can be scanned via the `fseek()` function
 - File should be opened in "binary" mode on MS Windows
- The function repositions the current file position

```
int fseek(FILE * fp, long offset, int origin);
```

↑
file to be
'sought'

↑
number of
bytes from
origin

↑
SEEK_SET beginning of file
SEEK_CUR current position
SEEK_END end of file

- Returns non-zero on error
- There are other, platform specific, functions
 - For example: `fsetpos()`, `lseek()`, `SetFilePointer()`
 - Files might be larger than `LONG_MAX` bytes

```
/* Example of how to use fseek() to handle indexed files */
#include <stdio.h>
#include <errno.h>

int main(void)
{
    FILE          *fp;
    struct rec     buffer;
    long          offset;

    if ((fp = fopen("data.rec", "rb")) == NULL)
    {
        perror ("Cannot open data file data.rec for reading");
        return errno;
    }

    if (fread(&offset, sizeof(offset), 1, fp) != 1)
    {
        perror ("Failed to read index from data.rec");
        return errno;
    }

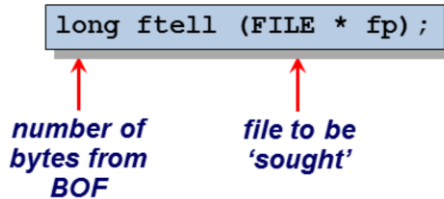
    if (fseek(fp, offset, SEEK_SET))
    {
        perror ("Failed to seek to indexed position in data.rec");
        return errno;
    }

    if (fread (&buffer,  sizeof(buffer),  1,  fp)  !=  1)
    {
        perror ("Failed to read data rec from data.rec");
        return errno;
    }

    /* Use the data rec in buffer ... */...
```


Indexed files - details

- **Offsets can be obtained via the *ftell()* function**
 - File should be opened in "binary" mode on MS Windows
- **The function returns the current file position from BOF**



- **Returns -1 on error**
- **There are other, platform specific, functions**
 - For example: *fgetpos()*, *lGetFilePointer()*
 - files might be larger than LONG_MAX bytes

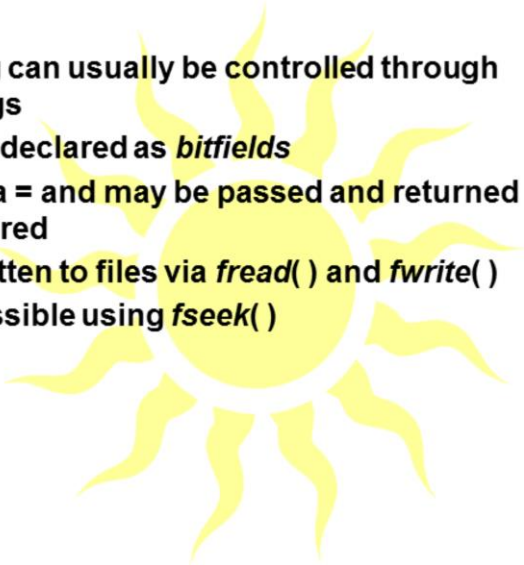
```

/* Example of how to use ftell() to handle indexed files */
#include <stdio.h>
#include <errno.h>
int main(void)
{
    FILE          *fp;
    struct rec     buffer;
    long           offset;

    if ((fp = fopen("data.rec", "rb")) == NULL)
    {
        perror ("Cannot open data file data.rec for reading");
        return errno;
    }
    /* Read some records ... */
    if (fread (&buffer,  sizeof(buffer),  1,  fp) !=  1)
    {
        perror ("Failed to read data rec from data.rec");
        return errno;
    }
    /* Get position after read: */
    offset = ftell( fp );
    if ( offset == -1 )
    {
        perror (("Failed to get position in data.rec");
        return errno;
    }
    printf("Position after reading: %ld\n", offset);
    /* Store the offset in an index file, along with the key ... */

```

Summary

- Anonymous structures may be defined by omitting the structure tag
 - Initialisation of structures
 - Structure padding and packing can usually be controlled through the use of special compiler flags
 - Members of structures may be declared as *bitfields*
 - Structures may be assigned via `=` and may be passed and returned by value from a function if desired
 - Structures can be read and written to files via *fread()* and *fwrite()*
 - Indexed file techniques are possible using *fseek()*
- 

The identifier immediately following the `struct` keyword is called a *tag*. If it is omitted then an anonymous structure is created.

The compiler is allowed to align the members of structures onto boundaries as it sees fit. This is to make access to the members more efficient. This introduces *padding*. Compilers may provide a *structure packing* option which gives rise to smaller structures. Members may be skewed across non-aligned bytes, meaning that access will be slower.

Members of structures may be declared as bitfields by using the notation "*type name:n*", where *n* is the number of bits required. If *name* is omitted an anonymous member is created which is *n* bits wide. If *n* is zero the next member is aligned onto the next allocation unit boundary.

The ISO committee have officially sanctioned the assignment of structures *of the same type* via the `=` assignment operator. However, structures *may not* be compared with `==` and must be compared *carefully*, otherwise any padding bytes (to which there can be no direct access in your program) will be compared as well.

Structures may be passed by value to functions and may be returned by value too.