Structures

Structures

A structure is similar to an array in that it is an *aggregate* data structure, meaning it is composed of multiple elements or parts. It differs from an array in that each element can be of a different type. (Arrays are homogeneous structures where all elements are of the same type.)

The general syntax of a **struct**:

```
struct tag { members } variable-list;
```

Or formatted appropriately:

```
struct tag
{
  member1
  member2
   ...
  memberN
} variable-list;
```

The *tag* and *variable-list* are optional but the structure declaration *must* end with a semicolon.

Create a structure **type** named **TIME**, (no space is allocated at this point):

Structure	Layout
`struct TIME { int hours; int minutes; int seconds; };`	

```
struct TIME
{
  int hours;
  int minutes;
  int seconds;
};
```

- By including the optional *tag*, we are giving the structure a name.
- Giving the structure a name allows us to create TIME variables in the future.
- This structure declaration acts like a *template*, in that it will be used to create `TIME` objects at a later time. (Think of this structure as you would a built-in type like **int** or **double**.)

This example creates two variables of type **struct** `TIME`, (space is allocated). Compare to an array:

```
struct TIME t1, t2; /* You must include the struct keyword */
int t3[3];          /* An array of 3 integers              */
```

Visually: (the structures and array are uninitialized)

Assigning values to the fields:

```
  /* Set the fields of t1 */
t1.hours = 8;
t1.minutes = 15;
t1.seconds = 0;
```

```
  /* Set the fields of t2 */
t2.hours = 11;
t2.minutes = 59;
t2.seconds = 59;
```

```
  /* Set the elements of t3 */
t3[0] = 8;
t3[1] = 15;
t3[2] = 0;
```

- Accessing members of a structure is a common operation.
- Array elements are accessed anonymously using offsets from the address of the array.
- Structure members are accessed *by name* using the structure member operator, which is the period (or decimal point).
- The structure member operator has a higher precedence than most operators.
- Unlike arrays, a structure can be assigned to another structure:

```
  /* Print out the times */
printf("Time 1 is %02i:%02i:%02i\n", t1.hours, t1.minutes, t1.seconds);
printf("Time 2 is %02i:%02i:%02i\n", t2.hours, t2.minutes, t2.seconds);

  /* Assign all fields from t2 to t1 (Legal for structures, illegal for arrays) */
t1 = t2;

  /* Print out the times again */
printf("Time 1 is %02i:%02i:%02i\n", t1.hours, t1.minutes, t1.seconds);
printf("Time 2 is %02i:%02i:%02i\n", t2.hours, t2.minutes, t2.seconds);

Output:
Time 1 is 08:15:00
Time 2 is 11:59:59
Time 1 is 11:59:59
Time 2 is 11:59:59
```

Initializing Structures

Structures are initialized much like arrays:

Structure declaration

```
struct TIME
{
  int hours;
  int minutes;
  int seconds;
};
```

Initializing `TIME` variables

```
struct TIME t1 = {10, 15, 0}; /* 10:15:00 */
struct TIME t2 = {10, 15};    /* 10:15:00 */
struct TIME t3 = {10};        /* 10:00:00 */
struct TIME t4 = {0};         /* 00:00:00 */

struct TIME t5 = {};          /* Illegal  */
struct TIME t6 = { , , 5};    /* Illegal  */
```

Another example:

Structure declaration

```
struct STUDENT
{
  char first_name[20];
  char last_name[20];
  int age;
  float GPA;
};
```

Initializing `STUDENT` variables

```
  /* Initialization statement */
struct STUDENT s1 = {"Johnny", "Appleseed", 20, 3.75F};

  /* Equivalent assignment statements */
strcpy(s1.first_name, "Johnny");
strcpy(s1.last_name, "Appleseed");
s1.age = 20;
s1.GPA = 3.75F;

  /* Don't try and do this (you can't use assignment with arrays) */
s1.first_name = "Johnny";    /* Illegal */
s1.last_name = "Appleseed"; /* Illegal */
```

Review of array initialization vs. assignment:

```
char string[20]; /* Array of 20 characters, uninitialized */

string = "Johnny";         /* Illegal, "pointer" is const  */
strcpy(string, "Johnny"); /* Proper assignment            */
```

More examples:

Structure declaration

```
struct STUDENT
{
  char first_name[20];
  char last_name[20];
  int age;
  float GPA;
};
```

Initializing `STUDENT` variables

```
  /* Initialize all fields */
struct STUDENT s2 = {"Tom", "Sawyer", 15, 1.30F};

  /* Set age and GPA to 0  */
struct STUDENT s3 = {"Huckleberry", "Finn"};

struct STUDENT s4 = {""};  /* Initialize everything to 0 */
struct STUDENT s5 = {{0}}; /* Initialize everything to 0 */

  /* Initializing arrays */
char first_name[20] = 0;   /* Illegal, need curly braces */
char last_name[20] = {0};  /* Ok                         */
```

Slightly different structure:

Structure declaration

```
struct STUDENT2
{
  char *first_name;
  char *last_name;
  int age;
  float GPA;
};
```

Initializing `STUDENT2` variables

```
  /* Initialization statement */
struct STUDENT2 s1 = {"Johnny", "Appleseed", 20, 3.75F};

  /* Equivalent assignment statements */
s1.first_name = "Johnny";
s1.last_name = "Appleseed";
s1.age = 20;
s1.GPA = 3.75F;

strcpy(s1.first_name, "Johnny");   /* BAD IDEA */
strcpy(s1.last_name, "Appleseed"); /* BAD IDEA */
```

More review of pointers vs. arrays and initialization vs. assignment:

```
char s1[20] = "Digipen"; /* sizeof(s1)?, strlen(s1)? */
char s2[20];             /* sizeof(s2)?, strlen(s2)? */

char *p1 = "Digipen";    /* sizeof(p1)?, strlen(p1)? */
char *p2;                /* sizeof(p2)?, strlen(p2)? */

s2 = "Digipen";        /* Illegal              */
```

```
strcpy(s2, "Digipen"); /* OK                   */

p2 = "Digipen";        /* OK                   */
strcpy(p2, "Digipen"); /* Legal, but very bad */
```

There is a caveat when initializing a structure with less initializer than there are fields in the **`struct`**. The GNU compiler will give you warnings with the `-Wextra` switch:

```
 1. struct TIME
 2. {
 3.   int hours;
 4.   int minutes;
 5.   int seconds;
 6. };
 7.
 8. struct TIME t1 = {10, 15, 0}; /* 10:15:00 */
 9. struct TIME t2 = {10, 15};    /* 10:15:00 */
10. struct TIME t3 = {10};        /* 10:00:00 */
11. struct TIME t4 = {0};         /* 00:00:00 */
```

Warnings:

```
main2.c:9: warning: missing initializer
main2.c:9: warning: (near initialization for `t2.seconds')
main2.c:10: warning: missing initializer
main2.c:10: warning: (near initialization for `t3.minutes')
main2.c:11: warning: missing initializer
main2.c:11: warning: (near initialization for `t4.minutes')
```

Starting with version 4.0.4, this can be suppressed with `-Wno-missing-field-initializers`.

Structures as Parameters

Structures can be passed to functions just like any other value. However, they are different than arrays in that they are *passed by value*, meaning that the entire structure is *copied* onto the stack. This is actually a very big difference, and, as you can imagine, it can be a very expensive operation.

Given this structure:

```
struct TIME
{
  int hours;   /* 4 bytes */
  int minutes; /* 4 bytes */
  int seconds; /* 4 bytes */
};
```

Passing a TIME structure to a function:

Function to print a TIME

```
void print_time(struct TIME t)
{
  printf("The time is %02i:%02i:%02i\n",
         t.hours, t.minutes, t.seconds);
}
```

Calling the function

```
void foo(void)
{
    /* Create time of 10:30:15 */
  struct TIME t = {10, 30, 15};

    /* Pass by value and print */
  print_time(t);
}

Output:
The time is 10:30:15
```

A more expensive example:

```
struct STUDENT
{
  char first_name[20]; /* 20 bytes */
  char last_name[20];  /* 20 bytes */
  int age;             /*  4 bytes */
  float GPA;           /*  4 bytes */
};
```

Function to print a STUDENT

Calling the function

```c
void print_student(struct STUDENT s)
{
  printf("Name: %s %s\n", s.first_name, s.last_name);
  printf(" Age: %i\n", s.age);
  printf(" GPA: %.2f\n", s.GPA);
}
```

```c
void foo(void)
{
  struct STUDENT s1 = {"Johnny",
                       "Appleseed",
                       20,
                       3.75F
                      };

  print_student(s1);
}
```

```
Output:
Name: Johnny Appleseed
 Age: 20
 GPA: 3.75
```

Structures as Parameters (revisited)

Since the default method for passing structures to functions is pass-by-value, we should modify the function so that it's more efficient. We do that by simply passing the structure by address instead.

Calling the function

```c
void f12(void)
{
  struct STUDENT s1 = {"Johnny",
                       "Appleseed",
                       20,
                       3.75F
                      };
  print_student2(&s1);
}
```

Function to print a STUDENT (more efficient)

```c
void print_student2(const struct STUDENT *s)
{
  printf("Name: %s %s\n", s->first_name, s->last_name);
  printf(" Age: %i\n", s->age);
  printf(" GPA: %.2f\n", s->GPA);
}
```

```
Output:
Name: Johnny Appleseed
 Age: 20
 GPA: 3.75
```

Note:

- In practice, you will rarely pass a structure (by value) to a function. You will almost always pass a pointer instead.
- Passing structures to functions by value can be very expensive. Unfortunately, compilers won't warn you when you do so.
- Be sure to remember to use the `const` keyword if your function does not modify the structure that was passed in.

Structures as Members

Structures can contain almost any data type, including other structures. Sometimes these are called *nested* structures.

```c
#define MAX_PATH 12

struct DATE
{
  int month;
  int day;
  int year;
};

struct TIME
{
  int hours;
  int minutes;
  int seconds;
};

struct DATETIME
{
  struct DATE date;
  struct TIME time;
};

struct FILEINFO
{
  int length;
  char name[MAX_PATH];
  struct DATETIME dt;
};
```

Given this code:

```c
struct FILEINFO fi;
```

We can visualize the struct in memory like this:

Now highlighting the two fields of the DATETIME struct:

Now highlighting the fields of the DATE and TIME structs:

Example:

```c
void f1(void)
{
  struct FILEINFO fi;   /* Create FILEINFO struct on stack */

    /* Set date to 7/4/2005 */
  fi.dt.date.day = 4;
  fi.dt.date.month = 7;
  fi.dt.date.year = 2005;

    /* Set time to 9:30 am */
  fi.dt.time.hours = 9;
  fi.dt.time.minutes = 30;
  fi.dt.time.seconds = 0;
```

```
  fi.length = 1024;           /* Set length */
  strcpy(fi.name, "foo.txt"); /* Set name */
}
```

An example using initialization:

```
struct FILEINFO fi = {1024, "foo.txt", { {7, 4, 2005}, {9, 30, 0} } };
```

A very fast way to set everything to 0:

```
struct FILEINFO fi = {0};
```

Same operations using a pointer to the structure:

```
void f2(void)
{
  struct FILEINFO fi;          /* Create FILEINFO struct on stack */
  struct FILEINFO *pfi = &fi; /* Pointer to a FILEINFO struct    */

    /* Set date to 7/4/2005 */
  (*pfi).dt.date.day = 4;
  (*pfi).dt.date.month = 7;
  (*pfi).dt.date.year = 2005;

    /* Set time to 9:30 am */
  (*pfi).dt.time.hours = 9;
  (*pfi).dt.time.minutes = 30;
  (*pfi).dt.time.seconds = 0;

  (*pfi).length = 1024;           /* Set length */
  strcpy((*pfi).name, "foo.txt"); /* Set name */
}
```

Due to the order of precedence, we need the parentheses above. Otherwise:

Accessing a member of a pointer:

```
  /*  error: request for member 'length' in something not a structure or union */
pfi.length = 1024;
```

Closer look:

```
Expression     Description
------------------------------------------------------------------
pfi            A pointer to a FILEINFO struct
*pfi           A FILEINFO struct
(*pfi).        Accessing a member of a FILEINFO struct
pfi->          Accessing a member of a FILEINFO struct (shorthand)
```

The *structure pointer operator* (or informally the *arrow operator*) is another programmer convenience along the same vein as the subscript operator and is high on the precedence chart. It performs the indirection "behind the scenes" so:

```
(*pfi).  is the same as   pfi->
```

That's why using the structure pointer operator on a structure is illegal; we're trying to dereference something (a structure) that isn't a pointer. And that's a no-no. Same example using the structure pointer operator:

```
  /* Set date to 7/4/2005 */
pfi->dt.date.month = 7;
pfi->dt.date.day = 4;
pfi->dt.date.year = 2005;

  /* Set time to 9:30 am */
pfi->dt.time.hours = 9;
pfi->dt.time.minutes = 30;
pfi->dt.time.seconds = 0;

pfi->length = 1024;           /* Set length */
strcpy(pfi->name, "foo.txt"); /* Set name */
```

Arrays vs. Structures vs. Built-in Types

Unlike arrays, which prohibit most aggregate operations, it is possible in some cases to manipulate structures as a whole.

Operation	Arrays	Structures	Built-in types (e.g. `int`)
Arithmetic	No	No	Yes
Assignment	No	Yes	Yes
Comparison	No	No	Yes
Input/Output(e.g. printf)	No (except strings)	No	Yes

Parameter passing	By address only	By address or value	Yes
Return from function	No	Yes	Yes

Summary of `struct` Syntax

The general form:

```
struct tag { members } variable-list;
```

Create a structure named **TIME**, (no space is allocated):

```
struct TIME
{
  int hours;
  int minutes;
  int seconds;
};
```

Create two variables of type **struct** TIME, (space is allocated):

```
struct TIME t1, t2; /* You need the struct keyword */
```

We can do both in one step:

```
struct TIME
{
  int hours;
  int minutes;
  int seconds;
}t1, t2;            /* This allocates space            */

struct TIME t3, t4; /* Create more TIME variables here */
```

Leaving off the tag creates an anonymous structure:

```
struct     /* No name given to this struct */
{
  int hours;
  int minutes;
  int seconds;
}t1, t2;  /* We won't be able to create others later */
```

Create a new type with the **typedef** keyword:

```
typedef struct  /* use typedef keyword to create a new type */
{
  int hours;
  int minutes;
  int seconds;
}TIME;           /* TIME is a type, not a variable    */

TIME t1, t2;     /* Don't need the struct keyword now */
```

Using a **typedef**:

```
typedef struct TIME Time; /* Time is a new type */
```

Now we don't need the the `struct` keyword:

```
Time t1, t2;        /* Create to TIME structs */
struct TIME t3, t4; /* This still works       */
```

We can create the `typedef` when we declare the `struct`:

```
typedef struct TIME /* tag name     */
{
  int hours;
  int minutes;
  int seconds;
}Time;            /* typedef name */

Time t1, t2;        /* use typedef name                   */
struct TIME t3, t4;  /* use tag name, needs struct keyword */
```

The *tag* name and **typedef** name can be the same:

```
typedef struct TIME /* tag name           */
{
  int hours;
  int minutes;
  int seconds;
```

```c
}TIME;                /* typedef same as tag */

TIME t1, t2;          /* use typedef name                   */
struct TIME t3, t4;   /* use tag name, needs struct keyword */

TIME times[10];     /* an array of 10 TIME structs          */
TIME *pt;           /* a pointer to a TIME struct           */
TIME foo(TIME *p);  /* function take a TIME struct pointer  */
                    /*   and returns a TIME struct          */
```

Self-referencing structures

Before any data type can be used to create a variable, the size of the type must be known to the compiler:

```c
struct NODE
{
  int value;
  struct NODE next;  /* illegal */
};
```

Since the compiler hasn't fully "seen" the NODE struct, it can't be used anywhere, even inside itself. However, this works:

```c
struct NODE
{
  int value;
  struct NODE *next; /* OK */
};
```

Since all pointers are of the same size, the compiler will accept this. The compiler doesn't fully know what's in a NODE struct (and doesn't need to know yet), but it knows the size of a pointer to it.

Two structures that are mutually dependent on each other won't pose any problems. In the source file, one of the declarations must come *after* the other. The use of the `struct` keyword gives the compiler enough information:

```c
struct A
{
  int value;
  struct B *b; /* B is a struct */
};

struct B
{
  int value;
  struct A *a; /* A is a struct */
};
```

Unions

Suppose we want to parse simple expressions and we need to store information about each symbol in the expression. We could use a structure like this:

```c
enum Kind {OPERATOR, INTEGER, FLOAT, IDENTIFIER};

struct Symbol
{
  enum Kind kind;
  char op;
  int ival;
  float fval;
  char id;
};
```

A Symbol struct in memory would look something like this:

If we wanted to store the information about this expression:

$A + 23 * 3.14$

We could do this:

```c
void main(void)
{
  struct Symbol sym1, sym2, sym3, sym4, sym5;

  sym1.kind = IDENTIFIER;
  sym1.id = 'A';

  sym2.kind = OPERATOR;
  sym2.op = '+';

  sym3.kind = INTEGER;
  sym3.ival = 23;

  sym4.kind = OPERATOR;
  sym4.op = '*';

  sym5.kind = FLOAT;
  sym5.fval = 3.14F;
```

```
}
```

Memory usage would look something like this:

When dealing with mutually exclusive data members, a better solution is to create a **union** and use that instead:

The union

```
union SYMBOL_DATA
{
  char op;
  int ival;
  float fval;
  char id;
};
```

The new struct

```
struct NewSymbol
{
  enum Kind kind;
  union SYMBOL_DATA data;
};
```

Note that `sizeof(SYMBOL_DATA)` is 4, since that's the size of the largest member.

The same rules for naming structs apply to unions as well, so we could even typedef the union:

The union

```
typedef union
{
  char op;
  int ival;
  float fval;
  char id;
}SYMBOL_DATA;
```

The new struct

```
struct NewSymbol
{
  enum Kind kind;
  SYMBOL_DATA data;
};
```

Often, however, if the union is not intended to be used outside of a structure, we define it within the structure definition itself *without* the tag:

```
struct NewSymbol
{
  enum Kind kind;
  union
  {
    char op;
    int ival;
    float fval;
    char id;
  } data;
};
```

Our NewSymbol struct would look like this in memory:

Our code needs to be modified slightly:

New Code (`union`)

```
void main(void)
{
  struct NewSymbol sym1, sym2, sym3, sym4, sym5;

  sym1.kind = IDENTIFIER;
  sym1.data.id = 'A';

  sym2.kind = OPERATOR;
  sym2.data.op = '+';

  sym3.kind = INTEGER;
  sym3.data.ival = 23;

  sym4.kind = OPERATOR;
  sym4.data.op = '*';

  sym5.kind = FLOAT;
  sym5.data.fval = 3.14F;
}
```

Old Code (`struct`)

```
void main(void)
{
  struct Symbol sym1, sym2, sym3, sym4, sym5;

  sym1.kind = IDENTIFIER;
  sym1.id = 'A';

  sym2.kind = OPERATOR;
  sym2.op = '+';

  sym3.kind = INTEGER;
  sym3.ival = 23;

  sym4.kind = OPERATOR;
  sym4.op = '*';

  sym5.kind = FLOAT;
  sym5.fval = 3.14F;
}
```

And the memory usage with unions would look something like this:

Using a union to get at individual bytes of data:

```
void TestUnion(void)
{
  union
```

```
  {
    int i;
    unsigned char bytes[4];
  }val;

  val.i = 257;
  printf("%3i  %3i  %3i  %3i\n",
    val.bytes[0], val.bytes[1], val.bytes[2], val.bytes[3]);

  val.i = 32767;
  printf("%3i  %3i  %3i  %3i\n",
    val.bytes[0], val.bytes[1], val.bytes[2], val.bytes[3]);

  val.i = 32768;
  printf("%3i  %3i  %3i  %3i\n",
    val.bytes[0], val.bytes[1], val.bytes[2], val.bytes[3]);
}
```

This prints out:

```
  1    1    0    0
255  127    0    0
  0  128    0    0
```

The values in binary:

```
  257: 00000000 00000000 00000001 00000001
32767: 00000000 00000000 01111111 11111111
32768: 00000000 00000000 10000000 00000000
```

As little-endian:

```
  257: 00000001 00000001 00000000 00000000
32767: 11111111 01111111 00000000 00000000
32768: 00000000 10000000 00000000 00000000
```

Changing the union to this:

```
union
{
  int i;
  signed char bytes[4];
}val;
```

Gives this output (the bit patterns are the same):

```
  1     1     0     0
 -1   127     0     0
  0  -128     0     0
```

Initializing Unions

The type of the initializer must be the same type as the first member of the union:

```
struct NewSymbol sym1 = {OPERATOR, {'+'} }; /* fine, op is first member    */
struct NewSymbol sym2 = {FLOAT, {3.14} };   /* this won't work as expected */
```

Given the code above, what is printed below?

```
printf("%c, %i, %f, %c\n", sym1.data.op, sym1.data.ival,
                           sym1.data.fval, sym1.data.id);

printf("%c, %i, %f, %c\n", sym2.data.op, sym2.data.ival,
                           sym2.data.fval, sym2.data.id);
```

Structure Alignment

What will be printed out by the following code?

```
struct Symbol sym1;

printf("sizeof(sym1.kind) =  %i\n", sizeof(sym1.kind));
printf("sizeof(sym1.op)   =  %i\n", sizeof(sym1.op));
printf("sizeof(sym1.ival) =  %i\n", sizeof(sym1.ival));
printf("sizeof(sym1.fval) =  %i\n", sizeof(sym1.fval));
printf("sizeof(sym1.id)   =  %i\n", sizeof(sym1.id));
printf("sizeof(sym1)      = %i\n", sizeof(sym1));
```

Recall the *Symbol* structure and diagram:

```
struct Symbol
{
  enum Kind kind;
  char op;
  int ival;
  float fval;
  char id;
};
```

The actual output on from `gcc` is this:

```
sizeof(sym1.kind) =  4
sizeof(sym1.op)   =  1
sizeof(sym1.ival) =  4
sizeof(sym1.fval) =  4
sizeof(sym1.id)   =  1
sizeof(sym1)      = 20
```

But

```
4 + 1 + 4 + 4 + 1 != 20
```

What's going on?

- By default, data is aligned on "natural" boundaries, which are addresses that are multiples of the word size of the computer.
- Most computers we deal with today are 32-bit (4-byte) machines, so data is stored at address that are multiples of 4.
- This means that data that is less that 4-bytes causes "wasted" space in memory.
- You can override this default using a compiler directive.

So, a more accurate diagram of the *Symbol* structure would look like this:

which is 20 bytes in size because all data is aligned on 4-byte boundaries. This means that the **char** data is actually padded with 3 bytes extra so the data that *follows* will be aligned properly. (Note the term "follows").

To change the structure alignment, use this compiler directive:

```
#pragma pack
```

where n is the alignment. The *n* specifies the value, in bytes, to be used for packing. In Microsoft Visual Studio, the default value for *n* is 8. Valid values are 1, 2, 4, 8, and 16.

The alignment of a member will be on a boundary that is either a multiple of *n* or a multiple of the size of the member, whichever is *smaller*.

For example, to align the fields of the *Symbol* structure on 2-byte boundaries:

```
#pragma pack(2)     /* align on 2-byte boundaries */
struct Symbol
{
  enum Kind kind;
  char op;
  int ival;
  float fval;
  char id;
};
#pragma pack()      /* restore compiler's default alignment setting */
```

Now, it would look like this in memory:

To align the fields on 1-byte boundaries:

```
#pragma pack(1)     /* align on 1-byte boundaries */
struct Symbol
{
  enum Kind kind;
  char op;
  int ival;
  float fval;
  char id;
};
#pragma pack()      /* restore compiler's alignment setting */
```

Now, it would look like this in memory:

An actual printout from Microsoft's compiler:

```
   #pragma pack(4)              #pragma pack(2)             #pragma pack(1)
--------------------------------------------------------------------------------
     &sym1 = 0012FEDC            &sym1 = 0012FEE0            &sym1 = 0012FEE0
&sym1.kind = 0012FEDC       &sym1.kind = 0012FEE0       &sym1.kind = 0012FEE0
  &sym1.op = 0012FEE0         &sym1.op = 0012FEE4         &sym1.op = 0012FEE4
&sym1.ival = 0012FEE4       &sym1.ival = 0012FEE6       &sym1.ival = 0012FEE5
&sym1.fval = 0012FEE8       &sym1.fval = 0012FEEA       &sym1.fval = 0012FEE9
  &sym1.id = 0012FEEC         &sym1.id = 0012FEEE         &sym1.id = 0012FEED
```

```
sizeof(sym1.kind) =  4       sizeof(sym1.kind) =  4       sizeof(sym1.kind) =  4
sizeof(sym1.op)   =  1       sizeof(sym1.op)   =  1       sizeof(sym1.op)   =  1
sizeof(sym1.ival) =  4       sizeof(sym1.ival) =  4       sizeof(sym1.ival) =  4
sizeof(sym1.fval) =  4       sizeof(sym1.fval) =  4       sizeof(sym1.fval) =  4
sizeof(sym1.id)   =  1       sizeof(sym1.id)   =  1       sizeof(sym1.id)   =  1
sizeof(sym1)      = 20       sizeof(sym1)      = 16       sizeof(sym1)      = 14
```

The code to print the addresses:

```
struct Symbol sym1;

printf("&sym1 = %p\n", &sym1);
printf("&sym1.kind = %p\n", &sym1.kind);
printf("&sym1.op = %p\n", &sym1.op);
printf("&sym1.ival = %p\n", &sym1.ival);
printf("&sym1.fval = %p\n", &sym1.fval);
printf("&sym1.id = %p\n", &sym1.id);
```

Note that if we used any of these alignments:

```
#pragma pack(4)     /* align on 4-byte boundaries */
#pragma pack(8)     /* align on 8-byte boundaries */
#pragma pack(16)    /* align on 16-byte boundaries */
```

the layout would still look like this:

This is because none of the members of the structure are larger than 4 bytes (so they will never need to be aligned on 8-byte or 16-byte boundaries.)

Notes

- Choosing your alignment is a trade-off between speed and memory.
- When you require lots of structures (e.g. large arrays of them) with some small data fields and large ones, aligning on large boundaries could waste a lot of space.
- Accessing data that is **not** aligned on the natural boundaries of a computer is slower. (Memory may be accessed by some hardware only on the word boundary.)
- Using the **pack** directive, you can selectively choose which structures to align and how to align them.

Pragmas are not part of the ANSI C language and are compiler-dependent. Although most compilers support the **pack** pragma, you should be aware that different compilers may have different default alignments. Also, MS says that the default alignment for Win32 is 8 bytes, not 4. You should consult the documentation for your compiler to determine the behavior.

This is from the top of ***stdlib.h*** from MS VC++ 6.0:

```
#ifdef  _MSC_VER
/*
 * Currently, all MS C compilers for Win32 platforms default to 8 byte
 * alignment.
 */
#pragma pack(push,8)
```

Given these two logically equivalent structures, what are the ramifications of laying out the members in these ways?

```
struct BEAVIS          struct BUTTHEAD
{                      {
  char a;                char a;
  double b;              char c;
  char c;                char e;
  double d;              char g;
  char e;                double b;
  double f;              double d;
  char g;                double f;
  double h;              double h;
};                     };
```

[Structure packing with GNU compilers]()

Accessing Structures Using Pointer/Offset

Much like arrays, the compiler converts structure.member notation into pointer + offset notation:

```
structvar.member ==> *([address of structvar] + [offset of member])
```

So using the *Symbol* structure example above with the address of *sym1* being 100:

```
struct Symbol sym1;

sym1.ival ==> *(&sym1 + 8)
sym1.id   ==> *(&sym1 + 16)
```

Or more accurately:

```
sym1.ival ==> *( (int *)( (char *)&sym1 + 8) )
sym1.id   ==> *( (char *)&sym1 + 16 )
```

Note that the code above assumes structures are aligned on 4-byte boundaries:

Code to print the values of a Symbol structure variable using pointer/offset with 4-byte alignment:

```
TestStructOffset4(void)
{
  struct Symbol sym1 = {IDENTIFIER, '+', 123, 3.14F, 'A'};
  char *psym = (char *)&sym1;

  int kind   = *((int *)(psym + 0));    /* 3    */
  char op    = *(psym + 4);             /* '+'  */
  int ival   = *((int *)(psym + 8));    /* 123  */
  float fval = *((float *)(psym + 12)); /* 3.14 */
  char id    = *(psym + 16);            /* 'A'  */

    /* 3, +, 123, 3.140000, A */
  printf("%i, %c, %i, %f, %c\n", kind, op, ival, fval, id);
}
```

Code to print the values of a Symbol structure variable using pointer/offset with 1-byte alignment:

```
TestStructOffset1(void)
{
  struct Symbol sym1 = {IDENTIFIER, '+', 123, 3.14F, 'A'};
  char *psym = (char *)&sym1;

  int kind   = *((int *)(psym + 0));    /* 3    */
  char op    = *(psym + 4);             /* '+'  */
  int ival   = *((int *)(psym + 5));    /* 123  */
  float fval = *((float *)(psym + 9));  /* 3.14 */
  char id    = *(psym + 13);            /* 'A'  */

    /* 3, +, 123, 3.140000, A */
  printf("%i, %c, %i, %f, %c\n", kind, op, ival, fval, id);
}
```

Bit Fields in Structures

- C allows a structure to have fields which are smaller than a *char* (8 bits).
- Specifically, they can have fields as small as a single bit.
- These fields are called *bit fields* and their type is either **int**, **signed int** or **unsigned int**.
- You should always specify either *signed* or *unsigned* because the type of **int** in a bit field is implementation-dependent. (The original C definition only allowed unsigned int, but ANSI C allows all three types.)

Suppose we wanted to track these attributes of some object:

```
  /* Variables for each attribute of some object              */
  /* Comments represent the range of values for the attribute */
unsigned char level;   /* 0 - 3    */
unsigned char power;   /* 0 - 63   */
unsigned short range;  /* 0 - 1023 */
unsigned char armor;   /* 0 - 15   */
unsigned short health; /* 0 - 511  */
unsigned char grade;   /* 0 - 1    */
```

Given the sizes of each data type, we could say that the *minimum* amount of memory require to hold these attributes is 8 bytes. However, given a 32-bit computer, it's possible that the amount of memory required could actually be 24 bytes, depending on where these variables exist. Why?

Declared local to a function: (on the stack)

Microsoft	GNU	Borland
Address of level = 0012FF28	Address of level = 0x22F047	Address of level = 0012FF83
Address of power = 0012FF24	Address of power = 0x22F046	Address of power = 0012FF82
Address of range = 0012FF20	Address of range = 0x22F044	Address of range = 0012FF80
Address of armor = 0012FF1C	Address of armor = 0x22F043	Address of armor = 0012FF7F
Address of health = 0012FF18	Address of health = 0x22F040	Address of health = 0012FF7C
Address of grade = 0012FF14	Address of grade = 0x22F03F	Address of grade = 0012FF7B

Declared globally:

Microsoft	GNU	Borland
Address of level = 004310BE	Address of level = 0x405030	Address of level = 004122E0
Address of power = 004310BC	Address of power = 0x406060	Address of power = 004122E1
Address of range = 004312FC	Address of range = 0x406050	Address of range = 004122E2
Address of armor = 004310BD	Address of armor = 0x406080	Address of armor = 004122E4
Address of health = 00431142	Address of health = 0x407110	Address of health = 004122E6
Address of grade = 004310BF	Address of grade = 0x407160	Address of grade = 004122E8

Our first attempt to save memory is to put them in a structure:

```
/* Put into a struct */
typedef struct
{
  unsigned char level;   /* 0 - 3    */
  unsigned char power;   /* 0 - 63   */
  unsigned short range;  /* 0 - 1023 */
  unsigned char armor;   /* 0 - 15   */
  unsigned short health; /* 0 - 511  */
  unsigned char grade;   /* 0 - 1    */
}ENTITY_ATTRS;
```

What is the memory requirements for this struct? Of course, it depends on how the compiler is packing structures. Given a default pack value of 8, the layout looks like this:

What about this structure:

```
/* Put into a struct and pack */
#pragma pack(1)    /* align on 1-byte boundaries */
typedef struct
{
  unsigned char level;   /* 0 - 3    */
  unsigned char power;   /* 0 - 63   */
  unsigned short range;  /* 0 - 1023 */
  unsigned char armor;   /* 0 - 15   */
  unsigned short health; /* 0 - 511  */
  unsigned char grade;   /* 0 - 1    */
}ENTITY_ATTRS;
#pragma pack()
```

This code yields a layout like this:

Of course, looking closer, we realize that we only need 32 bits for all 6 variables, so we'll just use an unsigned integer to store the values:

To set the fields to these values:

```
level = 3;     /*  2 bits wide */
power = 32;    /*  6 bits wide */
range = 1000;  /* 10 bits wide */
armor = 7;     /*  4 bits wide */
health = 300;  /*  9 bits wide */
grade = 1;     /*  1 bit wide  */
```

We can use "simple" bit manipulation:

```
unsigned int attrs;

attrs = 3 << 30;              /* set level to 3    */
attrs = attrs | (32 << 24);   /* set power to 32   */
attrs = attrs | (1000 << 14); /* set range to 1000 */
attrs = attrs | (7 << 10);    /* set armor to 7    */
attrs = attrs | (300 << 1);   /* set health to 300 */
attrs = attrs | 1;            /* set grade to 1    */
```

After shifting, we **OR** all of the values together:

```
Left shifts             Binary
-----------------------------------------------
   3 << 30     11000000000000000000000000000000
  32 << 24       100000000000000000000000000000
1000 << 14             111110100000000000000000
   7 << 10                       01110000000000
 300 <<  1                           1001011000
        1                                     1
-----------------------------------------------
               11100000111110100001111001011001
```

Of course, there's got to be a better way...

```
/* Use bitfields for the attributes */
typedef struct
{
  unsigned int level  :  2; /* 0 - 3    */
  unsigned int power  :  6; /* 0 - 63   */
  unsigned int range  : 10; /* 0 - 1023 */
  unsigned int armor  :  4; /* 0 - 15   */
  unsigned int health :  9; /* 0 - 511  */
  unsigned int grade  :  1; /* 0 - 1    */
}ENTITY_ATTRS_B;
```

The **sizeof** the structure above is 4, which is the same size as the unsigned integer used before. However, this structure allows for a much cleaner syntax:

```
ENTITY_ATTRS_B attrs;

  /* Easier to read, understand, and self-documenting */
attrs.level = 3;
attrs.power = 32;
```

```
attrs.range = 1000;
attrs.armor = 7;
attrs.health = 300;
attrs.grade = 1;
```

Much like a lot of syntax in C, the compiler is doing the work for you behind-the-scenes.

Notes

- You cannot take the address of a bit field (most computers can't address "odd" sized fields)
- Bit fields are supported in all compilers, but the implementations may differ.
- Some bit fields are stored left to right on one compiler, but right to left on another.
- Some compilers may pack the bits of two fields together, some may add padding to align on a word boundary.
- The maximum number of bits in a field may differ from one compiler to another, especially when dealing with 16-bit, 32-bit, or 64-bit compilers.
- For the most part, the compiler-generated code is similar to what the programmer would write. The shifting, masking, and'ing, or'ing may still need to be done.
- The elegance in the source code needs to be weighed against possible loss of portability.

Alignment Example Using BITMAPFILEHEADER

Given these definitions:

```
typedef unsigned short WORD;
typedef unsigned long DWORD;

typedef struct tagBITMAPFILEHEADER {
  WORD    bfType;       /* 2 bytes */
  DWORD   bfSize;       /* 4 bytes */
  WORD    bfReserved1;  /* 2 bytes */
  WORD    bfReserved2;  /* 2 bytes */
  DWORD   bfOffBits;    /* 4 bytes */
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

And this function:

```
void PrintBitmapHeader(BITMAPFILEHEADER *header)
{
  printf("Type: %c%c (%04X)\n", header->bfType & 0xFF,
                                header->bfType >> 8,
                                header->bfType);
  printf("Size: %lu (%08X)\n", header->bfSize, header->bfSize);
  printf("Res1: %lu (%04X)\n", header->bfReserved1, header->bfReserved1);
  printf("Res2: %lu (%04X)\n", header->bfReserved2, header->bfReserved2);
  printf("Offs: %lu (%08X)\n", header->bfOffBits, header->bfOffBits);
}
```

What should this program display? (Hint: the size of the file is 207,158 bytes, the offset to the bitmap itself is 1078 bytes, and the two reserved fields are 0.)

```
void main(void)
{
  BITMAPFILEHEADER header;
  FILE *fp = fopen("foo.bmp", "rb");
  assert(fp);

  fread(&header, sizeof(BITMAPFILEHEADER), 1, fp);
  PrintBitmapHeader(&header);
  fclose(fp);
}
```

Given the "hint" above, the expected output should be:

```
Type: BM (4D42)
Size: 207158 (00032936)
Res1: 0 (0000)
Res2: 0 (0000)
Offs: 1078 (00000436)
```

However, the actual output is:

```
Type: BM (4D42)
Size: 3 (00000003)
Res1: 0 (0000)
Res2: 1078 (0436)
Offs: 2621440 (00280000)
```

Why is this incorrect?

The actual bytes in the bitmap file look like this:

```
42 4D 36 29 03 00 00 00 00 00 36 04 00 00 28 00 . . . .
```

Separated by fields it looks like this:

```
 Type       Size        Res1     Res2       Offset        Other stuff
```

```
42 4D | 36 29 03 00 | 00 00 | 00 00 | 36 04 00 00 | 28 00 . . . .
```

And the BITMAPFILEHEADER structure in memory looks like this:

Why is the structure aligned like this? This means that:

```
  sizeof(BITMAPFILEHEADER) == 16
```

Reading the header with the code:

```
fread(&header, sizeof(BITMAPFILEHEADER), 1, fp);
```

causes the first 16 bytes (sizeof(BITMAPFILEHEADER)) of the file to be read into the buffer (memory pointed to by &header), which yields:

Which gives the values we saw (adjusting for little-endian):

```
Member              Hex        Decimal
---------------------------------------
bfType               4D42        19778
bfSize           00000003            3
bfReserved1          0000            0
bfReserved2          0436         1078
bfOffBits        00280000      2621440
```

Again, the correct output should be:

```
Type: BM (4D42)
Size: 207158
Res1: 0
Res2: 0
Offs: 1078
```

To achieve the correct results, we need to pack the structure:

```
#pragma pack(2)
typedef struct tagBITMAPFILEHEADER {
  WORD    bfType;       /* 2 bytes */
  DWORD   bfSize;       /* 4 bytes */
  WORD    bfReserved1;  /* 2 bytes */
  WORD    bfReserved2;  /* 2 bytes */
  DWORD   bfOffBits;    /* 4 bytes */
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;
#pragma pack()
```

Now, the structure in memory looks like this:

and:

```
  sizeof(BITMAPFILEHEADER) == 14
```

so now when we read in 14 bytes, the structure is filled like this:

which gives the correct values:

```
Member              Hex        Decimal
---------------------------------------
bfType               4D42        19778
bfSize           00032936       207158
bfReserved1          0000            0
bfReserved2          0000            0
bfOffBits        00000436         1078
```

The actual structure in *wingdi.h* looks like this:

```
#include <pshpack2.h>
typedef struct tagBITMAPFILEHEADER {
        WORD    bfType;
        DWORD   bfSize;
        WORD    bfReserved1;
        WORD    bfReserved2;
        DWORD   bfOffBits;
} BITMAPFILEHEADER, FAR *LPBITMAPFILEHEADER, *PBITMAPFILEHEADER;
#include <poppack.h>
```

and *pshpack2.h* looks like this:

```
#if ! (defined(lint) || defined(_lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 ) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
```

```
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push)
#endif
#pragma pack(2)
#else
#pragma pack(2)
#endif
#endif /* ! (defined(lint) || defined(_lint) || defined(RC_INVOKED)) */
```

[Complete listings]

Addresses and values at different **pack** values:

```
#pragma pack(1)              #pragma pack(2)              #pragma pack(4)
-----------------------------------------------------------------------------------
bfType = 0012FEE0            bfType = 0012FEE0            bfType = 0012FEE0
bfSize = 0012FEE2            bfSize = 0012FEE2            bfSize = 0012FEE4
bfRes1 = 0012FEE6            bfRes1 = 0012FEE6            bfRes1 = 0012FEE8
bfRes2 = 0012FEE8            bfRes2 = 0012FEE8            bfRes2 = 0012FEEA
bfOffs = 0012FEEA            bfOffs = 0012FEEA            bfOffs = 0012FEEC
Type: BM (4D42)              Type: BM (4D42)              Type: BM (4D42)
Size: 207158 (00032936)      Size: 207158 (00032936)      Size: 3 (00000003)
Res1: 0 (0000)               Res1: 0 (0000)               Res1: 0 (0000)
Res2: 0 (0000)               Res2: 0 (0000)               Res2: 1078 (0436)
Offs: 1078 (00000436)        Offs: 1078 (00000436)        Offs: 2621440 (00280000)
```