

# Structures in C

CSE 2421


Recommended Reading: *Pointers On C*, Chapter 10, through Section 10.3



# Basics on Structures

- An aggregate data type is one that can hold more than one individual piece of data at a time.
- C has two aggregate data types, arrays and structures.
- A structure is a collection of values, called *members*, but the members of a structure *may be of different types*, unlike the elements of an array.
- Array elements can be accessed with an index or subscript because all of the elements are of the same size/type, and because they are stored at contiguous locations in memory.
- The situation is different with structures. Because a structure's members can be of different sizes, subscripts or indexes cannot be used to access them.
- This distinction is important. A structure is *not* an array of its members. Unlike an array name, the name of a structure variable is not replaced with a pointer when it is used in an expression.
- Subscripts/indexes cannot be used on a structure variable to access its members.
- Instead, structure members are given names (identifiers), and are accessed by those names.

# Structure Basics cont.

- ▶ The following can all be done with structures:
    - Structures may be passed as arguments to functions;
    - They may be returned by functions;
    - Structures of the same type may be assigned to one another;
    - We can declare pointers to structures;
    - We can take the address of a structure variable;
    - We can declare arrays of structures.
- 

# Structure Declaration

- ▶ Here is the format of a structure declaration:

struct ***tag*** { member-list with type of each member } ***variable list*** ;

- ▶ **struct** is a keyword in C.
- ▶ Generally, the ***tag*** field and the ***variable-list*** field are each optional, but at least one of them is required:
  - If the tag is omitted, only the variables with identifiers which appear in the variable-list field are declared to be of this structure type, and no other variables of this type can be declared subsequently.
  - If the tag field is included, then the variable-list field can be empty or omitted. In this case, the tag can be used later to declare variables of this structure type. ***The tag field is needed if you wish to use malloc() or calloc().***
- ▶ Examples on the next page.

# Structure Declaration Examples

Consider two structure declarations:

```
struct {  
    int a;  
    float b;  
} x;          /* If these two structures are declared in the */  
struct {      /* same program, x is of a different type from */  
    int a;    /* the elements of y, and the type of *z is also */  
    float b;  /* different from the type of x. */  
} y[10], *z;
```

Therefore, the following are **invalid**:

```
z = &x; /* Invalid – compilation will fail (compiler gives an error). */  
y[2] = x; /* Also invalid – compilation will fail (compiler gives an error). */
```

# Structure Declaration Examples cont.

- ▶ If we want x, y and \*z to be of the same type, we have two options:

1) Use a tag (suppose the tag is Simple):

```
struct Simple{  
    int a;  
    float b;  
} ;  
struct Simple x, y[10], *z; /*Notice that the keyword struct is required here*/
```

2) Use no tag, but declare all of the variables together following the member-list:

```
struct {  
    int a;  
    float b;  
} x, y[10], *z;
```

# Structure Declarations cont.

- ▶ Another technique is to create a new type using typedef:

```
typedef struct {  
    int a;  
    float b;  
} Simple;
```

- ▶ Some C software writers prefer not to use typedef for structures, in order to improve documentation, because it is not clear from the name of the type that variables of the type are structure variables if we use typedef.
- ▶ This technique has almost the same effect as using a structure tag. What is different is that Simple is now a **type name**, rather than a structure tag, so subsequent declarations of variables of this type will look like this:

```
Simple x, y[10], *z;      /* This declares x, the elements of y[10], */  
                          /* and *z to be of the same type          */
```



# Structure Declaration cont.

- ▶ Any of the above techniques will work, but the use of tags and struct is preferable, because their use improves readability of the code, and makes it easier to maintain.
- ▶ If you want to use a particular structure type throughout a source file, you should use a typedef or a structure declaration with a tag *before main* (this will be necessary in lab 3); this is necessary to give file scope to the type in the typedef, or the struct type.
- ▶ If you want to use a particular structure type in more than one source file, you should put the tag declaration or typedef in *a header file*. You can then #include the header file with the declaration wherever it is needed.



# Structure Members

- ▶ In the examples so far, all structure members have been of simple types.
- ▶ Generally, though, any kind of variable that can be declared outside of a structure can also be used as a structure member.
- ▶ Specifically, all of the following are possible as members (and other derived types also):
  - Arrays
  - Pointers
  - Structures
  - And obviously, primitive data types (char, int, float, etc.) as well
- ▶ Structure members can have identical names to members of a structure of a different type. The way that the members are accessed allows them to be referred to and accessed unambiguously.

# Example of a structure with a structure member

```
typedef struct {  
    float tax_rate;  
    float gross_salary;  
} Tax_info;
```

```
struct Employee {  
    char *first_name;  
    char *last_name;  
    int id_number;  
    Tax_info tax;        /* Another structure of type declared above */  
} employees[50], *emp_ptr;
```

# Direct Member Access

- ▶ The members of a structure are directly accessed with **the dot operator**, which takes two operands:
  - The left operand is a structure variable (either an identifier which names a declared structure variable, or a dereferenced pointer to a structure variable).
  - The right operand is the name of a member of that type of structure variable.

# Examples of Direct Member Access

## ► Consider:

```
struct Employee {  
    char *first_name;  
    char *last_name;  
    int id_number;  
    Tax_info tax;  
} emp1, emp2;
```

To access members for emp1: emp1.first\_name, emp1.last\_name, emp1.id\_number, emp1.tax

Similarly for emp2: emp2.first\_name, emp2.last\_name, emp2.id\_number, emp2.tax

Question: Where are the Employee names stored??? In this structure? Somewhere else?

# Structures as Structure Members

- ▶ For the employee structure above, one of the members, tax, is another structure of type TaxInfo.
- ▶ Suppose it has been declared as follows:

```
typedef struct {  
    float tax_rate;  
    float gross_salary;  
} TaxInfo;
```

- ▶ Of course, this declaration would have to be made prior to the declaration of struct Employee shown on the preceding slide.
- ▶ Now, to access the members of emp1.tax or emp2.tax, we use two dot operators (the dot operator has L-R associativity):

```
emp1.tax.tax_rate  
emp1.tax.gross_salary  
emp2.tax.tax_rate  
emp2.tax.gross_salary
```

# Indirect Member Access

- ▶ How do we access the members of a structure using a pointer to the structure?
- ▶ Suppose we have:  
    struct Employee \*ptr;
- ▶ Now, suppose we pass the structure pointer to a function:  
    void func (struct Employee \*ptr);
- ▶ How can the members of the structure be accessed inside the function?

# Indirect Member Access cont.

- ▶ The function must dereference the pointer, and can use:
  - `(*ptr).first_name`
  - `(*ptr).last_name`
  - etc.
- ▶ The parentheses **MUST** be used, because the dot operator has higher precedence than the dereference operator. Notice that this usage requires the `*` within the parenthesis rather than outside of them.
- ▶ Because this notation is a nuisance (and can be confusing), C also provides another operator which is more convenient, the `->` (arrow) operator (Same precedence as dot operator – both have L-R associativity).
- ▶ The arrow operator can be typed as two characters: hyphen followed by greater than (but the compiler interprets these two characters as a single operator).
- ▶ Instead of `(*ptr).first_name`, we can use `ptr->first_name`
- ▶ Instead of `(*ptr).last_name`, we can use `ptr->last_name`



# Indirect Member Access cont.

- ▶ Note that the left operand of the `->` operator must be *a pointer to a structure*. This operator is not valid with any other type of left operand.
- ▶ Also notice that the left-hand operand of the arrow operator is ALWAYS dereferenced, even though no dereference operator is used; it is *implied by*, (i.e. “built-into”) the arrow operator.

# Indirect Member Access cont.

- ▶ Since the left-hand operand of the arrow operator must be a structure pointer, and since it is always dereferenced, we can convert expressions with dereference and dot to expressions with arrow (and vice-versa) as follows:

left-operand->right-operand    *is equivalent to*    (\*left-operand).right-operand

- ▶ What if we have multiple arrow operators in an expression?

op1->op2->op3    *is equivalent to*    ((\*op1).op2).op3

- ▶ As you can see, writing such expressions with dereference and dot operators is very cumbersome, so the arrow operator is almost always used by experienced C programmers in these cases.

# Let's go back to the example of a structure with a structure member

```
typedef struct {  
    float tax_rate;  
    float gross_salary;  
} Tax_info;
```

```
struct Employee {  
    char *first_name;  
    char *last_name;  
    int id_number;  
    Tax_info tax;      /* Another structure of type declared above */  
} employees[50], *emp_ptr;
```

We can use `emp_ptr->first_name` to reference the address to each employees first name.

How do we reference `gross_salary`?

# Let's go back to the example of a structure with a structure member

```
typedef struct {  
    float tax_rate;  
    float gross_salary;  
} Tax_info;
```

```
struct Employee {  
    char *first_name;  
    char *last_name;  
    int id_number;  
    Tax_info tax;        /* Another structure of type declared above */  
} employees[50], *emp_ptr;
```

We can use `emp_ptr->first_name` to reference the address to each employees first name.

How do we reference `gross_salary`?      `emp_ptr->tax.gross_salary`

# Results of two options

test\_struct1.c:

```
int main()
{
    typedef struct {
        float tax_rate;
        float gross_salary;
    } Tax_info;
    struct Employee {
        char *firstname;
        char *last_name;
        int id_number;
        Tax_info tax;
    } employees[50], *emp_ptr;

    emp_ptr = employees;
    emp_ptr->tax.gross_salary = 1200.00;
    return(0);
}
```

```
$ gcc -ansi -pedantic test_struct1.c -o test_struct1
$
```

test\_struct2.c:

```
int main()
{
    typedef struct {
        float tax_rate;
        float gross_salary;
    } Tax_info;
    struct Employee {
        char *firstname;
        char *last_name;
        int id_number;
        Tax_info tax;
    } employees[50], *emp_ptr;

    emp_ptr = employees;
    emp_ptr->tax->gross_salary = 1200.00;
    return(0);
}
```

```
$ gcc -ansi -pedantic test_struct2.c -o test_struct2
test_struct2.c: In function `main':
test_struct2.c:24: error: invalid type argument of `->' (have `Tax_info')
$
```

In this instance, tax isn't a pointer, it's a member of a structure.

# Self-Referential Structures

- ▶ Is it legal for a structure to contain a member that is the same type as the structure?
- ▶ Suppose we declare a structure such as the following:

```
struct Self_Ref1 {  
    int a;  
    struct Self_Ref1 b;  
    int c;  
};
```

## Self-Referential Structures cont.

- ▶ This type of self-reference is not legal, because the member `b` is another complete structure, which will contain its own member `b`. This second member `b` is yet another complete structure, and contains its own member `b`, and so forth.
- ▶ The problem with this declaration is that it is **INFINITELY RECURSIVE**, so the compiler would have to attempt to allocate an infinite amount of storage.
- ▶ Let's compare this with the self-referential declaration on the following slide.




## Self-Referential Structure Definitions cont.

- ▶ Following is another self-referential structure. Can you distinguish it from the previous one?

```
struct Self_Ref2 {  
    int a;  
    struct Self_Ref2 *b;  
    int c;  
};
```

- ▶ Is this kind of self-referential structure legal?

## Self-Referential Structure Definitions cont.

- ▶ The kind of self-referential structure on the preceding slide *is* legal, and is used to build linked data structures, such as linked lists, and trees.
  - ▶ The difference from the previous recursive structure which does not terminate is that in this case, the structure does not contain a member that is another structure of the same type, but rather, *a pointer* to a structure of the same type.
  - ▶ The compiler of course can allocate space for a pointer to a structure, which requires a finite amount of space.
  - ▶ We will see how to build a linked list with such a self-referential structure soon.
- 

# Initializing Structures

- ▶ Structures can be initialized in much the same way as arrays. For example:

```
struct Initialization_example {  
    int a;  
    short b[10];  
    Simple c;          /* Defined with typedef above */  
} y = {  
    10,  
    {1, 2, 3, 4, 5},    /* initialization of array member */  
    {25, 1.9}           /* initialization of struct member */  
};
```

- ▶ The initial values for the members must be in the order given in the member list, and they are separated by commas.
- ▶ Missing values (at the end of the list of values) cause the remaining members to get default initialization.

# Structures and Assignment

- ▶ Let's modify the previous example slightly, by declaring an additional structure variable x, without initialization:

```
struct Init_example {  
    int a;  
    short b[10];  
    Simple c;  
} x, y = {  
    10,  
    {1, 2, 3, 4, 5},  
    {25, 1.9}  
};
```

- ▶ Now, because x and y have been declared to be of the same type, we can assign y to x:  

```
x = y;
```
- ▶ This will assign the value of each member of y to the corresponding member of x.
- ▶ Of course, we can also initialize, or assign to, the members of a structure one by one, with individual assignment statements, as shown on the following slide.

# Structures and Assignment cont.

- ▶ Assignment to individual members, given the following declarations:

```
struct Init_example {  
    int a;  
    short b[10];  
    Simple c;  
} x, y = {  
    10,  
    {1, 2, 3, 4, 5},  
    {25, 1.9}  
};  
x.a = 10;  
x.b = {1,2,3,4,5};  
x.c = {25, 1.9};
```

# Structures as Function Arguments

- ▶ It is legal to pass a structure variable as an argument to a function, but it is rarely the best option.
- ▶ If the structure is large, because parameters are passed by value in C, all of the values of the members in the structure will have to be copied and pushed onto the stack when the function is called.
- ▶ On the other hand, if we pass a pointer to the structure as a parameter, then only the value of the pointer is passed, and pushed onto the stack, rather than copies of all the values in the structure.
- ▶ The price for this is that dereferencing must be used inside of the function in order to access the structure members (a small price for the storage space saved, and the speed up in execution time).
- ▶ Therefore, *only structures which are not much larger than a pointer* should be passed directly.

# Remember keyword const

- ▶ As we have seen before, C has the keyword `const`, which can be used to declare constants, for example:

```
int const MAX_LENGTH = 1000;  
const int MAX_LENGTH = 1000;
```

- ▶ This keyword can also be used when passing function arguments with pointers, to prevent modification of the value(s) to which the pointer points:

```
int func1(int const *ptr) {  
    . . . . . /* func1 will not be able to change the values  
               in the array or structure */  
}
```

- ▶ Unless the function needs to change values using the pointer, **we should declare the parameter with *const***, to indicate that the value(s) pointed to by the parameter will not be changed, and also to restrict the interaction between the function and the calling environment. This also aids debugging.
- ▶ This can be done with structures also: `int func2 (const struct Employee *structPtr);`



# typedef is different

```
typedef struct
{
    int lb_thrust, payload;
} Booster; /* Booster is the name
of the new type */
```

```
typedef struct Chip /* creates a
new type 'struct Chip' we will
need.. */
{
    char codename[21];
    int MHz;
    /* to self-reference here: */
    struct Chip *predecessor;
} Chip; /* this is another
new type name */
```

```
int main()
{
    /* struct Booster saturn5;
       no such type exists */
    /* the only way to declare one
of these is: */
    Booster falcon9;

    /* these two are actually the same
type since one is a typedef of the
other */
    struct Chip i386;
    Chip PentiumPro;
}
```