C Basics 12: Structs (Structures)

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory

C Basics 12: Structs (Structures)

11. Multidimensional and String Arrays
12. **Structs (Structures)**
    - **structs vs. classes**
    - **defining structs**
    - **accessing members/fields**
    - **struct initialization**
    - **typedef**
    - **unions**
    - **structs, unions, and arrays**
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Structs/Structures vs. Classes

The main **heterogeneous composite data type** in C is the **struct** (short for **structure**).

Structs are similar to OOP **classes**:

- define a new **type**
- composed of multiple **members** (also often called **fields**)
- the component *members* can be of different types

Structs differ from classes:

- no **inheritance** (subclass/superclass structure)
- no **encapsulation** of **methods**
- no **information hiding** capabilities (e.g., `private`)

Defining a Struct

The syntax for defining a struct is:

```
struct [struct_name] {
  member_definition1;
  member_definition2;
  ...
  member_definitionn;
} [variable(s)];
```

The elements inside `[ ]`'s are *optional*.

Each `member_definition` is similar to a standard C variable definition, with a *type* and a *name* (*identifier*), which define the type and name of the next struct member:

```
int count;
char name[20];
```

Defining a Struct (contd.)

For example, a definition of a **named struct** to hold addresses:

```
struct address {
  char street[50];
  char city[25];
  char state[2];
  int zip;
};
```

An `address` struct variable `user_addr` can then be declared as:

```
struct address user_addr;
```

Note that the name of the resulting type is actually "`struct address`" rather than just "`address`."

Defining a Struct (contd.)

Struct variable(s) can be declared as part of a struct definition:

```
struct address {
  char street[50];
  char city[25];
  char state[2];
  int zip;
} user_addr1, user_addr2;
```

These variables can still be "initialized" with **initializer notation**, but a *cast* is required (technically we are copying a **compound literal** rather than initializing a struct):

```
user_addr1 = (struct address){"Lincoln Ave",...};
```

Instead, we could set each field separately:

```
user_addr1.street = "Lincoln Ave";
...
```

Unnamed Structs

As shown earlier, the `struct_name` in a definition is *optional*.

A struct without a name is called an **unnamed struct**.

An *unnamed struct* along with variable definitions can be used if the struct type is not required again in the program:

```
struct {
  char street[50];
  char city[25];
  char state[2];
  int zip;
} user_addr1, user_addr2;
```

Nested Structs

The type of a struct member can be any type, including another struct.

Structs with member(s) that are structs, are called **nested structs**:

```
struct address {
  ...
};

struct person_record {
  char name[100];
  struct address mailing_address;
  int age;
};
```

It is important to understand the syntax for *initialization* and *member access* with nested structs (see below).

Nested Structs (contd.)

One limitation on members' types is that a struct *cannot* contain a member of the *struct's own type* (i.e., it cannot be *"recursive"*).

It can, however, contain a member that is of type *pointer* to the struct's own type (so struct objects can be linked together):

```
struct address {
  char street[50];
  char city[25];
  char state[2];
  int zip;
  struct address *next_addr;
};
```

Nested Structs (contd.)

Unnamed structs can be used in nested structs but add little:

```
struct person_record {
  char name[100];
  struct {
    char street[50];
    char city[25];
    char state[2];
    int zip;
  };
  int age;
};
```

The members of the nested struct (`street`, etc.) are treated just like non-nested members (e.g., `name`) for access.

Dot/Member Operator

Structure members/fields are accessed using the **dot operator** (**member operator**): `user_addr.zip`

This is similar to Java/C++ notation for accessing a *class member*.

Example:

```
...
struct address bob_addr;
bob_addr.street = "101 Main St.";
bob_addr.city = "Carbondale";
bob_addr.state = "IL";
bob_addr.zip = 62901;
...
printf("City: %2s", bob_addr.city);
```

Dot/Member Operator (contd.)

Precedence/associativity allows simple *chaining* of dot operators with *nested structs*:

```
...
struct person_record {
  char name[100];
  struct address mailing_address;
  int age;
};

struct person_record bob_rec;
...
bob_rec.name = "Bob Smith";
bob_rec.mailing_address.zip = 62901;
...
int bob_zip = bob_rec.mailing_address.zip;
```

Struct Pointers

If a *function parameter* is a struct type, C's **pass-by-value** approach will result in actually *copying the struct* argument.

This could cause function calls to become very costly with large struct arguments.

For this reason, when struct parameters are required, it is *most common to use parameters that are struct pointers*.

For example:

```
int store_addr(const char *path, struct address *addrptr);
```

All that gets copied now is the *pointer* (memory address) of the struct rather than the entire struct.

Arrow Operator

It is common to use struct pointers, but because of precedence, accessing a member of a struct pointer requires using parens:

```
(*addrptr).zip
```

The **arrow operator** simplifies accessing fields of struct pointers:

```
addrptr->zip
```

(note that the "arrow operator" is two characters: - + >)

Example of function using arrow operator:

```
void print_address(struct address *addrptr)
{
  printf("%s\n%s,%s %5d\n", addrptr->street, addrptr->city,
                            addrptr->state, addrptr->zip);
}
```

Arrow Operator (contd.)

The arrow operator can also be used on the LHS of *assignments*:

```
void change_zip(struct address *addrptr, int newzip)
{
  addrptr->zip = newzip;
}
```

Note that because a struct *pointer* is being passed, the function can modify the values of the struct argument's members!

Struct Initialization

Structs can be initialized using *brace notation* similar to that used for array initialization:

```
struct address bob_addr =
  {"101 Main St.", "Carbondale", "IL", 62901 };
```

Members are filled in sequentially, so a nested struct could be initialized in two ways:

```
struct person_record bob =
  {"Bob Smith", {"101 Main St.", "Carbondale", "IL", 62901}, 52};
```

Or simply:

```
struct person_record bob =
  {"Bob Smith", "101 Main St.", "Carbondale", "IL", 62901, 52};
```

Struct Initialization (contd.)

If struct variables are *not initialized*, their member values are indeterminate (i.e., likely to be random values).

However, if an initializer is used, but there are fewer elements in the *brace-enclosed list* than members (or submembers), the remaining members will be *initialized implicitly*.

Implicitly initialized members end up with the same values as objects that have **static storage duration**.

Struct Initialization: Copying

A struct can also be initialized as a *copy* of another struct:

```
struct address fred_addr = bob_addr;
```

Note: this does what is called a **shallow copy** only!

A shallow copy means that the "top-level" members get copied; if any members are *pointers*—to separate objects—those objects do *not* get copied (recursively).

This means that when a struct contains pointers, copying it via such an assignment will result in two structs containing copies of the *same* pointer/address—i.e, sharing (sub)object(s).

So be very careful when copying structs via assignments!

Struct Initialization: Designators

With C99 and on, partial or out-of-order initialization can be done using **designators** of the form **.membername=**:

```
struct address bob_addr = {.zip = 62901, .city = "Carbondale"}
```

When using *designators*, unspecified members are automatically initialized with the same values as objects that have **static storage duration** (zero/`NULL`).

Initializers can mix designators with plain brace lists of members:

- each brace-enclosed initializer list has an associated member
- *w/o designator*, members of the *current* object are initialized, in struct declaration order
- *w/designator*, members of the *designated* member are initialized, in struct declaration order
- initialization then continues forward in struct declaration order, beginning with the next member

Struct Initialization (contd.)

With the addition of *designators*, there can be many alternative *brace notations* for initializing a struct.

Consider the nested structs used with POSIX timers:

```
struct itimerspec {
  struct timespec it_interval;  /* Timer interval */
  struct timespec it_value;     /* Initial expiration */
};

struct timespec {
  time_t tv_sec;                /* Seconds */
  long   tv_nsec;               /* Nanoseconds */
};
```

Struct Initialization (contd.)

An `itimerspec` struct could be initialized as:

```
struct itimerspec tspec = {...};
```

where many variations are possible for the brace list, e.g.:

- `{5,0,10,0};`
- `{{5,0},{10,0}};`
- `{.it_interval={5,0},.it_value={10,0}};`
- `{.it_value={10,0},.it_interval={5,0}};`
- `{.it_interval={5},.it_value={10}};`
- `{.it_interval=5,.it_value=10};`
- `{.it_interval={.tv_sec=5,.tv_nsec=0},.it_value={.tv_sec=10,.tv_nsec=0}};`
- `{.it_value={.tv_nsec=0,.tv_sec=10},.it_interval={.tv_nsec=0,.tv_sec=5}};`
- `{.it_interval={.tv_sec=5},.it_value={.tv_sec=10}};`
-
```
{.it_value.tv_sec=10,.it_value.tv_nsec=0,
 .it_interval.tv_sec=0,.it_interval.tv_nsec=0};
```

Structs vs. Arrays

Because C effectively represents arrays as pointers but does not do this for structures, there are some confusing differences in using the two different composite data types.

One difference is that you cannot use the *equality operator* (==) to compare structures, while you can use it with arrays.

Of course with arrays, equality means *same memory block*; there is no builtin operator to compare the contents of two arrays to see if they are equal.

If you want to be able to compare structure variables to see if their members have the same values, you must write your own equality function.

Structs vs. Arrays (contd.)

A second key issue is that using a struct in an *assignment statement* will cause the same copying as we discussed above in relation to function parameters:

```
...
struct address bob_addr, tom_addr;
...
tom_addr = bob_addr;
```

This causes a *copy* to be made of the `bob_addr` struct and stored in `tom_addr`.

Sometimes this is what you want: `tom_addr` is separate from `bob_addr` but has the same member values (initially).

Structs vs. Arrays (contd.)

When using a struct pointer, **dynamic memory** may be used to allocate the struct's storage for the struct pointer variable:

```
...
struct address *bob_addrp;
bob_addrp = malloc(sizeof(struct address));
bob_addrp->street = "101 Main St.";
```

We can create struct pointer **alias** as follows:

```
...
struct address *tom_addrp;
tom_addrp = bob_addrp;  //Struct is not copied, only pointer
```

Now `bob_addrp` and `tom_addrp` point to the same single struct object (allocated from the heap).

typedef

C provides limited methods for users to *define new* **data types**.

`typedef` allows users to create new **named types** from other C types.

The syntax is basically:

`typedef` *C_type_spec new_type_identifier*;

It has two primary uses:

- to provide a type name that reflects intended usage
- to allow simpler specification of composite types

`typedef` is how POSIX types like `size_t` are defined:

```
typedef unsigned long int size_t;
```

(defines type `size_t` to be equivalent to `unsigned long int`)

typedef (contd.)

Example using `typedef` to simplifying using array as parameter:

```
typdedef int 2d_int_array[10][10];

void 2dfun(2d_int_array 2d_arr)
{...}
```

`2d_int_array` is defined as a type equivalent to a 10 x 10 `int` array.

Without the `typedef`, the function definition would have to be:

```
void 2dfun(int 2d_arr[10][10])
{...}
```

typedef (contd.)

`typedef`'s are frequently used to simplify struct usage:

```
typedef struct point{
  float x;
  float y;
} line[2];
```

This defines `line` as a type that is equivalent to an array of two `struct point` objects.

We can now define `line` parameters:

```
int linelen(line l)
{...}
```

instead of the less clear:

```
int linelen(struct point l[2])
{...}
```

typedef (contd.)

Example program showing use of `struct` and `typedef`:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct point{
  double x;
  double y;
} line[2];

double linelen(line l)
{
  return sqrt(pow(l[0].x - l[1].x, 2.0) + pow(l[0].y - l[1].y, 2.0));
}

int main(int argc, char *argv[])
{
  line l1 = {{0.0, 0.0}, {1.0, 1.0}};
  struct point p1 = {0.5, 1.5};
  struct point p2 = {3.25, 4.0};
  line l2 = {p1, p2};

  printf("Line #1 length: %f\n", linelen(l1));
  printf("Line #2 length: %f\n", linelen(l2));
  exit(EXIT_SUCCESS);
}
```

More on typedefs with Structs

Another common use of `typedef`'s with structs is eliminating the need to include the keyword `struct` with struct types.

For example, we can define `struct address` with a `typedef` as follows:

```
typedef struct address {
  char street[50];
  ...
} address_t;
```

This defines type `address_t` as equivalent to type `struct address`.

This allows us to now do:

```
address_t user_addr;
```

instead of:

```
struct address user_addr;
```

More on typedefs (contd.)

In fact, we can now use either "`address_t`" or "`struct address`".

By using an **unnamed struct** with a `typedef`, you can end up with a single type, e.g., `address_t`:

```
typedef struct {
  char street[50];
  ...
} address_t;
```

Using `typedef`'s with structs is common, but it is by no means universally accepted as good style.

Some people believe you should always include the "`struct`" part of a struct type declaration, to make the struct aspect clear.

Unions

A **union** is a C data type mechanism that allows a *single memory address* to be accessed as different types, using different names.

For example, a type that can store a `signed` or `unsigned int`:

```
union soruint {
  signed int sval;
  unsigned int uval;
} var;
```

We can now access `soruint_var` as a `signed int` by doing:

```
var.sval
```

or as an `unsigned int` by doing:

```
var.uval
```

Note type of this union is: `union soruint`

Unions (contd.)

Unions can be *initialized* using *brace notation* similar to that for structs (and arrays).

Designators can also be used with unions (C99 and on).

However, there are some key differences between initializers for unions vs. for structs:

- a union's brace initializer expression must contain just a *single* expression
- without a designator, only the *first element* of the union can be initialized

Thus, brace list initializers are not terribly useful with unions (can simply do assignment to desired member).

Unions vs. Structs

Unions appear similar to structs:

- same syntax for defining a union as for a struct
- members/fields accessed with *dot and arrow operators*
- define new type: `union union_name`

Unions and structs are significantly different, however:

- *all struct members* will be valid for any struct object vs. *only one union member* will be valid for any union object at each point in time (same memory for all)
- the *size* of a struct object is the *sum* of the sizes of all its members vs. size of union object is the size of its largest member

Unions and Structs

Years ago, unions were often used to save address space in processes, by overloading the same memory for different uses in different parts of a program.

These days, the most common use for unions is in conjuction with structs: union(s) within a struct, or structs within a union.

Using the two together can provide *data type flexibility* that C otherwise lacks due to its lack of **type inheritance** (i.e., subclasses and superclasses).

For example, suppose a student record type (struct) must be able to accomodate one of any of several different ID types.

This could be done using multiple ID members of different types, but would be wasteful since only one will be in use at any time.

Unions and Structs (contd.)

Using a union nested within the struct avoids this waste:

```
struct student_record {
  struct name fullname;
  short id_type;
  union {
    drivlic_t dlic;
    socsec_t ss;
    visa_t visa; }
  ...
};
```

We can use this struct as so:

```
struct student_record newrecord;
...
newrecord.id_type = 1;
newrecord.dlic = "C61...";
```

Note the ability to use the names of the union's members just as if they were struct members.

Unions and Structs (contd.)

An example from the C standard, using structs inside a union:

```
union {
  struct {
    int alltypes; } n;
  struct {
    int type;
    int intnode; } ni;
  struct {
    int type;
    double doublenode; } nf;
} u;

u.nf.type = 1;
u.nf.doublenode = 3.14;
...
if (u.n.alltypes == 1)
  if (sin(u.nf.doublenode) == 0.0)
    ...
```

Note ability to access `int` *"common initial sequence"* in every struct, using any of union's member names.

Structs, Unions, and Arrays

Structs often contain members that are *structs*, *unions*, and/or *arrays*.

Since these types can all be initialized using *brace notation*, nested brace-enclosed initializer lists are quite common.

Consider this struct with embedded arrays, struct, and union:

```
struct test {
  int a;
  int b[2];
  union {
    int c;
    char d;
  };
  struct {
    int e;
    char f[5];
  } g;
  char h[3];
};
```

Structs, Unions, and Arrays (contd.)

Here are the results of various initializations:

- `struct test var = {'A'};`
  `a:65 b[0]:0 b[1]:0 c:0 d: g.e:0 g.f: h:`
- `struct test var = {'A','B','C','D','E','F','G'};`
  `a:65 b[0]:66 b[1]:67 c:68 d:D g.e:69 g.f:FG h:`
- `struct test var = {'A',{'B'},'C','D','E','F','G'};`
  `a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EFG h:`
- `struct test var = {'A','B',.d='C','D','E','F','G'};`
  `a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EFG h:`
- `struct test var = {'A',{'B'},'C','D',{'E','F'},'G'}};`
  `a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:G`
- `struct test var = {'A',{[1]='B'},'C','D',{'E','F'},'G'};`
  `a:65 b[0]:0 b[1]:66 c:67 d:C g.e:68 g.f:EF h:G`
- `struct test var = {'A',{'B'},{'C'},{'D','E','F'},{'G','H','I'}};`
  `a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:GHI`
- `struct test var = {'A',{'B'},{'C'},{'D',{'E','F'}},{'G','H','I'}};`
  `a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:GHI`

Note: `'A'` has decimal value 65, etc.

Details from the Standards

Key points from the C99/C11 standards:

- "a *structure* is a type consisting of a *sequence of members*, whose *storage is allocated in an ordered sequence*"
- "a *union* is a type consisting of a sequence of members whose storage *overlap*...size of a union is sufficient to contain the largest of its members"
- "Each non-bit-field member of a structure or union object is *aligned* in an *implementation-defined manner* appropriate to its type."
- "Within a structure object, the non-bit-field members and [bit-field objects] have *addresses that increase in the order in which they are declared*."
- "There may be *unnamed padding* within a structure object, but *not at its beginning*, [and]...at the *end* of a structure or union."
- "A *pointer to a structure object*...points to its initial member (or if that member is a bit-field, then to [its] unit...)"
- "A *pointer to a union object*...points to each of its members (or if a member is a bit-field, then to [its] unit...)"
- "...a structure *shall not contain an instance of itself*, but may contain a *pointer to an instance of itself*..."