CS265
Advanced Programming Techniques

C Structures, Unions, Enumerations

C Structures

A struct (structure) is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling

- A structure is a collection of related data
- A struct is an aggregate data type, allowing related data elements to be accessed and assigned as a unit.
- It defines a user-defined data type
- It is similar to a table in a database
- It is a foundational component that allows you to build more complex data structures, like linked lists, stacks, trees, etc.

Defining Structure Variables

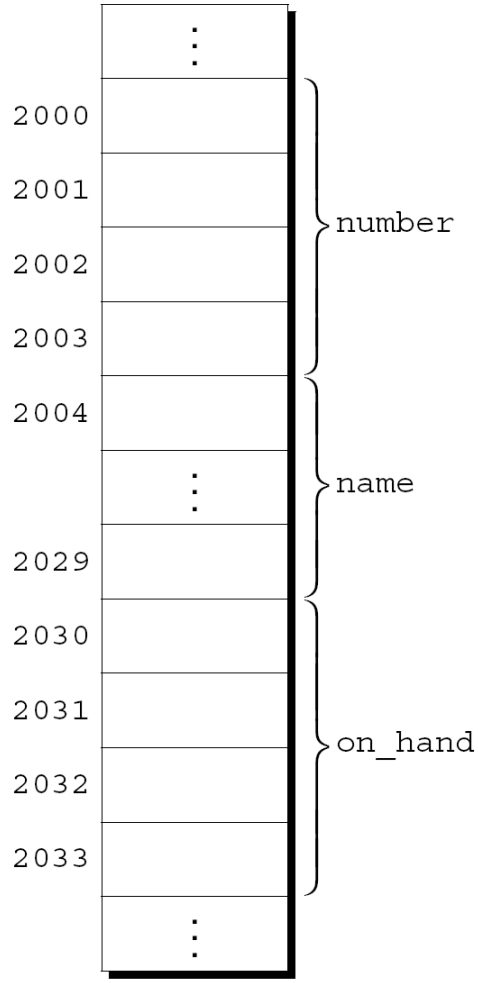
- You can define a variable and its structure at the same time
- Here is a declaration of two variables `part1` and `part2` as structures:

```
struct {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
} part1, part2;
```

How are structures stored in memory

- The members of a structure are stored in memory in the order in which they're declared

- Appearance of `part1`
- Assumptions:
  - `part1` is located at address 2000
  - Integers occupy four bytes
  - `NAME_LEN` has the value 25
  - There are no gaps between the members
  - There is no null character at the end
  - There is no null pointer at the end



Structures vs Arrays

Arrays

- Collections of related data
- Stored in memory consecutively
- All elements have the same data type
- Elements do not have names
- Access is via the index/position

```
int a[100];
```

Structures

- Collections of related data
- Stored in memory consecutively
- Elements need not have the same data type
- Elements have names
- Access is via the name

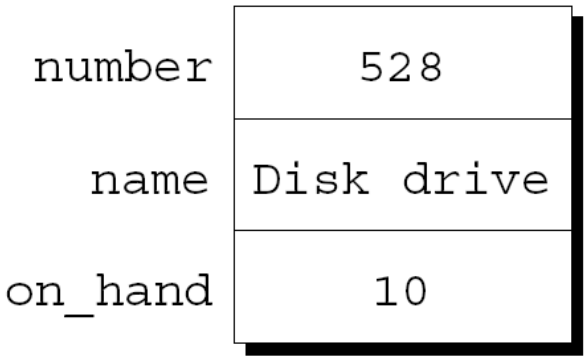
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
} part1 = {528, "Disk drive", 10};
```

- Appearance of `part1` after initialization:



Some rules about initializing structures

- Similar to how we initialize arrays

- If fewer members are initialized, the remaining members get 0

```
= {528, "Disk drive"};
```

- We can use designated initializers where each value is labeled with the member that it initializes (order does not matter)

```
= {.number = 528, .name = "Disk drive", .on_hand = 10}
```

- We can mix and match

```
= {.number = 528, "Disk drive", .on_hand = 10}
```

Operators on Structures – The . operator

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1;
```

- To access the members we use the . operator

```
part1.number
part1.name
part1.on_hand
```

Operators on Structures - Assignment

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

- Assignment copies all elements, one by one, including array elements

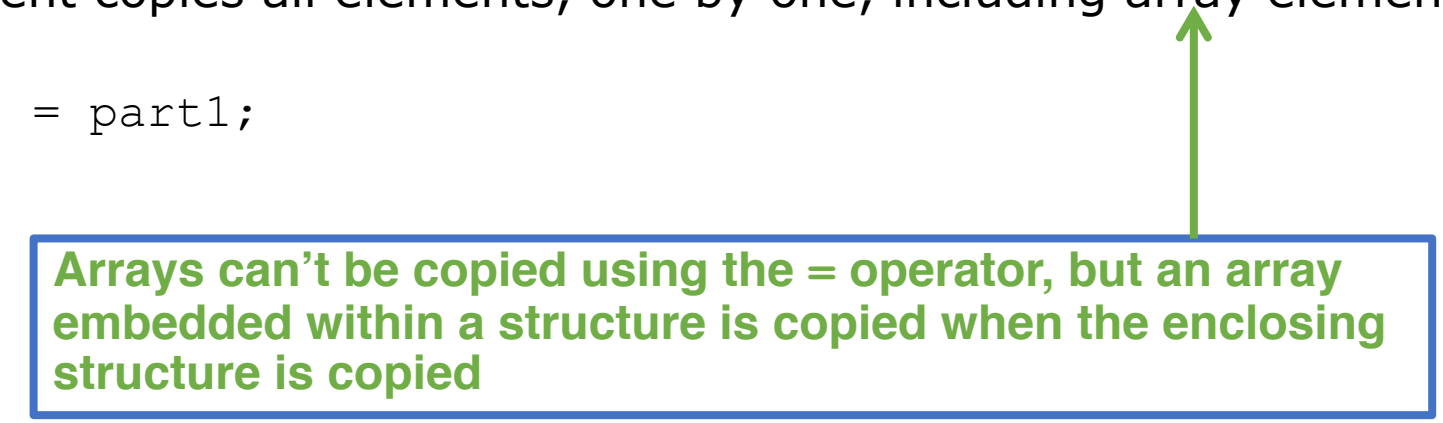
```
part2 = part1;
```

Operators on Structures - Assignment

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

- Assignment copies all elements, one by one, including array elements

```
part2 = part1;
```



Operators on Structures – NO COMPARISONS

- Only assignment can be used for structures

```
part2 = part1;          /* LEGAL */
```

- Comparison cannot be used with structures

```
if (part2 == part1) ..   /* ILLEGAL */
if (part2 != part1) ..   /* ILLEGAL */
```

Operations on Structures – Reading and Writing

- Read using `scanf`

```
scanf("%d", &part1.on_hand);
```

- Write using `printf`

```
printf("Quantity on hand: %d\n", part1.on_hand);
```

Defining Structures

- Using a structure tag named `part`

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Using a typedef to define a new user-defined type named `Part`

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

Defining Structures Using a Structure Tag

- Using a structure tag named `part`

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Declare variables

```
struct part part1, part2;
```

data type **variables**

Defining Structures Using `typedef`

- we can use `typedef` to define a structure

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- Declare variables

```
Part part1, part2;
```

data type **variables**

Structures as Arguments

- Functions may have structures as arguments

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- Example

```
print_part(part1);
```

Is it pass by value or pass by reference?

Structures as Return Values

- A function can return a structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- Example

```
part1 = build_part(528, "Disk drive", 10);
```

Compound Literals

- A compound literal creates a structure on the fly

```
(struct part) {528, "Disk drive", 10}
```

- You can pass it to a function:

```
print_part((struct part) {528, "Disk drive", 10});
```

- You can assign it to a variable:

```
part1 = (struct part) {528, "Disk drive", 10};
```

Structures and Arrays

- Arrays may have structures as elements

```
struct part inventory[100];
```

- Access

```
print_part(inventory[i]);
inventory[i].number = 883;
inventory[i].name[0] = '\0';
```

Structures and Arrays

- Structures may contain arrays as members

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1;
```

Structures can be Nested

- Nesting one structure inside another is often useful
- Suppose that `person_name` is the following structure:

```
struct person_name {
        char first[FIRST_NAME_LEN+1];
        char middle_initial;
        char last[LAST_NAME_LEN+1];
};
```

- We can define a student as

```
struct student {
        struct person_name name;
        int id, age;
} student1;
```

- Accessing `student1`'s first name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Nested Structures

- Makes it easier to pass as arguments to functions

```
display_name(student1.name);
```

- Makes it easier to copy too

```
struct person_name new_name;
…
student1.name = new_name;
```

Example using `struct`

```
#include <stdio.h>
#include <string.h>

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

void print_book(struct Books book) {
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}

int main() {
   struct Books book1;
   struct Books book2;

   strcpy(book1.title, "The C Programming Language");
   strcpy(book1.author, "Kernighan and Ritchie");
   strcpy(book1.subject, "C Programming");
   book1.book_id = 1;

   strcpy(book2.title, "C Programming: A Modern Approach");
   strcpy(book2.author, "K.N. King");
   strcpy(book2.subject, "C Programming");
   book2.book_id = 2;

   print_book(book1);
   printf("\n");
   print_book(book2);
}
```

Example using `typedef`

```
#include <stdio.h>
#include <string.h>

typedef struct {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Books;

void print_book(Books book) {
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}


int main() {
   Books book1;
   Books book2;
   strcpy(book1.title, "The C Programming Language");
   strcpy(book1.author, "Kernighan and Ritchie");
   strcpy(book1.subject, "C Programming");
   book1.book_id = 1;

   strcpy(book2.title, "C Programming: A Modern Approach");
   strcpy(book2.author, "K.N. King");
   strcpy(book2.subject, "C Programming");
   book2.book_id = 2;

   print_book(book1);
   printf("\n");
   print_book(book2);
}
```

Pointers to Structures

```
student s;

student p = &s;
```

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

```
p->x
```

- is equivalent to the expression

```
(*p).x
```

Unions

- A union, like a structure, consists of one or more members, possibly of different types
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space
- Assigning a new value to one member alters the values of the other members as well

Unions

- An example of a union variable:

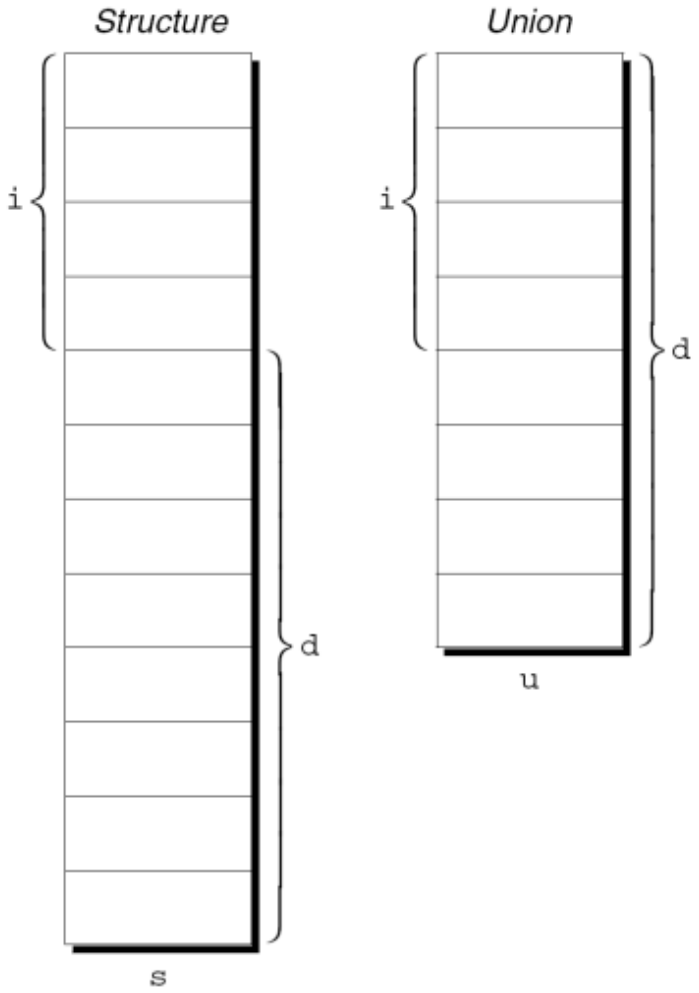
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```

Unions

- The structure `s` and the union `u` differ in just one way.
- The members of `s` are stored at different addresses in memory.
- The members of `u` are stored at the same address.



Enumerations

- Enumerated data types are possible with the `enum` keyword. They are freely interconvertible with integers.
- Examples: deck of cards

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- Can be defined with a structure tag

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};

enum suit s1, s2;
```

- Or with a typedef

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

Enumerations as integers

- Behind the scenes, C treats enumeration variables and constants as integers
- By default, the compiler assigns the integers 0, 1, 2, … to the constants in a particular enumeration

```
CLUBS      is 0
DIAMONDS   is 1
HEARTS     is 2
SPADES     is 3
```

- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

enum

```
#include <stdio.h>

enum day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY};


int main()

{

    enum day d = TUESDAY;

    printf("The day number stored in d is %d", d);

    return 0;

}
```

- Output:

  The day number stored in d is 2

Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages

Resources

- These notes
- *C Programming: A modern Approach* by K.N. King, 2008
- Chapter 16