Lecture 20: Structures, Unions, and Enumerations

By Ahmed E. Khaled
• Recap to the previous lecture:
  
  – String literals vs. String variables
  – Different string functions
    strcpy, strcat, strcmp and strlen
Structure Variable

• The only data structures covered so far is array.

• Array has two properties:
  1) all elements of the array have same type
  2) we use integer subscript to select an array element by index
Structure Variables

• A structure is a collection of values (members), such members may have different types.

• Each member of a structure has a name (not index as an array element).

• In some languages, structures are called *records*, and members are known as *fields*. 
Declaring Structure Variables

• A structure is a logical choice for storing a collection of related data items.
• A declaration of two structure variables that store information about parts in a warehouse:

```c
struct {
    int number;          // the part number
    char name[NAME_LEN+1];  // name of the part
    int on_hand;        // quantity on hand
} part1, part2;
```
• 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.
Each structure represents a new scope, thus names declared in that scope won’t conflict with other names in a program.

In C terminology, each structure has a separate *name space* for its members. For example, the following declarations can appear in the same program:

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

struct {
    char name[NAME_LEN+1];
    int number;
    char sex;
} employee1, employee2;
```
Initialize Structure Variables

- A structure declaration may include an initializer:
  ```c
  struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
  } part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
  ```

- Appearance of `part1` after initialization:
  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>528</td>
</tr>
<tr>
<td>name</td>
<td>Disk drive</td>
</tr>
<tr>
<td>on_hand</td>
<td>10</td>
</tr>
</tbody>
</table>
Operations on Structures

• To access a member within a structure, we write the name of the structure first, then a *period*, then the name of the member.

• Statements that display the values of *part1’s* members:

  ```c
  printf("Part number: %d\n", part1.number);
  printf("Part name: %s\n", part1.name);
  printf("Quantity on hand: %d\n", part1.on_hand);
  ```

• They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

  ```c
  part1.number = 258;  //changes part1's part number
  part1.on_hand++;     //increments part1's quantity
  ```
• The period used to access a structure member is actually a C operator.

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

The . operator takes precedence over the & operator, so & computes the address of `part1.on_hand`.
• The other major structure operation is *assignment*:

```c
part2 = part1;
```

• The effect of this statement is to copy

```c
part1.number into part2.number, part1.name into part2.name, and so on.
```

• The `=` operator can be used only with structures of *compatible* types.

• Two structures declared at the same time (as *part1* and *part2* were) are compatible.
Arrays can’t be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.

Some programmers exploit this property by creating “dummy” structures to enclose arrays that will be copied later:

```c
struct { int a[10]; } a1, a2;
a1 = a2;
/* legal, since a1 and a2 are structures */
```
Structure Types

• Suppose that a program needs to declare several structure variables with identical members.

• We need a name that represents a type of structure, not a particular structure variable.

• Way to name a structure is to declare a “structure tag”.

Declaring a Structure Tag

• A *structure tag* is a name used to identify a particular kind of structure.

• The declaration of a structure tag named `part`:

```c
struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
};
```
• The *part* tag can be used to *declare variables*:

```c
struct part part1;
struct part part2;
```

• We can’t drop the word *struct*:

```c
part part1, part2;  /*** WRONG ***/**
part isn’t a type name; without the word struct, it is *meaningless*.
```

• Since structure tags aren’t recognized unless preceded by the word *struct*, they don’t conflict with other names used in a program.
• All structures declared to have type `struct part` are compatible with one another:

```c
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
    /* legal; both parts have the same type */
```
#include <stdio.h>
#include <string.h>

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

void main() {
    struct Books Book1;    /* Declare Book1 of type Book */

    strcpy( Book1.title, "C Programming" );
    strcpy( Book1.author, "Nuha Ali" );
    Book1.book_id = 6495407;

    printf( "Book 1 title : %s\n", Book1.title);
    printf( "Book 1 author : %s\n", Book1.author);
    printf( "Book 1 book_id : %d\n", Book1.book_id);
}