Structures, Unions, and Enumerations (1)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Structure Variables
- Declaring Strucutre Variables
- Initializing Strucutre Variables
- Operations on Structures
- Strucutre Type
- Structures as Arguments and Return Values

Structure Variables

- The only data structure we've covered is the array.
- However, the array can only store the elements with same type.
- ***structure*** is another more flexible data structure!
- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.

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:

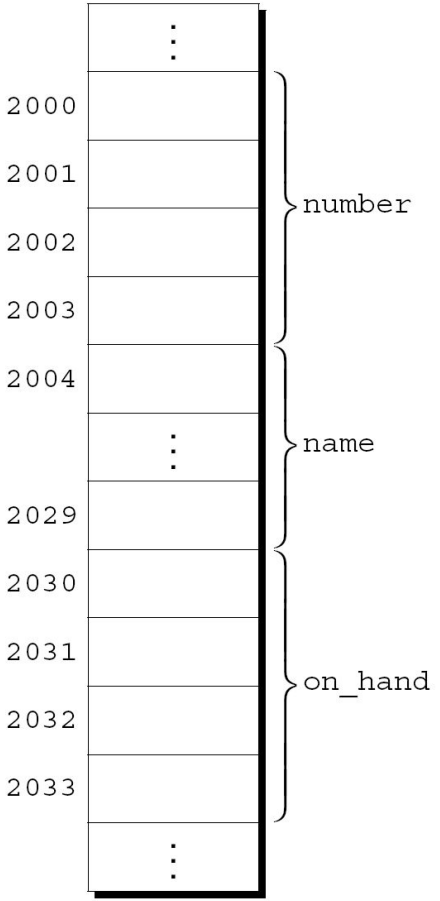
```
struct {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
} part1, part2;
```

3 members of the `struct`

variables of that type

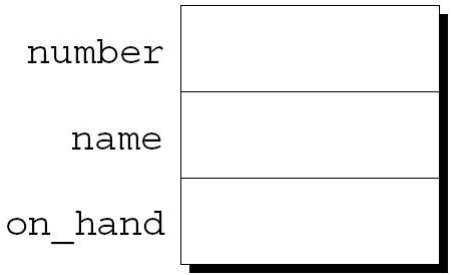
Declaring Structure Variables

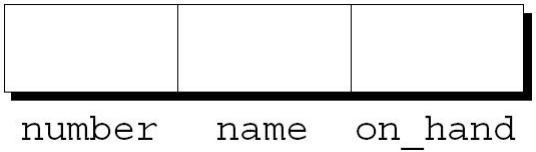
- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1`
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.



Declaring Structure Variables

- Abstract representations of a structure
- Member values will go in the boxes later.





Declaring Structure Variables

- Each structure represents a new scope.
- Any names declared in that scope won't conflict with other names in a program. For example, …
- The number and name members in `part1` doesn't conflict with those in `employee1`

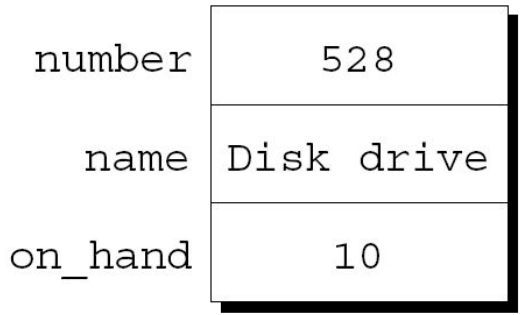
```
struct {
     int number;
     char name[NAME_LEN+1];
     int on_hand;
} part1, part2;

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

Initializing Structure Variables

- A structure declaration may include an initializer
  - a list of values enclosed in braces
  - the values in the initializer must appear in the same order as the members of the structure
- Appearance of `part1` after initialization

```
struct {
     int number;
     char name[NAME_LEN+1];
     int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```



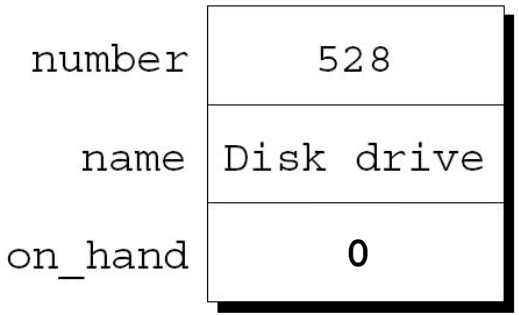
Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- An initializer can have fewer members than the structure it's initializing.
- Any "leftover" members are given 0 as their initial value.

```
int a[3] = {1, 2};
```

1	2	**0**

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



Designated Initializers

- Designated initializers can be used with structures.
- In a designated initializer, each value would be labeled by the name of the member that it initializes
- The combination of the period and the member name (`.number`) is called a **designator**.

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

Designated Initializers

- Designated initializers are easier to read and check for correctness.
- Also, values in a designated initializer don’t have to be placed in the same order that the members are listed in the structure.
  - The programmer doesn’t have to remember the order in which the members were originally declared.
  - The order of the members can be changed in the future without affecting designated initializers.

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

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

Operations on Structures

- The members of a structure are lvalues.
  - lvalues: object sotred in computer memory (not a constant or the result of a computation)
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258; /* changes part1's part number */
part1.on_hand++; /* increments part1's quantity on hand */
```

Operations on Structures

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.

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

Operations on Structures

- The other major structure operation is assignment:
- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.
  - copy all the members

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

part2 = part1;
```

Operations on Structures

- Some might be surprise that we can use = operator with structure, because arrays can’t be copied using the = operator.
- What even more suprising is that 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

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

Operations on Structures

- The = operator can be used only with structures of ***compatible*** types.
- What is ***compatible***?
- Two structures declared at the same time (as `part1` and `part2` were) are compatible.
- Structures declared using the same “structure tag” or the same type name are also compatible, which we will explain later.
- Other than assignment, C provides no operations on entire structures.
- In particular, the == and != operators can’t be used with structures.

slido

Which one is the correct printed message?

ⓘ Start presenting to display the poll results on this slide.

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 to avoid repeating structure information over and over again.
- Ways to name a structure:
  - Declare a “structure tag”
  - Use `typedef` to define a type name

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`
  - Note that a semicolon must follow the right brace.

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

Declaring a Structure Tag

- The `part` tag can be used to declare variables:
- We can't drop the word `struct`! `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.

```
struct part part1, part2;
part part1, part2;   /*** WRONG ***/
int part; //Correct; doesn't conflict with structure tag
```

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables*

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

Declaring a Structure Tag

- All structures declared to have type `struct part` are **compatible** with one another

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

part2 = part1; /* legal; both parts have the same type */
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`
  - `Part` can be used in the same way as the built-in types

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

Part part1, part2;
```

Defining a Structure Type

- When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`.
- However, declaring a structure tag is mandatory when the structure is to be used in a linked list (Ch 17).

Structures as Arguments and Return Values

- Since structures can be used as a type of data,
- Functions can also have structures as arguments and return values.
- A function with a structure argument:

```
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);
}
…
print_part(part1); //a call of print_part
```

Structures as Arguments and Return Values

- A function that returns a `part` 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;
}
…
part1 = build_part(528, "Disk drive", 10);; //a call of build_part
```

Structures as Arguments and Return Values

- Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
- If the strucuture is complicated (with many members), the program will spend many resources in copying strucutre members (overhead)
- To avoid this overhead, it's sometimes better to pass a **pointer** to a structure or return a pointer to a structure.
- Chapter 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.

Summary

- Structure Variables
- Declaring Strucutre Variables
- Initializing Strucutre Variables
  - Designated initializers
- Operations on Structures
  - . and = operator with structures
- Strucutre Type
  - Structure Tag and Structure Type
- Structures as Arguments and Return Values