# Structures, Unions, and Enumerations (2)

***Program Design (II)***

*2022 Spring*

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

# Quick Recap

- Structure Variables
  - `struct {}`
- Declaring and Initializing Strucutre Variables
- Operations on Structures
  - . and = operator with structures
- Strucutre Type
  - Structure Tag and Structure Type

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

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

slido

# Which one is structure tag?

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

slido

# How to declare new structure variable part3 and part4 using the structure tag?

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

slido

## Which is the correct initializer if I want to initialize the member number of part3 to be 1?

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

# Outline

- Structures as Arguments and Return Values
- Structure Pointer
- Nested Arrays and Structures

```
// we will use this structure part for following examples
struct part {
     int number;
     char name[NAME_LEN+1];
     int on_hand;
};
```

# 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
- Since we can use assignment operator = with `struct`, we can save the return value in the variable `part1` with the type of `struct part`

```
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.**
  - C only does pass by value which means C copies the value of the argument to parameter do any change to parameters will not affect the value of the argument.
- If the strucuture is **complicated** (with **many** members), the program will spend many resources in **copying** strucutre members (**overhead**)

```
void print_part(struct part p){
        …
}
…
print_part(part1); //a call of print_part
```

copy part1 to p every function calls

# Structure Pointer

- To avoid this overhead, it's sometimes better to pass a **pointer** to a **structure** or **return a pointer to a structure**.
- In this way, C will only need to copy the value stored in the structure pointer (the address) when passing by value instead of copy the entire structure variable.
- **How** to declare and use structure pointer (i.e., pointer to a structure variable)?

```
void print_part( structure pointer ){
        …
}
…
print_part( structure pointer );
```

copy address stored in the structure pointer

# Structure Pointer

- A pointer is a variable which points to the address of another variable of any data type like `int, char`
- Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable.
- For example, we declare a pointer `personPtr` that can store the address of the variable of type `struct person`

```
struct person
{
   int age;
   float weight;
};


struct person *personPtr;
```

# Structure Pointer

- We then declare another structure variable `person1`
- Then, we assign the address of variable `person1` to `personPtr` using `&` address operator
- Now, `personPtr` points to the structure variable `person1`

```
struct person
{
   int age;
   float weight;
};


struct person *personPtr;
struct person person1 = {25, 55.5};
personPtr = &person1;
```

# Structure Pointer

- The next question is how to access the members using struture pointer?
- There are two methods, the first method is using indirection `*` operator and dot `.` operation
  - Parentheses around `*personPtr` are necessary because the precedence of dot `.` operator is greater than that of indirection `*` operator.

```
…
struct person *personPtr;
struct person person1 = {25, 55.5};
personPtr = &person1;

printf("Enter age: ", (*personPtr).age);
```

# Structure Pointer

- The second method is to use arrow `->` operatore, which is more readable
- `personPtr->age` is equivalent to `(*personPtr).age`

```
…
struct person *personPtr;
struct person person1 = {25, 55.5};
personPtr = &person1;

printf("Enter age: ", (*personPtr).age);
printf("Enter age: ", personPtr->age);
```

# Structure Pointer - Revise `struct part` example

```c
#include <stdio.h>
#include <string.h>
#define NAME_LEN 20

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

```
Part number: 528
Part name: Disk drive
Quantity on hand: 10
```

```c
void print_part_ptr(const struct part *p){
	printf("Part number: %d\n", ____);
	printf("Part name: %s\n", ____);
	printf("Quantity on hand: %d\n", ____);
}


int main()
{
    struct part part1 = {528, "Disk drive", 10};
    ____part_ptr;
    part_ptr = ____
    print_part_ptr(____);

    return 0;
}
```

**Please complete the gaps**

# Let’s Take a Break!

# Compound Literals

- Compound literals can help us to reduce the number of variables only used once (Chapter 9)
- a compound literals of int array with length of 5

```
int sum_array(int n, int a[n]){
    int i, sum = 0;
    for (i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
int main(){ equivalent to ((int [5]){3, 0, 3, 4, 1})
    ...
    total = sum_array(5, (int []){3, 0, 3, 4, 1});
}
```

# Compound Literals

- A **compound literal** can be used to create a structure  without first storing it in a variable.
- A compound literal consists of a type name within parentheses, followed by a set of values in braces.
- It can be used to create a structure that will be passed to a function

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

# Compound Literals

- A compound literal can also be assigned to a variable
- A compound literal may contain designators, just like a designated initializer
    - Obey the same rules as the designated initializer for structure variable

```
…
part1 = (struct part) {528, "Disk drive", 10};
print_part((struct part) {.on_hand = 10,
                          .name = "Disk drive",
                          .number = 528});
```

# Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Structures may contain **arrays** and **structures** (Nested Structure) as members.
- And **arrays** may have **structures** as their **elements**

# Nested Structures

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

```
struct person_name {
     char first[10];
     char middle_initial;
     char last[10];
};
```

# Nested Structures

- We can use `person_name` as part of a larger structure

```
struct student {
     struct person_name name;
     int id, age;
     char sex;
} student1, student2;
```

# Nested Structures

- Accessing `student1`'s first name, middle initial, or last name requires **two** applications of the . operator:

```
struct student {
     struct person_name name;
     int id, age;
     char sex;
} student1, student2;

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

# Nested Structures

- Having `name` be a structure makes it easier to treat names as units of data.
- A function that displays a name could be passed one `person_name` argument instead of three arguments

```
struct person_name {
     char first[10];
     char middle_initial;
     char last[10];
};

struct student {
     struct person_name name;
     int id, age;
     char sex;
} student1, student2;

display_name(student1.name);
```

# Nested Structures

- Copying the information from a `person_name` structure to the `name` member of a student structure would take one assignment instead of three

```
struct person_name {
     char first[10];
     char middle_initial;
     char last[10];
};

struct student {
     struct person_name name;
     int id, age;
     char sex;
} student1, student2;

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

# Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts

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

struct part inventory[100];
```

# Arrays of Structures

- Accessing a part in the array is done by using subscripting
- Accessing a member within a `part` structure requires a combination of subscripting and member selection (`.` operator)
- Accessing a single character in a part `name` requires subscripting, followed by selection, followed by subscripting
  - accessing the `i`th `part`'s first character in the `name` stored in `inventory`

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

# Initializing an Array of Structures

- Initializing an array of structures is done in much the same way as initializing a **multidimensional** array.
- Each structure has its own **brace-enclosed** (`{ . . . }`) initializer;
- the array initializer wraps another set of braces around the structure initializers.

```
struct part {
      int number;
      char name[NAME_LEN+1];
      int on_hand;
};
                            initializer for inventory[0]
struct part inventory[2] = { {1, "book", 10}, {2, "CD", 20} };
```

# Initializing an Array of Structures

- One reason for initializing an array of structures is that it contains information that won’t change during program execution.
- Example: an array that contains country codes used when making international telephone calls.
- The elements of the array will be structures that store the name of a country along with its code

```
struct dialing_code {
      char *country;
      int code;
};
```

# Initializing an Array of Structures

depend on the length of the initializer

```
const struct dialing_code country_codes[] =
  {{"Argentina",            54}, {"Bangladesh",      880},
   {"Brazil",               55}, {"Burma (Myanmar)",  95},
   {"China",                86}, {"Colombia",         57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",            20},
   {"Ethiopia",            251}, {"France",           33},...};
```

# Initializing an Array of Structures

- We can also use the designated initializers to initialize an array of structures
- For example, the statement initializes the `0`th part (element) in `inventory[100]` with `number` of 528, `on_hand` of 10, and `name` of empty string

```
…
struct part inventory[100] = {[0].number = 528, [0].on_hand = 10,
                              [0].name[0] = '\0'};
```

slido

# (Multiple Choice) what are the correct initializers for st1 (gap 1)?

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

# (Multiple Choice) what are the correct expressions for gap 2?

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

slido

# (Multiple Choice) what are the correct expressions for gap 3?

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

# Summary

- Structures as Arguments and Return Values
- Structure Pointer
  - Reference for the structure pointer
    - https://www.programiz.com/c-programming/c-structures-pointers
    - https://www.javatpoint.com/structure-pointer-in-c
    - https://overiq.com/c-programming-101/pointer-to-a-structure-in-c/
- Compound Literals
- Nested Arrays and Structures
  - Nested Structures
  - Arrays of Structures
  - Initializing an Array of Structures