CS100
Introduction to Programming

Lecture 9. Structures

Learning Objectives

- At the end of this lecture, you should be able to understand and use the following:
  - Structures
  - Arrays of Structures
  - Nested Structures
  - Pointers to Structures
  - Function and Structures
  - The typedef Construct

Structures

- A **structure** is an aggregate of values, in which components are distinct and may possibly have different data types.

- For example, a **record** about a book in a library may contain:

```
char title[40];
char author[20];
float value;
int libcode;
```

Setting up a Structure Template

- A **structure template** is the master plan that describes how a structure is put together. To set up a structure template, e.g.

```
struct book {        /* template of book */
   char title[40];
   char author[20];
   float value;
   int libcode;
};
```

  - struct: the reserved keyword to introduce a structure
  - book: an optional tag name which follows the keyword “struct” to name the structure declared.
  - title, author, value and libcode: the members of the structure book.
- The above declaration declares a template, not a variable. No memory space is allocated.

Structures – Example

```
/* book.c -- one-book inventory */
#include <stdio.h>
struct book {
   char title[40];
   char author[20];
   float value;
   int libcode;
};
int main(void)
{
   struct book bookRec;
   printf("Please enter the book title\n");
   gets(bookRec.title);
   printf("Now enter the author.\n");
   gets(bookRec.author);
   printf("Now enter the value.\n");
   scanf("%f", &bookRec.value);
   printf("%s by %s: $%.2f\n", bookRec.title,
            bookRec.author, bookRec.value);
   return 0;
}
```

Output:

Please enter the book title:

The C Programming Language

Please enter the author:

K & R

Please enter the value:

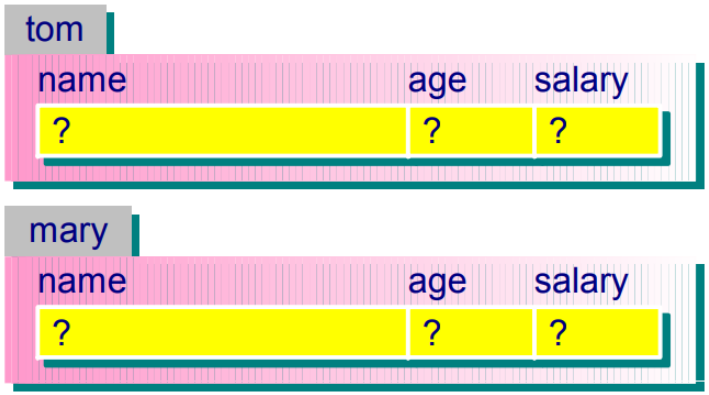
63.65

The C Programming Language by K & R: $63.65

Defining a Structure Variable

- **With tag**: separate the definition of structure template from the definition of structure variable.

```
struct person {
    char name[20];
    int age;
    float salary;
};
struct person tom, mary;
```



- **Without tag**: combine the definition of structure template with that of structure variable.

```
struct {                /* no tag */
    char name[20];
    int age;
    float salary;
} tom, mary;
```

Structure Initialization

- Syntax for initializing structures is **similar to** that for initializing arrays.
- When there are insufficient values assigned to all members of a structure, the remaining members are assigned **zero** by default.
- Initialization of structure variables can only be performed with constant values or constant expressions which deliver values of the required types.

```
struct personTag {
   char name[20];
   int id;
   int tel;
}personInfo;
struct personTag student = {“John”, 123, 20684863};
printf(“%s %d %d\n”, student.name, student.id, student.tel);
```

Output:

John 123 20684863

Structure Assignment and Accessing

Structure Assignment

- The values in one structure can be assigned to another:

```
struct personTag newMember;
newMember = student;
```

Accessing Structure Members

- Notation required to reference the members of a structure is

```
structureVariableName.memberName
```

as shown in the previous example

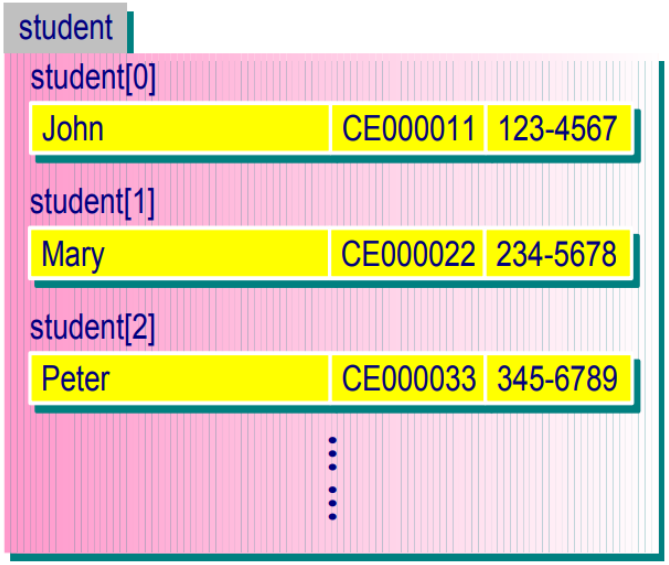
- The "." is a member access operator known as the **member operator**.

Arrays of Structures

- A structure variable can be seen as a **record**, e.g. the structure variable **student** in the previous example is a student record with the information of a student's name, address, id, etc.
- When student variables of the same type are grouped together, we have a **database** of that structure type.
- One can create a database by defining an **array** of certain structure type.

Arrays of Structures – Example

```
/* Define a database with up to 10 student records */
struct personTag {
   char name[20], id[20], tel[20];
};
struct personTag student[3] = {
   {“John”, “CE000011”, “123-4567”},
   {“Mary”, “CE000022”, “234-5678”},
   {“Peter”, “CE000033”, “345-6789”},
};
int main(void)
{
   int i;
   for (i=0; i < 3; i++) {
      printf(“Name: %s, ID: %s, Tel: %s.\n”,
         student[i].name, student[i].id, student[i].tel);
   }
}
```



Output:

Name: John, ID: CE000011, Tel: 123-4567.
Name: Mary, ID: CE000022, Tel: 234-5678.
Name: Peter, ID: CE000033, Tel: 345-6789.

Nested Structures

- A structure can also be included in other structures.
- For example, to keep track of the course history of a student, one can use a structure (**without any nested structures**) like

```
struct studentTag {
    char        name[40];
    char        id[20];
    char        tel[20];
    int         CS100Yr;        /* the year when CS100 is taken */
    int         CS100Sr;        /* the semester when CS100 is taken */
    char        CS100Grade;     /* the grade obtained for CS100 */
    int         CS102Yr;        /* the year when CS102 is taken */
    int         CS102Sr;        /* the semester when CS102 is taken */
    char        CS102Grade;     /* the grade obtained for CS102 */
}
struct studentTag student[1000];
```

Nested Structures

- Alternatively, student can be defined in a more elegant manner, **using nested structures**, as

```
struct personTag  {
    char    name[40];
    char    id[20];
    char    tel[20];
};
struct courseTag  {
    int     year, semester;
    char    grade;
};
struct studentTag  {
    struct personTag studentInfo;
    struct courseTag CS100, CS102;
};
struct studentTag student[1000];
```

- **student** denotes the complete array (database)

```
struct studentTag student[3] = {
      {{“John”, “CE000011”, “123-4567”},
            {2016, 1, ‘B’}, {2017, 1, ‘A’}},
      {{“Mary”, “CE000022”, “234-5678”},
            {2016, 1, ‘A’}, {2017, 1, ‘A’}},
      {{“Peter”, “CE000033”, “345-6789”},
            {2016, 1, ‘C’}, {2017, 1, ‘B’}},
};
/* To print individual elements of the new student array */
int i;
for (i=0; i <= 2; i++) {
      printf(“Name: %s, ID: %s, Tel: %s\n”,
            student[i].studentInfo.name,
            student[i].studentInfo.id,
            student[i].studentInfo.tel);
      printf(“CS100 in year %d semester %d : %c\n”,
            student[i].CS100.year,
            student[i].CS100.semester,
            student[i].CS100.grade);
      printf(“CS102 in year %d semester %d : %c\n”,
            student[i].CS102.year,
            student[i].CS102.semester,
            student[i].CS102.grade);
}
```

- **student[i]** denotes the (i+1)th record
- **student[i].studentInfo** denotes the personal information in the (i+1)th record
- **student[i].studentInfo.name** denotes the student’s name in this record
- **student[i].studentInfo.name[j]** denotes a single character value

Pointers to Structures

- Pointers are flexible and powerful in C. They can be used to point to structures.

```
/* The structure members can be accessed in 3 different ways,
   using pointers or not. */
struct personTag {
   char name[40], id[20], tel[20];
};
struct personTag student = {“John”, “CE000011”, “123-4567”};
struct personTag *ptr;
...
printf(“%s %s %s\n”, student.name, student.id, student.tel);
ptr = &student;
printf(“%s %s %s %s\n”, (*ptr).name, (*ptr).id, (*ptr).tel);
/* Why is the round brackets around *ptr needed? */
printf(“%s %s %s\n”, ptr->name, ptr->id, ptr->tel);
...
```

Pointers to Structures

- The operator **->** is called the **structure pointer operator**, which is reserved for a pointer pointing to a structure. Less typing is needed if one compares **ptr->tel** to **(*ptr).tel**
- 3 reasons for using pointers to structures:
  - Pointers to structures are easier to manipulate than structures themselves;
  - In older C implementation, a structure is passed as an argument to a function using pointer to structure;
  - Many advanced data structures require pointers to structures.

Pointers to Structures: Example

```
#include <stdio.h>
struct book {
   char title[40];
   char author[20];
   float value;
   int libcode;
};

int main(void)
{
   struct book bookRec = {
      “The C Programming Language”, “K&R”, 63.65, 123
   };
   struct book *ptr;
   ptr = &bookRec;
   printf(“The book %s (%d) by %s: $%.2f.\n”, ptr->title,
         ptr->libcode, ptr->author, ptr->value);
   return 0;
}
```

Output:

The book The C Programming Language (123) by K&R: $63.65.

Functions and Structures

- Four ways to pass structure information to a function:
  - Passing structure members as arguments using call by value, or call by reference;
  - Passing structures as arguments;
  - Passing pointers to structures as arguments;
  - Passing by returning structures.

Passing Structure Members as Argument

```
#include <stdio.h>
float sum(float, float);
struct account {
   char bank[20];
   float current;
   float saving;
};

int main(void)
{
   struct account john = {“OCBC Bank”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(john.current, john.saving));   // pass by value
   return 0;
}

float sum(float x, float y)
{
   return (x + y);
}
```

Output:
The account has a total of 5001.30.

- **Pass by value**
- **struct members** are used as arguments

Passing Structure as Argument

```
#include <stdio.h>
struct account {
   char bank[20];
   float current;
   float saving;
};
float sum(struct account); /* argument is a structure */

int main(void)
{
   struct account john = {“OCBC Bank”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(john));    // pass by value
   return 0;
}

float sum(struct account money)
{
   return (money.current + money.saving);
   /* not money->current */
}
```

Output:
The account has a total of 5001.30.

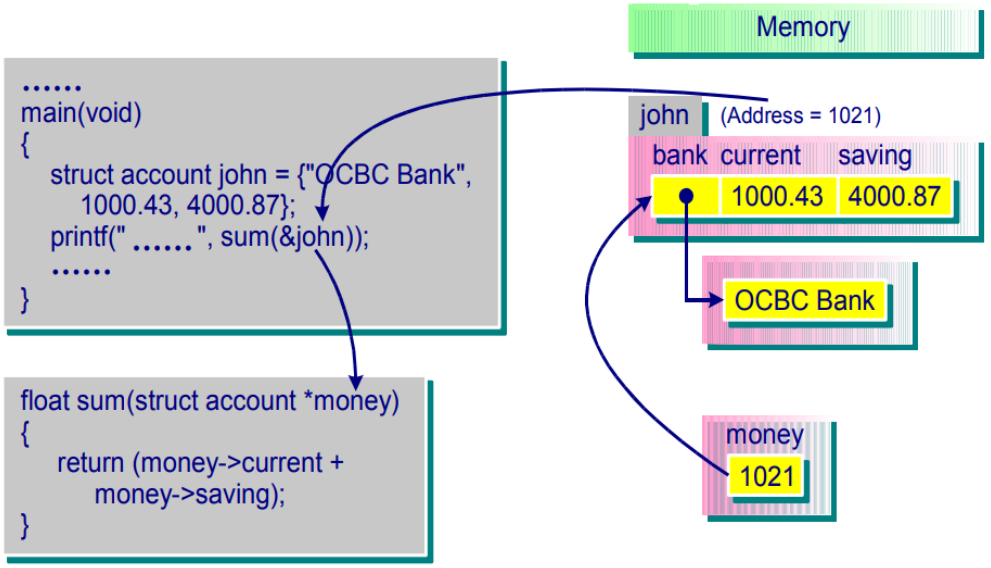
- **Pass by value**
- **struct account money** is used as parameter

Passing Structure Address as Argument

```
#include <stdio.h>
struct account {
   char bank[20];
   float current;
   float saving;
};
float sum(struct account*);

int main(void)
{
   struct account john = {“OCBC Bank”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(&john));   // pass by reference
   return 0;
}

float sum(struct account *money)
{
   return (money->current + money->saving);
}
```



- **Pass by reference**
- **struct account *money** is used as parameter

Returning a Structure in Function

```
#include <stdio.h>
struct nameTag {char Fname[20], Lname[20];};
int main(void) {
   struct nameTag name;
   name = getname();
   printf("Your name is %s %s\n", name.Fname, name.Lname);
   return 0;
}
struct nameTag getname(void) {
   struct nameTag newname;
   printf("Enter first name: ");
   gets(newname.Fname);
   printf("Enter last name: ");
   gets(newname.Lname);
   return newname;
}
```

Output:

Enter first name: ***Jie***

Enter last name: ***Zheng***

Your name is Jie Zheng.

- When is it better to use structures?
- When is it better to use pointers to structures?
- How to pass an array of structures into a function?

The typedef Construct

- **typedef** provides an elegant way in structure declaration. For example, having

```
struct date { int day, month, year; };
```

one can define a new data type **Date** as

```
typedef struct date Date;
```

- Variables can be defined either as

```
struct date   today, yesterday;
```

or

```
Date today, yesterday;
```

- When ***typedef*** is used, tag name is redundant, thus:

```
typedef struct {
        int day, month, year;
} Date;
Date today, yesterday;
```

The typedef Construct: Example

```
#include <stdio.h>
#define CARRIER 1
#define SUBMARINE 2
typdef struct {
    int shipClass;
    char *name;
    int speed, crew;
} warShip;

void printShipReport(warShip);

int main(void)
{
    warShip ship[10];
    int i;
    ship[0].shipClass = CARRIER;
    ship[0].name = “Liaoning”;
    ship[0].speed = 29;
    ship[0].crew = 3000;
    ship[1].shipClass = SUBMARINE;
    ship[1].name = “Changzheng-6”;
    ship[1].speed = 24;
    ship[1].crew = 140;
    for (i=0; i < 2; i++)
        printShipReport(ship[i]);
    return 0;
}

void printShipReport(warShip ship)
{
    if (ship.shipClass == CARRIER)
        print(“Carrier:\n”);
    else
        print(“Submarine:\n”);
    printf(“\tname = %s\n”, ship.name);
    printf(“\tspeed = %d\n”, ship.speed);
    printf(“\tcrew = %d\n”, ship.crew);
}
```

The typedef Construct: Example

Output:

Carrier:

name: Liaoning

speed = 29

crew = 3000

Submarine:

name: Changzheng-6

speed = 24

crew = 140

Recap

- The following concepts have been covered in this lecture:
  - Structures
  - Arrays of Structures
  - Nested Structures
  - Pointers to Structures
  - Function and Structures
  - The typedef Construct