7
Structures

1

Structures

1. In this lecture, we discuss Structures in C.

Why Learning Structures?

- Arrays are used to store a collection of unrelated data items of the **same** data type.
- C also provides a **data type** called ***structure*** that stores a collection of data items of different data types as a group.
- The individual components of a structure can be of any valid data types.
- In this lecture, we describe the **struct** data type.

2

Why Learning Structures?

1. Arrays are used to store a collection of unrelated data items of the same data type.
2. C also provides a **data type** called *structure* that stores a collection of data items of different data types as a group. The individual components of a structure can be of any valid data types.
3. In this lecture, we describe the **struct** data type.

Structures

- **Structure Declaration, Initialization and Operations**
- Arrays of Structures and Nested Structures
- Pointers to Structures
- Functions and Structures
- The typedef Construct
- Reading Inputs from Mixed Data Types

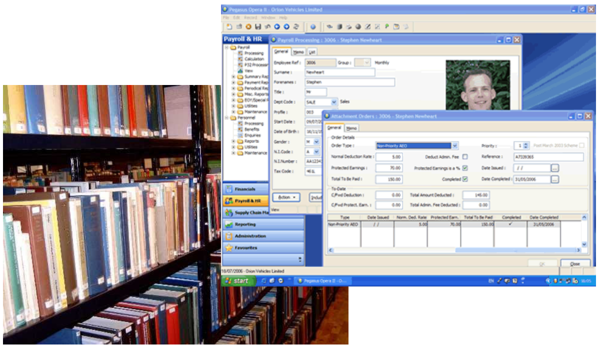
3

Structures

1. Here, we discuss structure declaration, initialization and operations.

Records

- Records are used to keep related information of an object together.
- Examples:
  - Medical Records
  - Book Records
  - Employee Records
  - Etc.



- Structure is similar to record in that it is used to keep related data together as a data type.

4

Records

1. Records are used to keep related information of an object together.
2. There are many examples of records such as medical records, book records, employee records, etc.
3. Structure is similar to record in that it is used to keep related data together as a data type.

Structures

- Structure is an **aggregate** of **values**, its components are **distinct**, and it may possibly have different types.
- For example, a **record** about a book (i.e. book record) in a library may contain:
  - **char** title[40];
  - **char** author[20];
  - **float** value;

[Note: a record may have data from **different** data types]

- Two steps in order to use a structure:
  1. Define a **structure template** (similar to a data type).
  2. Declare a **variable** on the structure template.

5

Structures

1. Structure is an aggregate of values. Their components are distinct, and may possibly have different types, including arrays and other structures.
2. For example, a book record may contain the title, author and book value.
3. We can create a **structure template**, which can be defined as a **data type** with different data members, to specify the book record. It tells the compiler the various components of a book record that make up the structure.
4. **Structure variables** can then be declared with the type of the structure.
5. Therefore, to use structure in a program, there are two steps:
   - Define a structure template (or data type).
   - Declare a variable based on the structure data type.

Defining 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];        /* members */
   float value;
};
```

  - **struct**: reserved keyword to introduce a structure
  - **book**: an **optional** **tag name** which follows the keyword struct to name the structure declared.
  - **title, author, value**: the **member** of the structure book.

  Note - The above declaration just declares a template, not a variable. **No memory space** is allocated.

6

Defining a Structure Template

1. A structure template (or data type) is the master plan that describes how a structure is put together.
2. A structure template can be set up as shown in **struct book**:

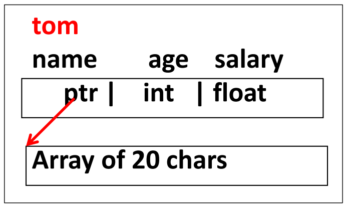
```
struct book {        /* struct book defines the template of book*/
    char title[40];  /* title, author, value are members of the structure */
    char author[20];
    float value;
};                   /* semicolon to end the definition */
```

2. The word **struct** is a reserved keyword to introduce a structure. The name **book** is an optional tag name that follows the keyword **struct** to name the structure declared. The **title**, **author** and **value** are the *members* of the structure **book**.
3. The members of a structure can be any of the valid C data types.
4. A semicolon after the closing brace ends the definition of the structure definition.
5. The declaration declares a template (or data type), not a variable. Therefore, no memory space is allocated. It only acts as a template for the named structure type. The tag name **book** can then be used for the declaration of variables.

Declaring Structure Variable: with Tag Name

- With tag name: 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;
```

- With tag name – we can use the structure type subsequently in the program.

7

Declaring Structure Variable: with Tag Name

1. The structure name or tag is optional.
2. With structure tag, the definition of structure template can be separated from the definition of structure variables. As shown in the declaration **struct person**, a structure template **person** comprising three components **name**, **age** and **salary** is created.
2. **tom** and **mary** are two structure variables which are declared using the structure **person**.
3. With tag name, we can use the structure data type subsequently in the program.

Declaring Structure Variable: without Tag Name

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

```
struct {                    /* no tag – person is not used */
   char name[20];
   int age;
   float salary;
} tom, mary;
```

- Without tag name – we cannot use the structure type elsewhere in the program.

8

Declaring Structure Variable: without Tag Name

1. Without structure tag, the definition of structure template must be combined with that of structure variables.
2. As shown in the structure declaration, a structure template is created with three components: **name**, **age** and **salary**.
3. The variables **tom** and **mary** are then defined using this structure.
4. Without structure tag name, we cannot use the structure elsewhere in the program.
5. It is always a good idea to include a structure tag when defining a structure.

Accessing Structure Members

- The notation required to reference the members of a structure is

  structureVariableName.memberName

- The "**.**" (dot notation) is a member access operator known as the **member operator**.

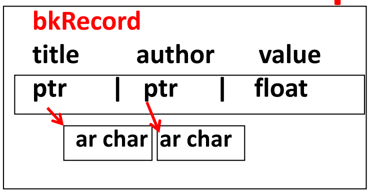
- For example, to access the member **age** of the variable **tom** from the struct person, we have **tom.age**.

9

Accessing Structure Members

1. The notation required to access a member of a structure is **structureVariableName.memberName**
2. The "**.**" is an access operator known as the **member operator**. The member operator has the highest (or equal) priority among the operators in the operator precedence table.
3. For example, to access the member **age** of the variable **tom** from the structure **person**, we have **tom.age**

Structure Declaration & Operation: Example




```c
#include <stdio.h>
#include <string.h>
struct book {
     char title[40];
     char author[20];
     float value;
};
int main()
{
     char *p;
     struct book bkRecord;
     printf("Please enter the book title: \n");
     fgets(bkRecord.title, 40, stdin);   /* to access member, using . notation */
     if ( p=strchr(bkRecord.title,'\n') ) *p = '\0';
     printf("Please enter the author: \n");
     fgets(bkRecord.author, 20, stdin);
     if ( p=strchr(bkRecord.author,'\n') ) *p = '\0';
     printf("Please enter the value: \n");
     scanf("%f", &bkRecord.value);  /*note: & is needed here*/
     printf("%s by %s: $%.2f\n", bkRecord.title, bkRecord.author,
               bkRecord.value);
     return 0;
}
```

Variable name

Output
Please enter the book title:
C Programming
Please enter the author:
SC Hui
Please enter the value:
30.00
C Programming by SC Hui: $30.00

10

Structure Declaration and Operation: Example

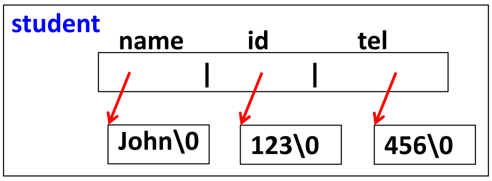
1. In the program, it defines the structure template (or data type) **book** and the declaration of a structure variable **bkRecord**.
2. The structure definition can be placed inside a function or outside a function. If it is defined inside the function, the definition can only be used by that function.
3. In the program, the structure template **struct book** is defined outside the **main()** function. It is a global declaration, and all the functions following the definition can use the template.
4. In the **main()** function, it declares a variable **bkRecord** of type **struct book**. The storage space is then allocated for the variable.
5. The **fgets()** function is used to read the user input on title and author which are character strings: To access a member of a structure, we use the dot notation such as **bkRecord.title** and **bkRecord.author**.
5. The **scanf()** statement **scanf("%f", &bkRecord.value);** will read the user input on book value which is of data type **float**.
6. After reading the user input, the book title, author and book value will be printed on the screen.

Structure Variable: Initialization

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

```
struct personTag{
        char    name[20];
        char    id[20];
        char    tel[20];
}student = {"John", "123", "456"};

printf("%s %s %s\n", student.name, student.id,
        student.tel);
```



Output
John 123 456

using . notation

11

Structure Variable: Initialization

1. The syntax for initializing structures is similar to that of initializing arrays. When there are insufficient values to be assigned to all members of the structure, the remaining members are assigned to zero by default.
2. The structure variable **student** is declared, and followed by an assignment symbol and a list of values defined within braces: **student = {"John", "123", "456"};**
3. Initialization of variables can only be performed with constant values or constant expressions that deliver a value of the required type. The initial values are assigned to the individual members of the structure in the order in which the members occur. The **name** member of **student** is assigned with "**John**", the **id** member is assigned with **"123"**, and the **tel** member is assigned with **"456"**.
4. The **printf()** statement prints the data of the structure variable **student** using dot notation to access the member of structure: **student.name, student.id, student.tel**.

Structure Assignment

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

  struct personTag **newmember**;

  newmember = **student**;

- This has the effect of copying the entire contents of the structure variable **student** to the structure variable **newmember**. Each **member** of the **newmember** variable is assigned with the value of the corresponding **member** in the **student** variable.

  Analogy (using primitive data type):

  ```
  int num=10;
  int member;
  member = num;
  ```

12

Structure Assignment

1. The value of one structure variable can be assigned to another structure variable of the same type using the assignment operator.
2. First, we define a new variable **newmember** under the data type **struct personTag**: **struct personTag newmember;**
3. Then, we can assign the **struct personTag** variable **student** to **newmember**: **newmember = student;**
4. This has the effect of copying the entire contents of the structure variable **student** to the structure variable **newmember**. Each member of the **newmember** variable is assigned with the value of the corresponding member in the **student** variable.
5. The structure assignment operation is similar to primitive variable assignment operation.

Structures

- Structure Declaration, Initialization and Operations
- **Arrays of Structures and Nested Structures**
- Pointers to Structures
- Functions and Structures
- The typedef Construct
- Reading Inputs from Mixed Data Types

13

Structures

1. Here, we discuss arrays of structures and nested structures.

Arrays of Structures

- **Record** - 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 name, id, tel, ...
- **Database** - When structure variables of the same type are grouped together, we have a database of that structure type.
- **Array of Structures** - One can create a database by defining an **array** of certain structure type.

14

Arrays of Structures

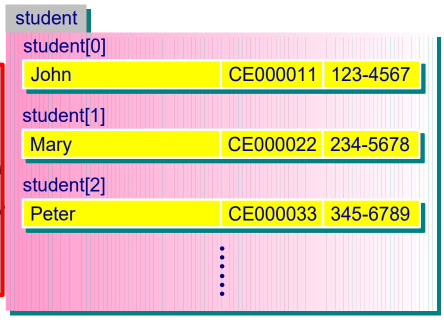
1. A structure variable can be seen as a **record**. For example, the structure variable **student** is a student record with the information of a student name, identity and telephone number.
2. When structure variables of the same type are grouped together, we can form a **database** of that structure type.
3. Therefore, we can create a database by defining an **array of structures**.

Arrays of Structures: Declaration & Initialization

```
/* Define a database with up to 10 student records */
struct personTag {
   char  name[40], id[20], tel[20];
} ;
struct personTag student[10] = {
    { "John", "CE000011", "123-4567"},
    { "Mary", "CE000022", "234-5678"},
    { "Peter", "CE000033", "345-6789"},
        ......
 };
int main( ) {

    // access each structure in array

}
```



15

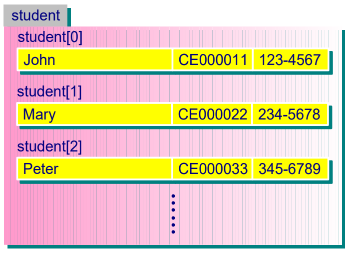
Arrays of Structures: Declaration and Initialization

1. In the program, the variable **student** defines an array of structures, which is a database of student records.
2. Each element of the array is of **struct personTag**. It means each array element contains three members, namely **name**, **id** and **telephone**, of the structure.
3. The syntax for declaring an array of structures is **struct personTag student[10];** where it starts with the keyword **struct** and followed by the name of the structure **personTag** that identifies the data type. This is then followed by the name of the array, **student**. The values specified within the square brackets specify the total number of elements in the array.
4. Array of structures can be initialized as shown. The initializers for each element are enclosed in braces, and each member is separated by a comma. An example is given as follows:

```
struct personTag student[10] = {
    {"John", "CE000011", "123-4567"},    /* for student[0] */
    {"Mary", "CE000022", "234-5678"},    /* for student[1] */
    {"Peter", "CE000033", "345-6789"},   /* for student[2] */
    ...
};
```

Arrays of Structures: Operation

```
/* Define a database with up to 10 student records */
 struct personTag {
   char  name[40], id[20], tel[20];
};
struct personTag student[10] = {
    { "John", “CE000011", "123-4567"},
    { "Mary", “CE000022", "234-5678"},
          ……
 };
int main( ) {
   int   i;
  for (i=0; i<10; 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

using array index and . operator

16

Arrays of Structures: Operation

1. Array index is used when accessing individual elements of an array of structures.
2. We use **student[i]** to denote the (i+1)th record. The first element starts with index 0.
3. To access a member of a specific element, we use **student[i].name** which denotes a member of the (i+1)th record.
4. Therefore, to access each array element, we use a **for** loop to traverse the array.
5. The array index is used to traverse the array, and the member (or dot) operator is used to access each member of the structure in the array element (e.g. **student[i].name, student[i].id, student[i].tel**).

Nested Structures

- For example, to keep track of the course history of a student, one can use a structure as follows :

```
struct studentTag {    // without any nested structures
    char name[40];
    char id[20];
    char tel[20];
    int   SC101Yr;  /* the year when SC101 is taken * /
    int   SC101Sr;  /* the semester when SC101 is taken */
    char SC101Grade;  /* the grade obtained for SC101 */
    int   SC102Yr;  /* the year when SC102 is taken */
    int   SC102Sr;  /* the semester when SC102 is taken */
    char SC102Grade;  /* the grade obtained for SC102 */
};
struct studentTag student[1000];
 // student – array of 1000 student records
```

(1) Student info

(2) Course info: SC101

(3) Course info: SC102

- Instead, we can use a nested structure – refers to a structure that **includes** other structures.

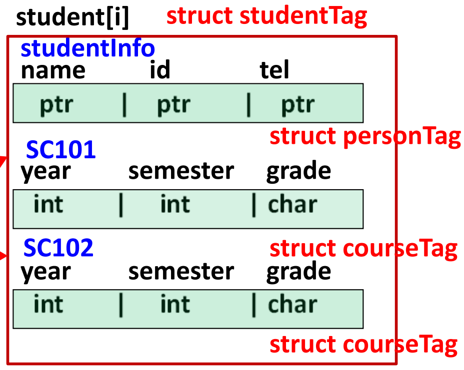
17

Nested Structures

1. In nested structures, a structure can also be included in other structures.
2. For example, to keep track of the course history of a student, one can define a structure (without any nested structures) as shown in **struct studentTag**:
2. In the structure template definition **struct studentTag**, the members are student information including **name**, **id** and **tel**. In addition, it also includes the courses (i.e. SC101 and SC102) that are taken by the student.
3. Once the **struct studentTag** is defined, an array variable **student** of 1000 elements of type **struct studentTag** is created.

Nested Structures

- Alternatively, struct **studentTag** can be defined in a more elegant manner using **nested structures**:




```
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     SC101, SC102;
};   // Nested structure
struct studentTag student[1000];
```

18

Nested Structures

1. Alternatively, the variable **student** can be defined in a more elegant manner using nested structures.
2. We create a structure template called **personTag** to contain the student information which has three members, namely **name**, **id** and **tel**, of array of characters.
3. We also create a structure template called **courseTag** to contain the course information which has three members, namely **year** and **semester** of type **int**, and **grade** of type **char.**
4. Then, we define the nested structure **studentTag** which has three members:
    - **studentInfo** which is a structure of **personTag**;
    - **SC101** and **SC102** which are structures of **courseTag**.
7. Note that the structure definition of **personTag** and **courseTag** must appear before the definition of structure **studentTag**.

Nested Structures: Initialization

- In the program, after defining the nested structure **studentTag**, the array of structures variable **student** can be declared and initialized with initial data.
- The initialization is very similar to that of initializing multi-dimensional arrays.
- In the following example code:

```
/* Array variable initialization */
struct studentTag student[3] = {
    { {"John","CE000011","123-4567"},          // for student[0]
     {2002,1,'B'},
     {2002,1,'A'} },
    { {"Mary","CE000022","234-5678"},          // for student[1]
     {2002,1,'C'},
     {2002,1,'A'} },
    { {"Peter","CE000033","345-6789"},
      {2002,1,'B'},                          // for student[2]
      {2002,1,'A'} }
};
```

19

Nested Structures: Initialization

1. In the program, after defining the nested structure **studentTag,**the array of structures variable **student** can be declared and initialized with initial data.
2. The initialization is very similar to that of initializing multi-dimensional arrays.

Nested Structures: Operation

```
/* To print individual elements of the array of structures*/
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("SC101 in year %d semester %d : %c\n",
        student[i].SC101.year,
        student[i].SC101.semester,
        student[i].SC101.grade);
        printf("SC102 in year %d semester %d : %c\n",
        student[i].SC102.year,
        student[i].SC102.semester,
        student[i].SC102.grade);
}
```

Note: Using dot (member operator) to access members of structures.

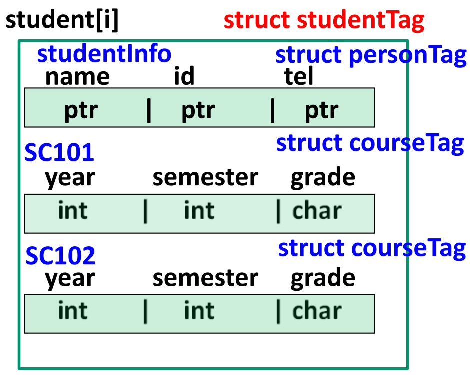
20

Nested Structures: Operation

1. To access each array element, we use a **for** loop to traverse the array.
2. The array notation and member operator are used for accessing each array element and structure member. The data can then be processed and printed on the screen.

Nested Structures: Notations

- **student[i]** denotes the *i+1th* array record. It consists of three members: studentInfo, SC101, SC102.
- **student[i].studentInfo** denotes the personal information in the *i+1th* record. It conssits of three members: name, id, tel.
- **student[i].studentInfo.name** denotes the student name in this record.
- **student[i].studentInfo.name[j]** denotes a single character value.
- **student[i].SC101, student[i].SC102** denote the course information in the *i+1th* record. Each consists of three members: year, semester, grade.



21

Nested Structures: Notations

1. In the nested structure variable **student**, we note the following notations:
   - **student**, which denotes the complete array (i.e. the database);
   - **student[i]**, which denotes the (i+1)th record;
   - **student[i].studentInfo**, which denotes the personal information in the (i+1)th record;
   - **student[i].studentInfo.name**, which denotes the student name in the (i+1)th record; and
   - **student[i].studentInfo.name[j]**, which denotes a single character value in the (i+1)th record.

Structures

- Structure Declaration, Initialization and Operations
- Arrays of Structures and Nested Structures
- **Pointers to Structures**
- Functions and Structures
- The typedef Construct
- Reading Inputs from Mixed Data Types

22

Structures

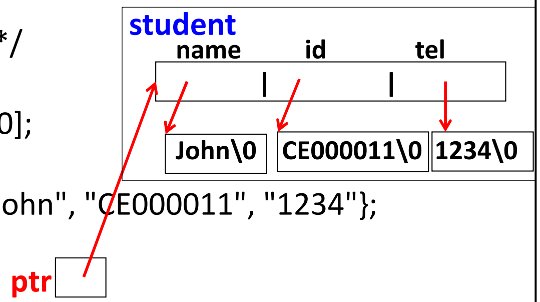
1. Here, we discuss pointers to structures.

Pointers to Structures

- **Pointers** can be used to point to structures.

```
/* Using pointer to structure */
struct personTag {
    char name[40], id[20], tel[20];
} ;
struct personTag student = {"John", "CE000011", "1234"};

struct personTag *ptr;
…
printf("%s %s %s\n", student.name, student.id, student.tel);
ptr = &student;
```



Analogy:

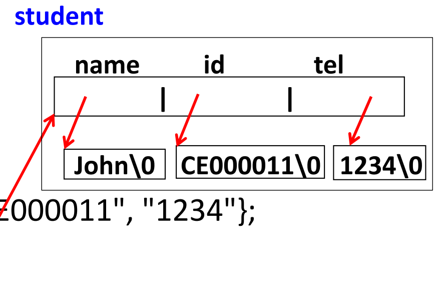
```
int num=10;
int *p;
p = &num;
```

23

Pointers to Structures

1. Pointers can be used to point to structures.
2. The variable **student** of **struct personTag** is declared with initialization: **struct personTag student={"John","CE011","1234"};**
2. Next, we create a pointer **ptr** to the structure **personTag**: **struct personTag *ptr;**
3. Then, we use the address operator (**&**) to obtain the address of a structure variable, and then assign the address to the pointer: **ptr = &student;**
4. As such, we can use the pointer variable **ptr** to access the contents in the structure variable **student**.

Pointers to Structures: Operation




```
/* Using pointers to structure */
struct personTag {
    char name[40], id[20], tel[20];
} ;
struct personTag student = {"John", "CE000011", "1234"};
struct personTag *ptr;
...
printf("%s %s %s\n", student.name, student.id, student.tel);
ptr = &student;

printf("%s %s %s\n", (*ptr).name, (*ptr).id, (*ptr).tel );

/* Why is the round brackets around *ptr needed?
  – op precedence */
```

24

Pointers to Structures: Operation

1. The **indirection operator** (*****) can be used to access a member of a structure via a pointer to the structure.
2. Since **ptr** points to the structure **student**, the notations **(*ptr).name**, **(*ptr).id** and **(*ptr).tel**, return the value of the member **name**, **id** and **tel** of **student** respectively.
3. Note that the parentheses are necessary to enclose ***ptr** as the member operator (**.**) has higher operator precedence than the indirection operator (*****).

Pointers to Structures: Operation

- To access a structure member via pointer, dereferencing is used as illustrated in the previous example:

  printf("%s %s %s\n", (*ptr).name, (*ptr).id, (*ptr).tel);

- Instead, we can use the **structure pointer operator (->)** for a pointer pointing to a structure:

  printf("%s %s %s\n", ptr->name, ptr->id, ptr->tel);

- Note that it is quite common to use the structure pointer operator (**->**) instead of the indirection operator (*) in pointers to structures.

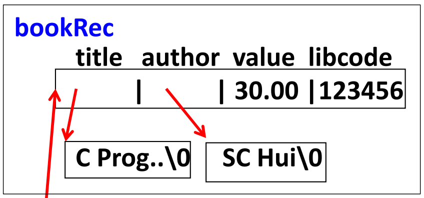
25

Pointers to Structures: Operation

1. Since dereferencing is very common in pointer to structure, C provides an operator called the **structure pointer operator** (**->**) for a pointer pointing to a structure. There is no whitespace between the symbols (**-**) and (**>**).
2. We can use the notations **ptr->name**, **ptr->id** and **ptr->tel** to obtain the values of the members of the structure **student**.
3. It takes less typing when **ptr->tel** is compared with **(*ptr).tel**, though they have exactly the same meaning.
4. It is quite common to use the structure pointer operator (**->**) instead of the indirection operator (*****) in pointers to structures.

Pointers to Structures: Example

```
#include <stdio.h>
struct book {
    char title[40];
    char author[20];
    float value;
    int libcode;
  };
int main()
  {
    struct book bookRec = {
      "C Programming", "SC Hui",
      30.00, 123456
    };
    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 C Programming (123456) by SC Hui: $30.00.

26

Pointers to Structures: Example

1. We can use the structure variable to access each member of the structure. We can also use pointer variable to access each member of the structure.
2. In the program, we define a structure called **book** with four members: **title**, **author**, **value** and **libcode**.
3. After that, we define a structure variable called **bookRec**, and initialize it with values.
4. We then define the pointer variable **ptr** to the **struct book** type: **struct book *ptr;**
5. We assign the address of the structure variable **bookRec** to the pointer variable **ptr**: **ptr = &bookRec;** Therefore, the pointer variable contains the address of **bookRec**.
6. As a result, we may access the members of **bookRec** via **ptr**.
7. In the **printf()** statement, it uses structure pointer operator to access each individual member of the **bookRec** structure and prints each member information of **bookRec**.

Structures

- Structure Declaration, Initialization and Operations
- Arrays of Structures and Nested Structures
- Pointers to Structures
- **Functions and Structures**
- The typedef Construct
- Reading Inputs from Mixed Data Types

27

Structures

1. Here, we discuss functions and structures.

Functions and Structures

- It is often necessary to pass structure information to a function. In C, there are **four** ways to pass structure information to a function:
  1. Passing **structure members** as arguments using call by value, or call by reference;
  2. Passing **structures** as arguments;
  3. Passing **pointers to structures** as arguments; and
  4. Passing by **returning structures**.
- Basically, parameter passing between functions using structure is **similar** to passing data of other basic data types such as int, float, etc.

28

Functions and Structures

1. It is often necessary to pass structure information to a function. In C, there are four ways to pass structure information to a function:
   1) Passing structure members as arguments using call by value, or call by reference;
   2) Passing structures as arguments;
   3) Passing pointers to structures as arguments; and
   4) Passing by returning structures.
2. Basically, parameter passing between functions using structure is similar to other basic data types such as **int**, **float**, etc.

Passing Structure Members as Arguments

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

- **Using call by value**
- **struct members** are used as arguments

29

Passing Structure Members as Arguments

1. In the program, a structure template **account** is defined with three members: **bank**, **current** and **saving**.
2. In the **main()** function, an **account** structure variable **john** is declared with initial values. The structure members **john.current** and **john.saving** are passed to the function **sum()** when it is called.
3. The function **sum()** is used to compute the total amount from the saving and current accounts. It has two parameters, **x** and **y**, of type **float**. When it is called, the structure members **john.current** and **john.saving** are passed to the parameters **x** and **y** respectively. Then, it computes the sum of **x** and **y**, and returns the result to the calling **main()** function.

Passing Structure as Argument: Call by Value

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

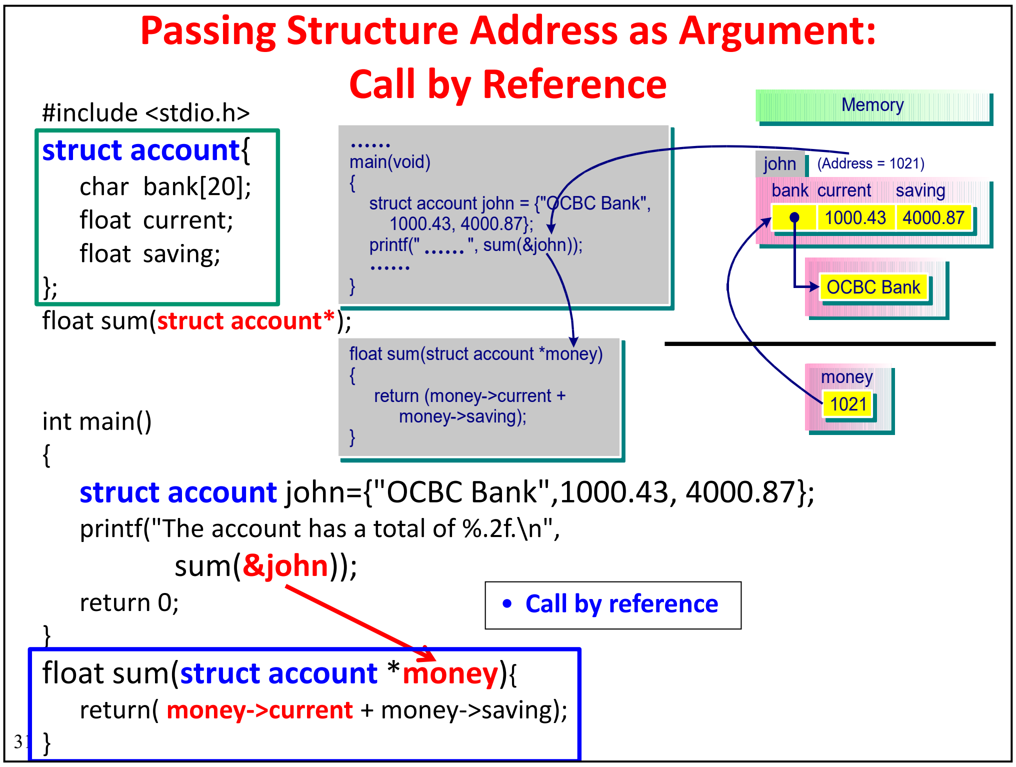
Output
The account has a total of 5001.30.

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

30

Using Call be Value by Passing Structures as Arguments

1. We can pass a structure as an argument to a function using the **call by value** method.
2. In the **main()** function, the structure variable **john** is passed as an argument to the function **sum()**.
3. The members of the structure parameter **money** in the function **sum()** are initialized with local copies. The function can only modify the local copies. Note that we simply use the member operator (**.**) to access the individual members of the structure variable.
4. The advantage of using this method is that the function cannot modify the members of the original structure variables, which is safer than working with the original variables.
5. However, this method is quite inefficient to pass large structures to functions. In addition, it also takes time and additional storage to make a local copy of the structure.



Using Call by Reference by Passing Structure Address as Argument

1. We can also pass the address of the structure as an argument to a function using the **call by reference** method.
2. In the **main()** function, the address of the structure variable **john** is passed as an argument to the function **sum()**.
3. In the function **sum()**, the pointer parameter **money** is used to point to the structure **john**. The structure pointer operator **(->)** is then used to access the members of the structure **account** to obtain the values of **john.current** and **john.saving**. This allows the function to access the structure variable and to modify its content.
4. This is a better approach than passing structures as arguments.

Passing by Returning a Structure

```
#include <stdio.h>
#include <string.h>
struct nameTag {
   char fname[20], lname[20];
};
struct nameTag getname();
int main()
{
   struct nameTag name;
   name = getname();
   printf("Your name is %s %s\n", name.fname, name.lname);
   return 0;
}
struct nameTag getname () {
   struct nameTag newname;
   char *p;
   printf("Enter first name: ");
   fgets(newname.fname, 20, stdin);
   if (p=strchr(newname.fname,'\n')) *p = '\0';
   printf("Enter last name: ");
   fgets(newname.lname, 20, stdin);
   if (p=strchr(newname.lname,'\n')) *p = '\0';
   return newname;
}
```

Output
Enter first name: *Siu Cheung*
Enter last name: *Hui*
Your name is Siu Cheung Hui

- **Call by value (mainly)**
- Returning the structure to the calling function
- Similar to returning a variable value in basic data type

32

Passing by Returning a Structure

1. The function **getname()** returns a structure **nameTag**.
2. To call this function, the calling **main()** function must declare a variable of type **struct nameTag** in order to receive the result from **getname().**
3. It assigns the returned structure data to the variable **name** in the **main()** function.

Structures

- Structure Declaration, Initialization and Operations
- Arrays of Structures and Nested Structures
- Pointers to Structures
- Functions and Structures
- **The typedef Construct**
- Reading Inputs from Mixed Data Types

33

Structures

1. Here, we discuss the **typedef** construct.

The typedef Construct

- **typedef** provides an elegant way in structure declaration. For example, after defining the structure template:

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

- We can define a **new data type Date** as

```
typedef struct date Date;
```

We use typedef to define a new data type Date with the structure template.

- Then, variables can be declared either as

```
struct date     today, yesterday;   or
Date            today, yesterday;
```

- Alternatively, when **typedef** is used, **tag name is redundant,** thus:

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

No tag name – date

Define variables

34

The typedef Construct

1. **typedef** provides an elegant way in structure declaration.
2. The general syntax for the **typedef** statement is **typedef dataType UserProvidedName;** The **typedef** keyword is followed by the data type and the user provided name for the data type.
2. It is very useful for creating simple names for complex structures.
3. For example, if we have defined the structure **struct date**, we can define a new data type **Date** as **typedef struct date Date;**
3. Variables can then be declared either as

   struct date today, yesterday; or

   Date today, yesterday;
4. We can also use the type **Date** in function prototypes and function definitions.
5. When **typedef** is used, tag name is redundant. Therefore, we can declare

   typedef struct {

   int day, month, year;

   } Date;

   Date today, yesterday;
6. There are a number of advantages of using **typedef**. It enhances program

documentation by using meaningful names for data types in the programs. It makes the program easier to read and understand. Another advantage is to define simpler data types for complex declarations such as structures.

The typedef Construct: Example

```
#define CARRIER    1
#define SUBMARINE   2
typedef struct {
   int  shipClass;     char  *name;
   int  speed,crew;
} warShip;
void printShipReport(warShip);
int main()  {
   warShip ship[2];    int i;
   ship[0].shipClass = CARRIER;
   ship[0].name = "Washington";
   ship[0].speed = 40;
   ship[0].crew = 800;
   ship[1].shipClass = SUBMARINE;
   ship[1].name = "Rogers";
   ship[1].speed = 100;
   ship[1].crew = 800;
   for (i=0; i<2; i++)
      printShipReport(ship[i]);
   return 0;  }
```

```
/* Printing each record */
void printShipReport(warShip ship)
{
   if (ship.shipClass == CARRIER)
      printf("Carrier:\n");
   else
      printf("Submarine:\n");
   printf("\tname = %s\n", ship.name);
   printf("\tspeed = %d\n", ship.speed);
   printf("\tcrew = %d\n", ship.crew);
}
```

Output

```
Carrier:
    name: Washington
    speed = 40
    crew = 800
Submarine:
    name = Rogers
    speed = 100
    crew = 800
```

35

The typedef Construct: Example

1. In this program, we use **typedef** to define a new structure type **warship** as shown:

   typedef struct {
   int shipClass;
   char *name;
   int speed,crew;
   } warShip;

2. In the **main()** function, we declare an array of **warShip** structures variable called **ship**. A **for** loop is used to print the member information of the variable **ship** by calling the function **printShipReport()**.
3. The function **printShipReport()** is used for printing the member information of the **warShip** structure.

Structures

- Structure Declaration, Initialization and Operations
- Arrays of Structures and Nested Structures
- Pointers to Structures
- Functions and Structures
- The typedef Construct
- **Reading Inputs from Mixed Data Types**

36

Structures

1. Here, we discuss the considerations when reading input data from mixed data types.

Common Error on Reading Input Data

```
#include <stdio.h>
int main( )
{
    int number;
    char reply;

    printf("Enter a number: ");
    scanf("%d", &number); // read in an integer
    printf("The number read is %d\n", number);

    printf("Correct (y/n)? ");
    scanf("%c", &reply);      // read in a char

    printf("your reply : %c\n", reply);   // display the char
    return 0;
}
```

Intended Input/Output:

Enter a number: ***1234<Enter>***
The number read is 1234
Correct (y/n)? ***y***
your reply : y

Can the program compile correctly?
Can the program run as intended?

37

Common Error on Reading Input Data

1. In the example program, it reads the user input using the **scanf()** function.
2. The intended execution of the program is that it first prompts the user to enter a number, then reads in the number with the **scanf()** function and asks the user whether the number is correct.
3. The **reply** will be read in as a single character (i.e. **y** or **n)** for yes or no using **scanf()**.
4. Then, the user's reply will be printed to the screen.
5. Question: Is there any syntax error in this program? If no, can the program be run as intended?

Common Error on Reading Input Data: Problem

```c
#include <stdio.h>
int main( )
{
      int number;
      char reply;
      printf("Enter a number: ");
      scanf("%d", &number); //read in an integer
      printf("The number read is %d\n", number);

      printf("Correct (y/n)? ");
      scanf("%c", &reply);       //read in a char

      printf("your reply : %c\n", reply);  //display the char
      return 0;
}
```

When the program runs:

Output

Enter a number: ***1234<Enter>***

The number read is 1234

Correct (y/n)? your reply :

// an error here

// the reply is not read

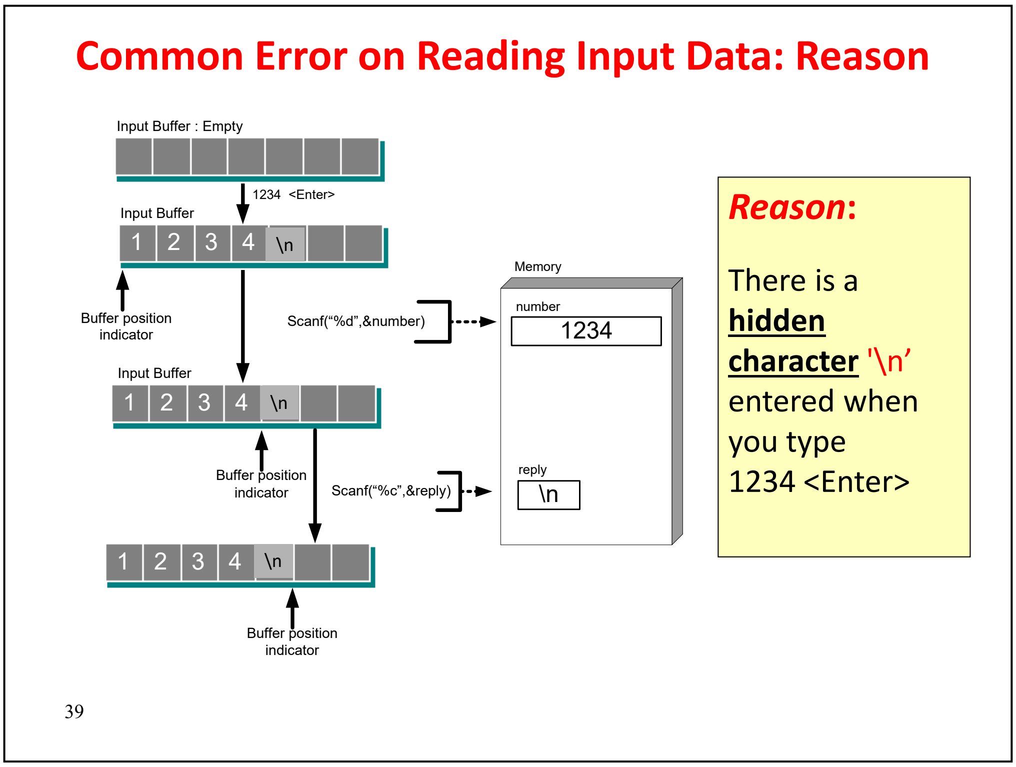
Can the program compile correctly?

Can the program run as intended?

38

Common Error on Reading Input Data: Problem

1. When the program is written and run, after the number has been entered with the **<Enter>** key, the number is read.
2. When the program prompts the user with the message "**Correct (y/n)?**", it does not wait for any new user input.
3. Instead, the newline character will be displayed as the reply of the user.
4. Therefore, the program does not run correctly as intended. It means that the program does not read in user reply on **yes** or **no**.



Common Error on Reading Input Data: Reason

1. When the user enters **1234<Enter>**, the value will first be stored in the input buffer.
2. After executing **scanf**("%d", **&**number); the value **1234** will then be assigned to the variable **number**.
3. When executing **scanf**("%c", **&**reply); the program tries to read in the user input from the input buffer.
4. As the input buffer still stores the **newline** character **'\n'**, which will then be used and assigned to the variable **reply**.
5. As such, the program does not wait for user to enter his reply.

Reading Mixed Data Input

- Note that data input errors may occur when the program reads in **data** from **mixed data types**.
- For example, the program reads in an **integer** (into the variable **number**), then followed by reading in a **character** (into the variable **reply**).
- Generally, this kind of problem will occur when the program first reads in a **number** or **float/double**, then followed by reading in a **character** or **string**, or vice versa.
- To tackle this kind of problem, we will need to remove the newline character **'\n'** from the input buffer before reading the next data input.

40

Reading Mixed Data Input

1. When you read data input which consists of data from different data types (e.g. to read integer first, then read character or vice versa), you may need to consider the extra newline character **'\n'** in the input buffer.

Reading Mixed Data Input: Solution

- **1: read in '\n' (recommended)**

```
    . . .
printf("Correct (y/n)?");
scanf("\n");   // read newline
scanf("%c", &reply);
printf("Your reply: %c\n", reply);
    . . .
```

2:

```
char dummy;
…
printf("Correct (y/n)?");
scanf("%c", &dummy);
scanf("%c", &reply);
printf("Your reply: %c\n", reply);
```

- **3: using fflush() (Not Recommended for APAS)**

```
int number; char reply;
printf("Enter a number: ");
scanf("%d", &number);
printf("The number read is %d\n", number);
fflush(stdin);   // flush the input buffer
printf("Correct (y/n)?");
scanf("%c", &reply);
printf("Your reply: %c\n", reply);
```

41

Reading Mixed Data Input: Solution

1. There are a few ways to avoid this problem.
   - The first way is to modify the **scanf()** to read in the newline character (**'\n'**) from the buffer: **scanf("\n%c", &reply);**
   - The second way is to declare a character variable **dummy** and then use the **scanf()** to read the newline character (**'\n'**) into the variable **dummy** from the buffer: **scanf("%c", &dummy);**
   - The third way is to use the **fflush(stdin)** function to flush or empty the input buffer that contains the newline character before asking for reply. **fflush()** is a standard C library function. However, it is **not recommended** to use **fflush()** as we found some C compilers do not support **fflush()**. Also, **fflush()** does not work with APAS compiler.
2. Note that this kind of problem may occur when programs read in **numerical** data and **character/string** data one after another.
3. For example, in this program, it first reads in an integer number, then followed by reading in a character. If the program reads in only integer numbers throughout the program, such problem will not occur.

Reading Mixed Data Input in APAS

- As mentioned, there are three possible ways to tackle the problem on reading mixed data input, please note:
  - Try to use **scanf("\n");** to get rid of the remaining newline character in the input buffer.
  - May also try (a) **getchar()**, (b) **scanf("%c", &dummychar)** or (c) **fgets()** to solve the problem.
  - However, try **not** to use **fflush()** as APAS does not support it.
- The "**TIMEOUT**" error in APAS means:
  - The program is waiting for input but no input is received. So timeout occurs as the program waits too long for input data.
  - Therefore, make sure all inputs are provided to the program when running it in APAS.

42

Reading Mixed Data Input in APAS

1. APAS uses a different compiler from Code::Blocks. APAS uses a C compiler while Code::Blocks uses a C++ like compiler to compile C programs. Therefore, in order for your code to run in APAS, you need to make sure that your code is written in **C** syntax, not C++ syntax.
2. Printout Data - It is extremely important to pay special attention to the **program input and output format** (including letter cases) when you write your programs. A simple mistake on letter case will cause your program to be marked as incorrect in APAS). Please follow exactly the data printout format shown in the sample input/output sessions. Please remember that it is your responsibility to make sure that your program printout data must follow the data requirements given in the sample input/output sessions.
3. Reading Data Input from Mixed Data Types – When you read data input which consists of data from different data types (e.g. to read integer first, then read character or vice versa), you may need to consider the extra newline character **'\n'** in the input buffer (this will be discussed in the Chapter on Structures).
4. In APAS, you should use the statement: **scanf("\n");** to achieve it. **Try NOT** to use **fflush();** to achieve that as it will not be accepted in APAS to help get rid of the extra newline character in the input buffer.
5. For this kind of problem, you may get an error message called "**TIMEOUT**" when you use "**Run Input**" option to run the program. So,

- **Do not use** fflush(); in your program.
- May try scanf("%c", &dummychar); getchar(); fgets(); to solve the problem.

Thank You!

Thank You

1. Thanks for watching the lecture video.