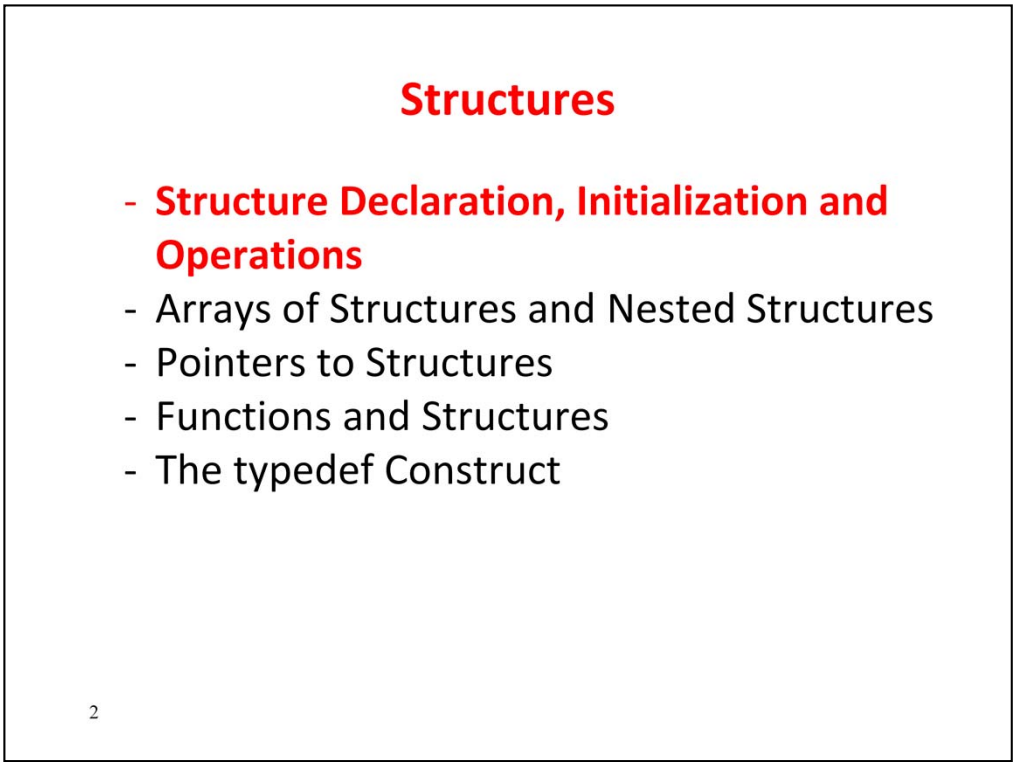
7
Structures

1

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 any valid data types.
3. In this lecture, we describe the **struct** data type.



Structures

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

Records

- Medical Records
- Employee Records
- Book Records
- Etc.

Note: Records usually contain data of **different types**.

3

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.
4. After defining a structure, we can then define a variable of that structure to store the different data together under that variable.

Structures

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

[Note: may have **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.

4

Structures

1. Structure is an aggregate of values. Their components are distinct, and may possibly have different types, including arrays and other structures.
2. To build our own data types using structures, we need to define the structure and declare variables of that type.
3. A structure template is used to specify a structure definition. It tells the compiler the various components that make up the structure.
4. Structure variables are then declared with the type of the structure.
5. For example, a book record may contain the title, author and book value. A structure can then be defined as a **data type** with the different data members. To use a structure in a program, there are two steps:
   - Define a structure template (or data type); and
   - 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.

5

Structure a Structure Template

1. A structure template (or data type) is the master plan that describes how a structure is put together. A structure template can be set up as follows:

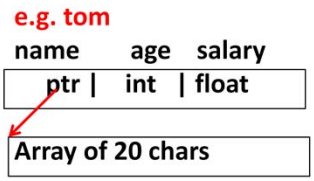
```
struct book {            /* struct book defines the template of book*/
    char title[40];      /* title, author, value are members of the structure */
    char author[20];
    float value;
};                       /* semicolon to end the definition */
```

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.

6

Declaring Structure Variable: with Tag Name

1. The structure name or tag is optional. With structure tag, the definition of structure template can be separated from the definition of structure variables. In the following declaration, a structure template **person** comprising three components **name**, **age** and **salary** is created.

```
struct person {                 /* with tag name */
    char name[20];
    int age;
    float salary;
};
struct person tom, mary;        /* structure variables */
```

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

/* no tag – person is not used */

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

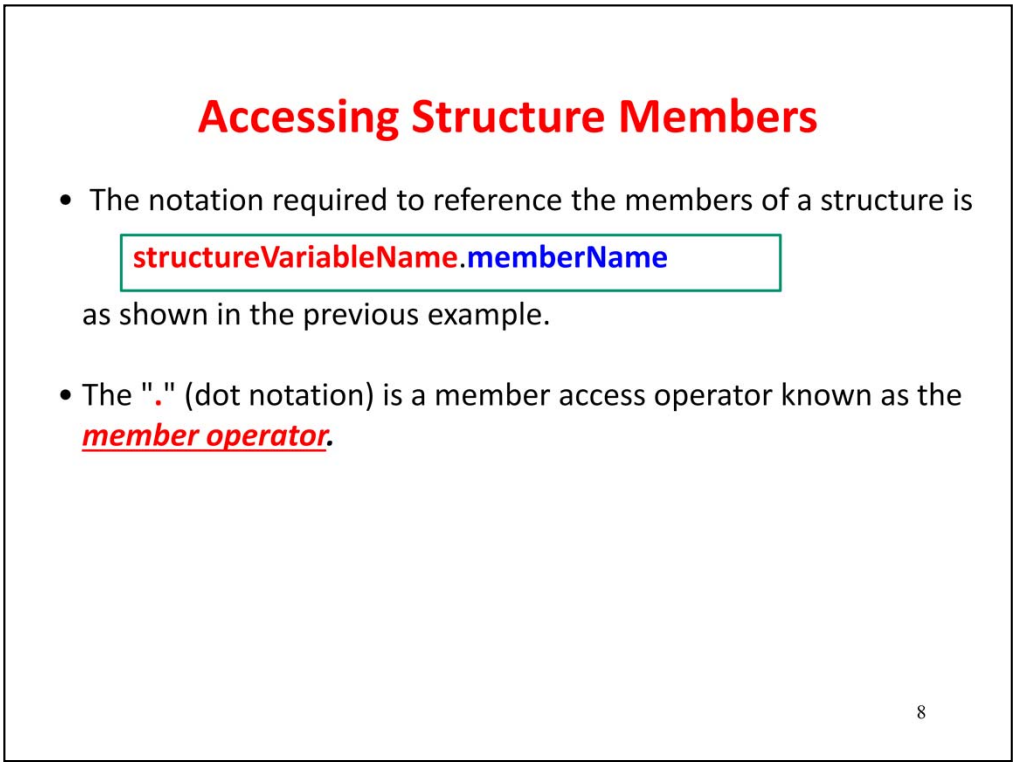
7

Declaring Structure Variable: without Tag Name

1. Without structure tag, the definition of structure template must be combined with that of structure variables.
2. In the declaration, a structure template is created with three components: name, age and salary.

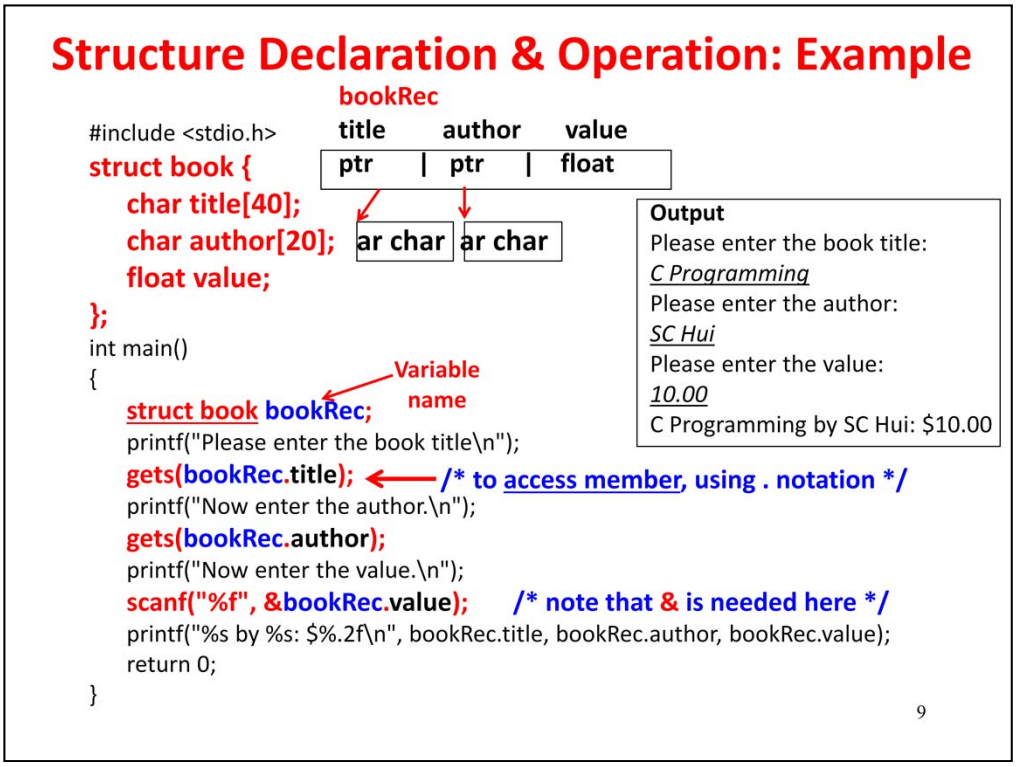
```
struct {                    /* no tag name */
    char name[20];
    int age;
    float salary;
} tom, mary;                /* structure variables */
```

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. It is always a good idea to include a structure tag when defining a structure.



Accessing Structure Members

1. The notation required to access a member of a structure is **structureVariableName.memberName**
2. For example, to access the member **id** of the variable **student**, we use **student.id**
3. 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.



Structure Declration and Operation: Example

1. In the program, it defines the structure template (or data type) and the declaration of a structure variable.
2. After defining the structure template **struct book** outside the **main()** function, the declaration **struct book bookRec;** declares a variable **bookRec** of type **struct book**. It also allocates storage for the variable.
3. 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. In the program, the definition is defined at the beginning of the file, it is a global declaration, and all the functions following the definition can use the template.
4. In the program, the following statements will read the user input on title and author which are character strings:

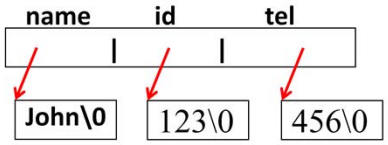
   gets(bookRec.title);

   gets(bookRec.author)
5. To access a member of a structure, we use the dot notation such as **bookRec.title** and **bookRec.author**.
6. The statement **scanf("%f", &bookRec.value);** will read the user input on book value which is of data type **float**.
7. After reading the user input, book title, author and book value will be printed.

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 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);
```



using . notation

Output
John 123 456

10

Structure Variable: Initialization

1. The syntax for initializing structures is similar to that of initializing arrays.
2. When there are insufficient values to be assigned to all members of the structure, the remaining members are assigned to zero by default.
3. The structure variable is followed by an assignment symbol and a list of values defined within braces:

```
struct personTag {
    char    name[40];
    char    id[20];
    char    tel[20];
} student = {"John", "123", "456"};      /* with initialization */
```

4. 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"**.
5. The following statement prints the data of the structure variable **student**:

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

6. Note that the dot notation is used to access the member of structure.

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;
```

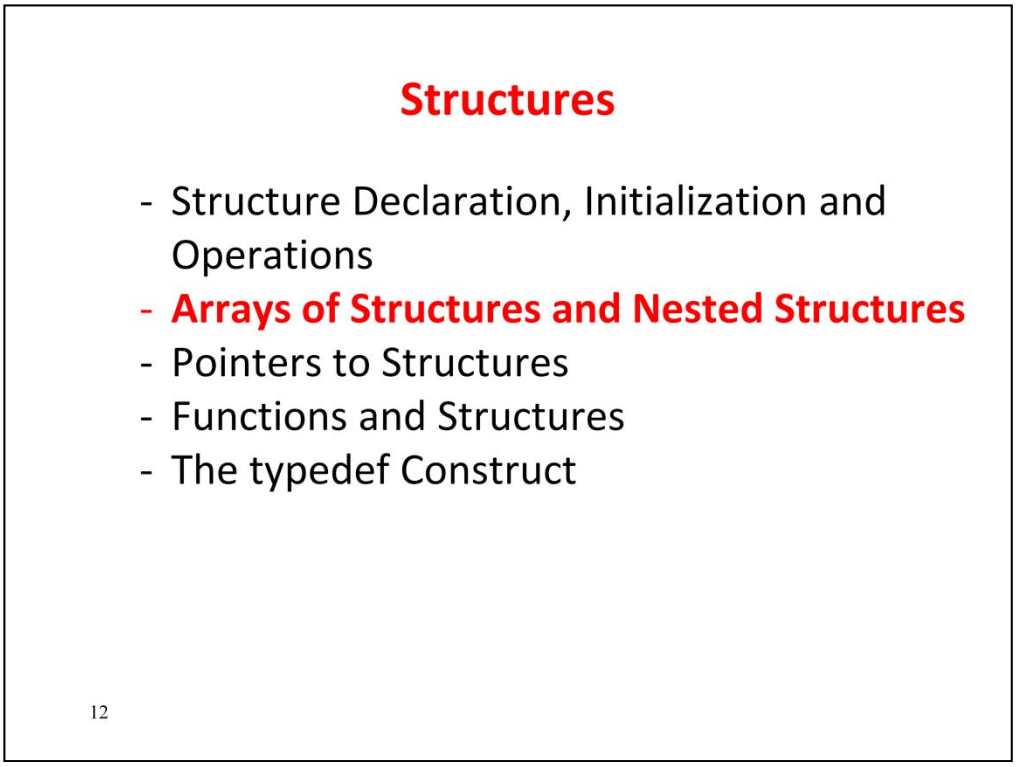
11

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**:

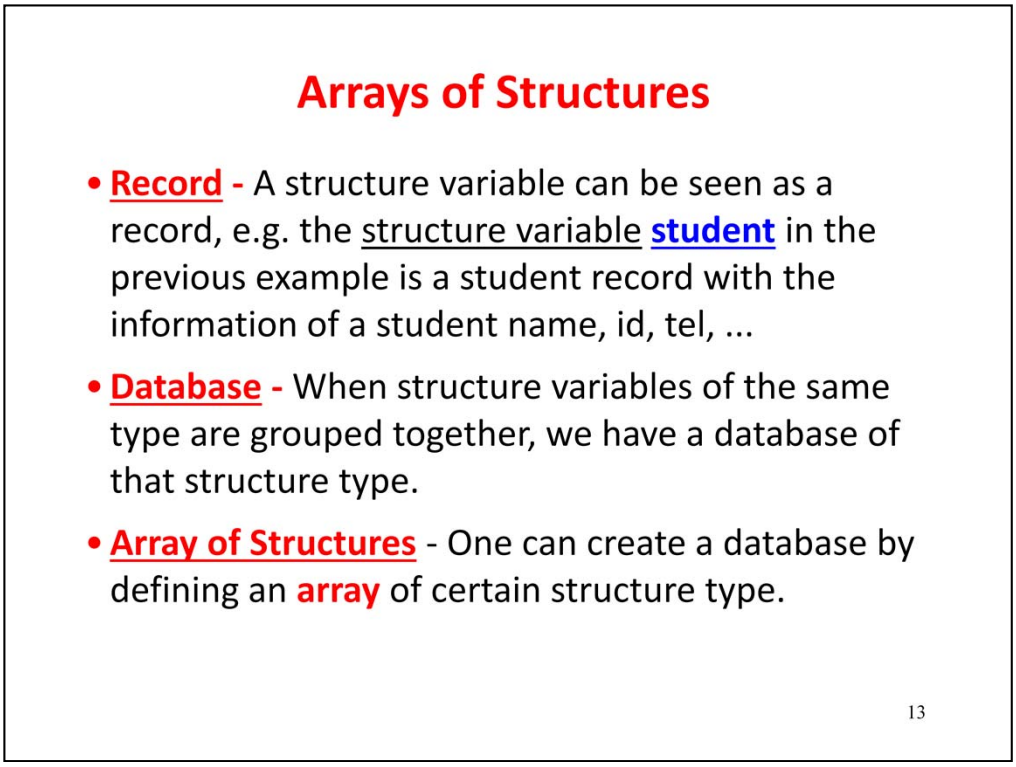
   struct personTag newmember;
3. Then, we can assign the **struct personTag** variable **student** to **newmember** as follows:

   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



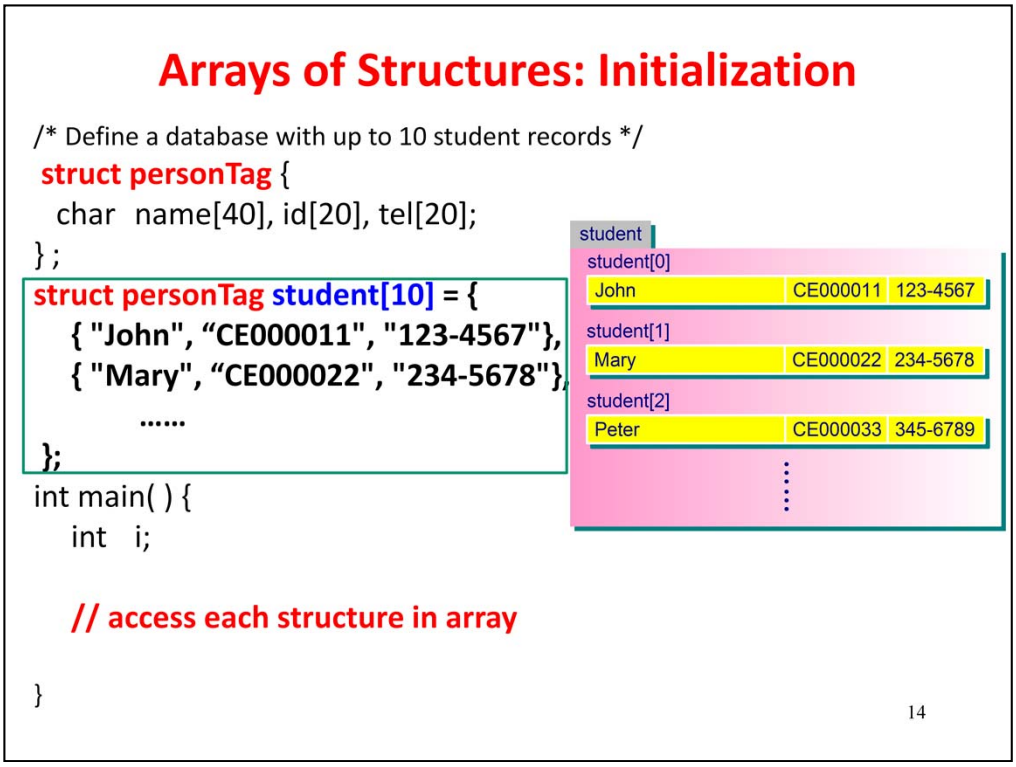
Structures

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



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.



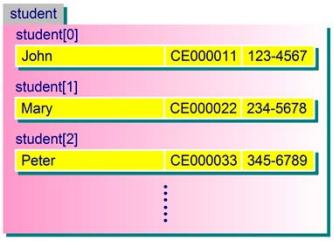
Arrays of Structures: Initialization

1. In the program, the variable **student** defines an array of structures, which is a database of student records. 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.
2. 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.
3. Array of structures can be initialized. 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"},    /* initialize values for student[0] */
    {"Mary", "CE000022", "234-5678"},    /* initialize values for student[1] */
    {"Peter", "CE000033", "345-6789"},   /* initialize values 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

15

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:

```
for (i=0; i<10; i++)
     printf("Name: %s,  ID: %s, Tel: %s\n",
          student[i].name, student[i].id, student[i].tel);
```

5. Note that 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.

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) such as:

```
struct studentTag { // without any nested structures
    char name[40];
    char id[20];                    (1) Student information
    char tel[20];
    int   SC101Yr;  /* the year when SC101 is taken * /
    int   SC101Sr;  /* the semester when SC101 is taken */     (2) Course: SC101
    char SC101Grade;  /* the grade obtained for SC101 */
    int   SC102Yr;  /* the year when SC102 is taken */
    int   SC102Sr;  /* the semester when SC102 is taken */     (3) Course: SC102
    char SC102Grade;  /* the grade obtained for SC102 */
};
 struct studentTag student[1000];
  // student – array of 1000 student records
```

16

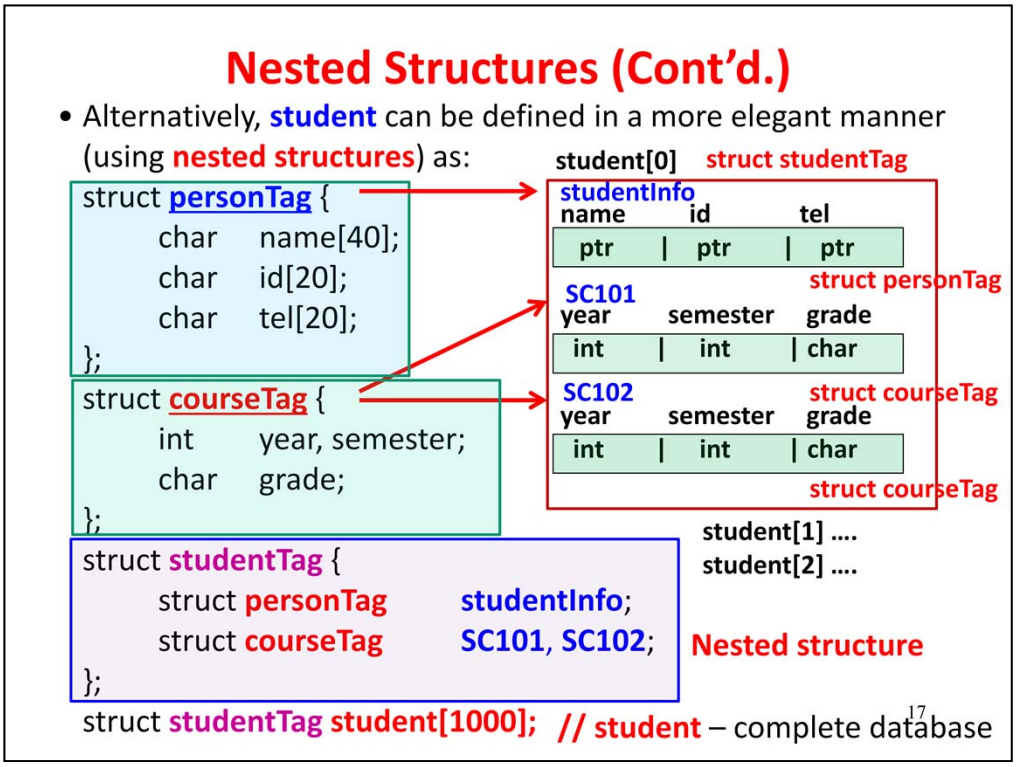
Nested Structures

1. A structure can also be included in other structures. This is called nested structures. For example, to keep track of the course history of a student, one can use a structure (without any nested structures) as follows:

```
struct studentTag {
    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];
```

2. In the structure template definition **struct studentTag**, the members are the student information including **name**, **id** and **tel**. In addition, it also includes the

courses that are taken by the student. Once the **struct studentTag** is defined, an array variable **student** of 1000 elements of type **struct studentTag** is created.



Nested Structures

1. Alternatively, the variable **student** can be defined in a more elegant manner using nested structures.
2. As we can observe that the members of the structure **studentTag** can be further grouped together to form other structures to make it more concise, we define the nested structure **studentTag** as follows:

```
struct studentTag {
    struct personTag    studentInfo;
    struct courseTag    SC101, SC102;
};
```

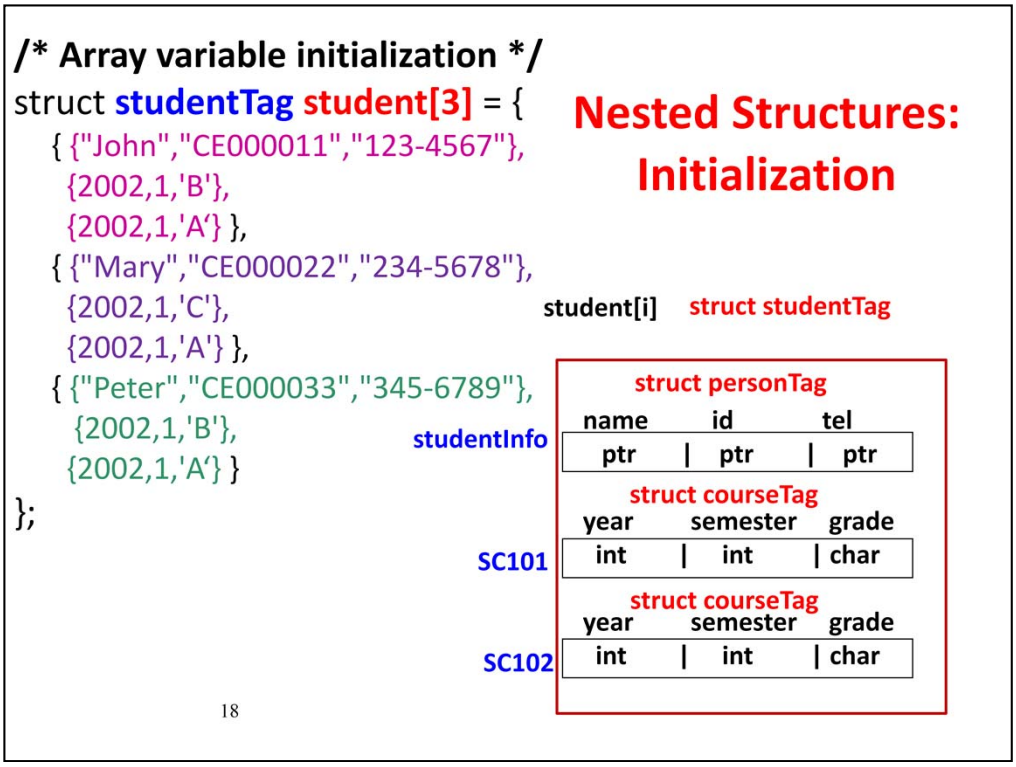
3. The structure **studentTag** has three members.
   - **studentInfo** which is a structure of **personTag**;
   - **SC101** and **SC102** which are structures of **courseTag**.
4. Then, we create a structure template called **personTag** to contain the student information. It has three members, namely **name**, **id** and **tel**, of the array data type.

```
struct personTag {
    char    name[40];
    char    id[20];
    char    tel[20];
};
```

5. We also create a structure template called **courseTag** to contain the course information as follows:

```
struct courseTag {
    int   year, semester;
    char grade;
};
```

6. The structure **courseTag** has three members, namely **year** and **semester** of type **int**, and **grade** of type **char**.
7. Note that the structure definition of **personTag** and **courseTag** must appear before the definition of structure **studentTag**.



Nested Structures: Initialization

1. In this program, after defining the nested structure **studentTag** and the array of structures variable **student**, we initialize the variable **student** 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*/
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);
}
```

E.g. Array of Structures:

```
#include  <stdio.h>
struct personTag {
  char name[40], id[20], tel[20];
} ;
int main( ) {
 struct personTag student[10] = {
{ "John", "CE000011", "123-4567"},
{ "Mary", "CE000022", "234-5678"},
      ……
};
 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);
}
```

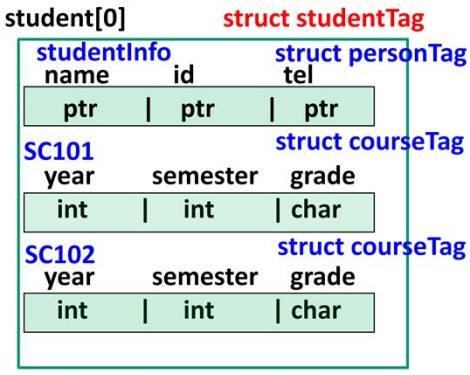
- Using dot (member operator) to access members of structures

Nested Structures: Example

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

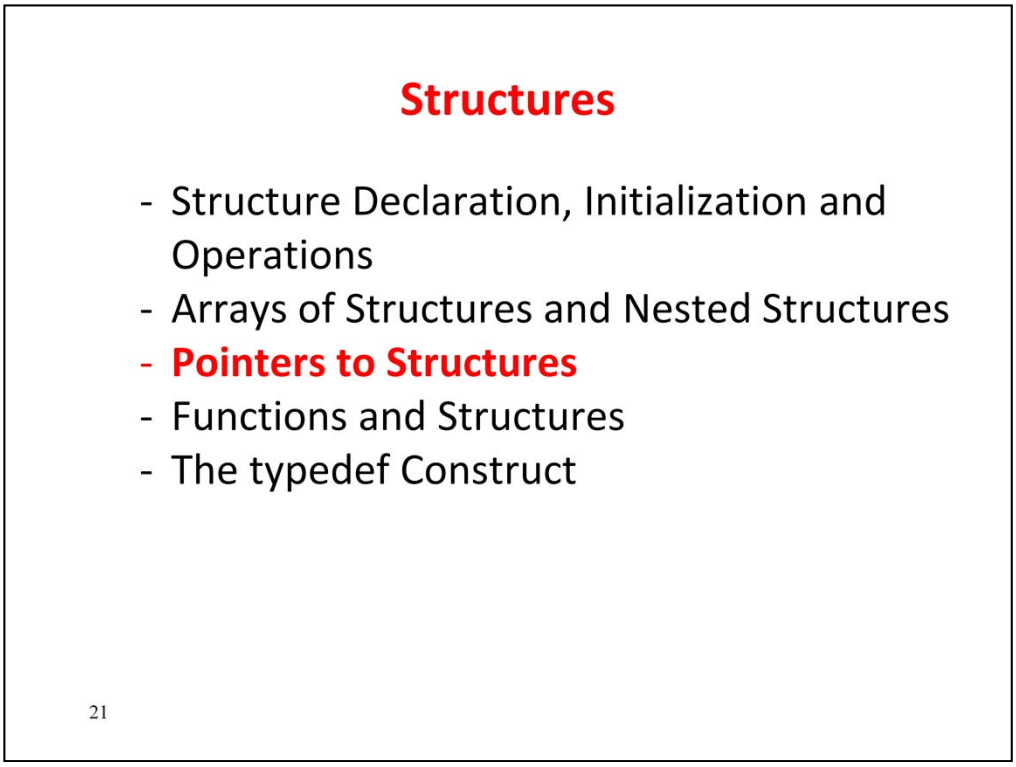
- **student[i]** denotes the *i+1th* array record. It consists of three members: studentInfo, SC101, SC102.
- **student[i].studentInfo** denotes the personal information in the *i+1th* record. It conssits of three members: name, id, tel.
- **student[i].studentInfo.name** denotes the student name in this 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.



20

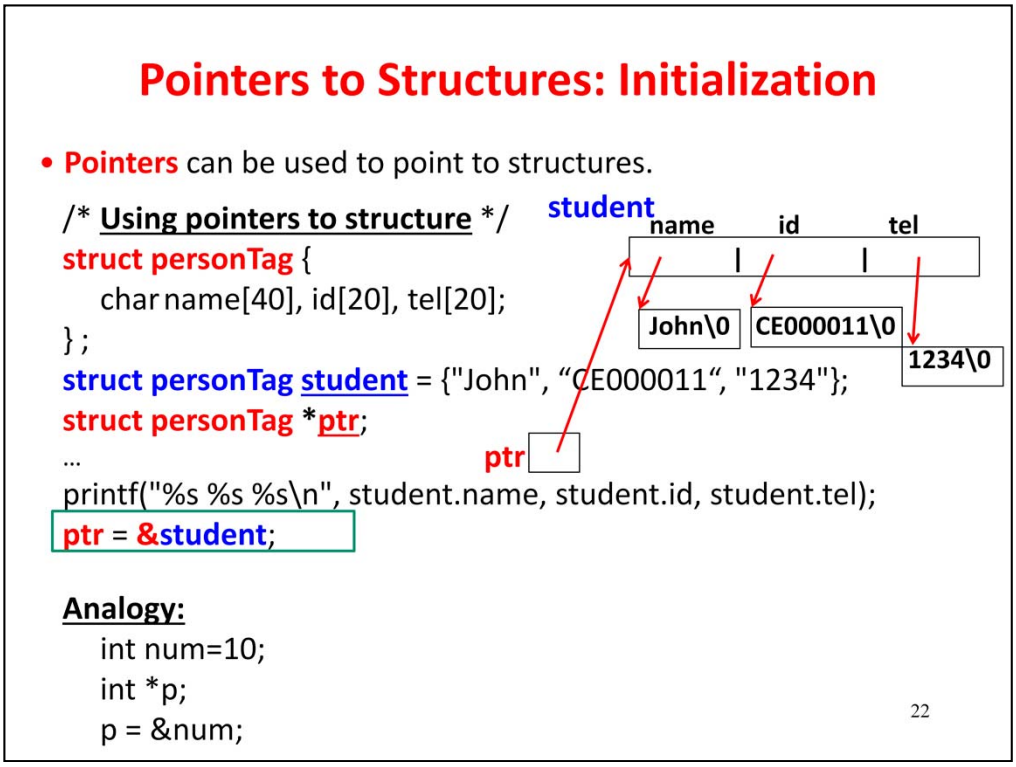
Nested Structures: Notations

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

1. Here, we discuss pointers to structures.



Pointers to Structures: Initialization

1. Pointers can be used to point to structures. The declarations

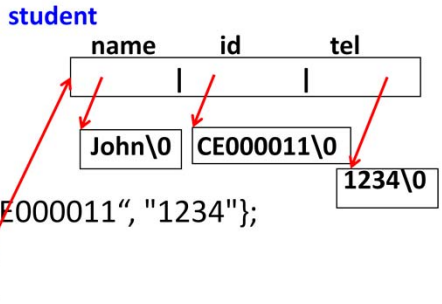
```
struct personTag {
    char name[40], id[20], tel[20];
};
struct personTag student={"John","CE011","1234"};
```

   define a structure template **personTag** and variable **student** with initialization.
2. Then, we create a pointer **ptr** to the structure **personTag**: **struct personTag *ptr;**
3. To initialize a pointer, we must use the address operator (**&**) to obtain the address of a structure variable, and then assign the address to the pointer as

```
ptr = &student;
```

4. The address of the structure variable **student** is assigned to the pointer variable **ptr**.
5. Then, 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 */
```

23

Pointers to Structures: Operation

1. The **indirection operator** (*****) can be used to access a member of a structure via a pointer to the structure. 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.
2. Note that the parentheses are necessary to enclose ***ptr** as the member operator (**.**) has higher precedence than the indirection operator (*****).

Pointers to Structures: Operation (Cont'd.)

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

Or it can also be written as:

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

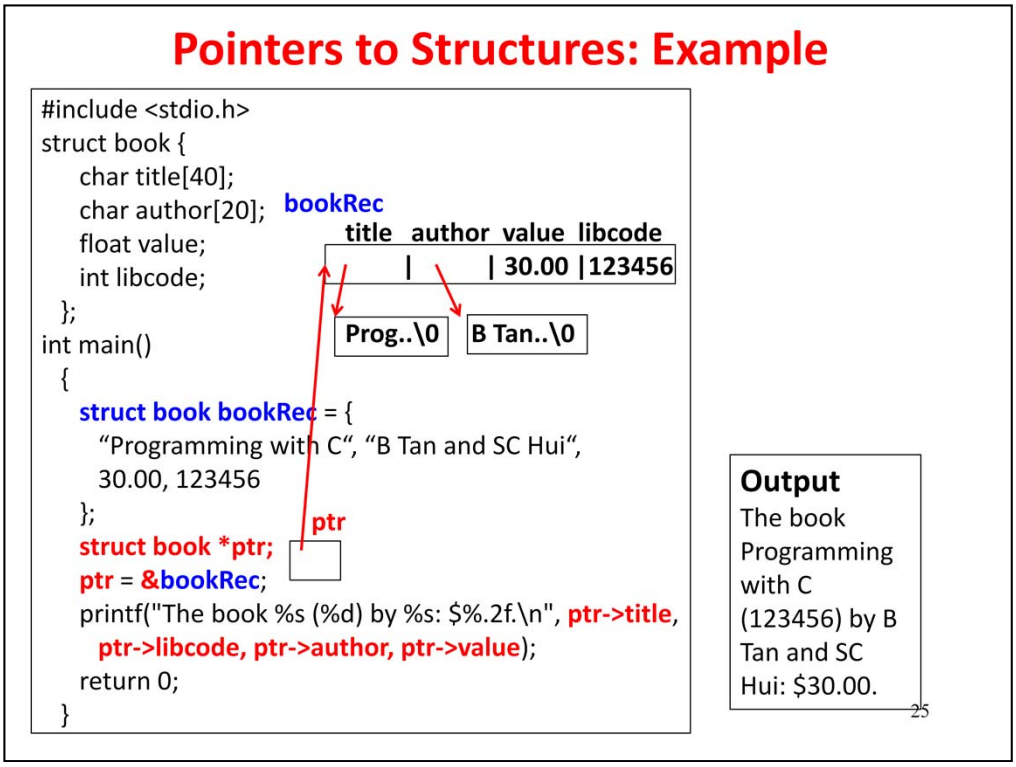
Note

- The operator **->** is called the **structure pointer operator** reserved for a pointer pointing to a structure.
- Less typing is needed if one compares **ptr->tel** to **(*ptr).tel**.
- It is quite common to use the structure pointer operator (**->**) instead of the indirection operator (*****) in pointers to structures.

24

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**. It takes less typing when **ptr->tel** is compared with **(*ptr).tel**, though they have exactly the same meaning. It is quite common to use the structure pointer operator (**->**) instead of the indirection operator (*****) in pointers to structures.



Pointers to Structures: Example

1.We can then use the structure variable to access each member of the structure. We can also use pointer variable to access each member of the structure.

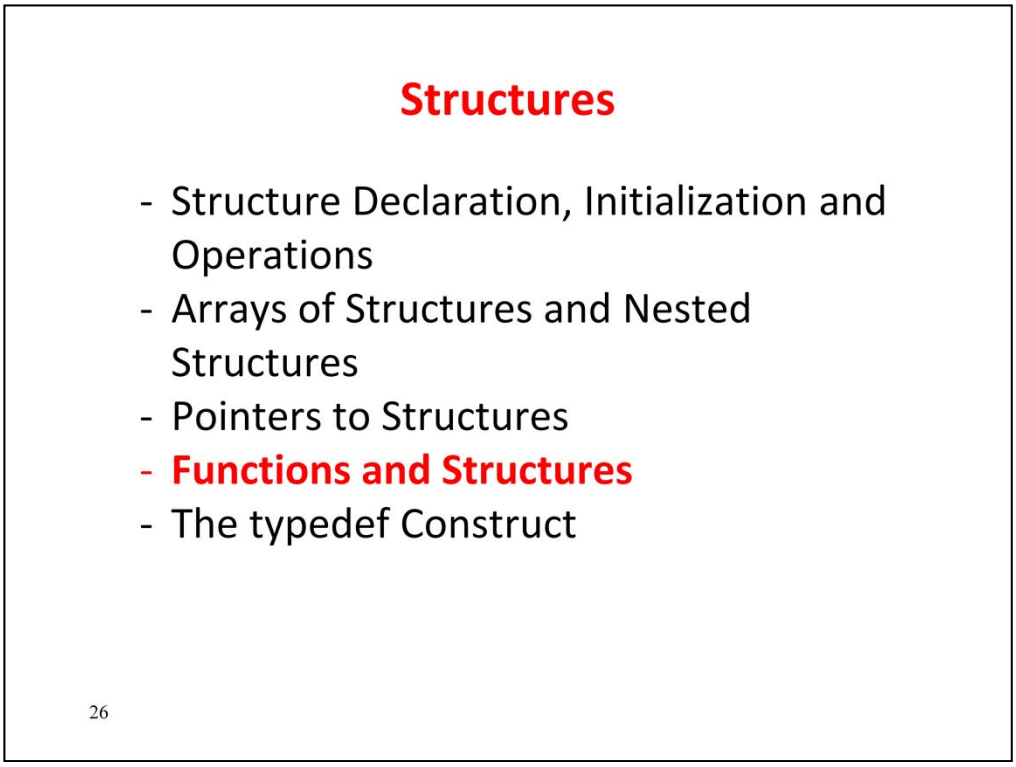
2.In the program, we define a structure called **book** with four members: **title**, **author**, **value** and **libcode**. After that, we define a structure variable called **bookRec**, and initialize it with values.

3.We then define the pointer variable **ptr** to the **struct book** type: **struct book *ptr;**

4.We assign the address of the structure variable **bookRec** to the pointer variable **ptr**: **ptr = &bookRec;**

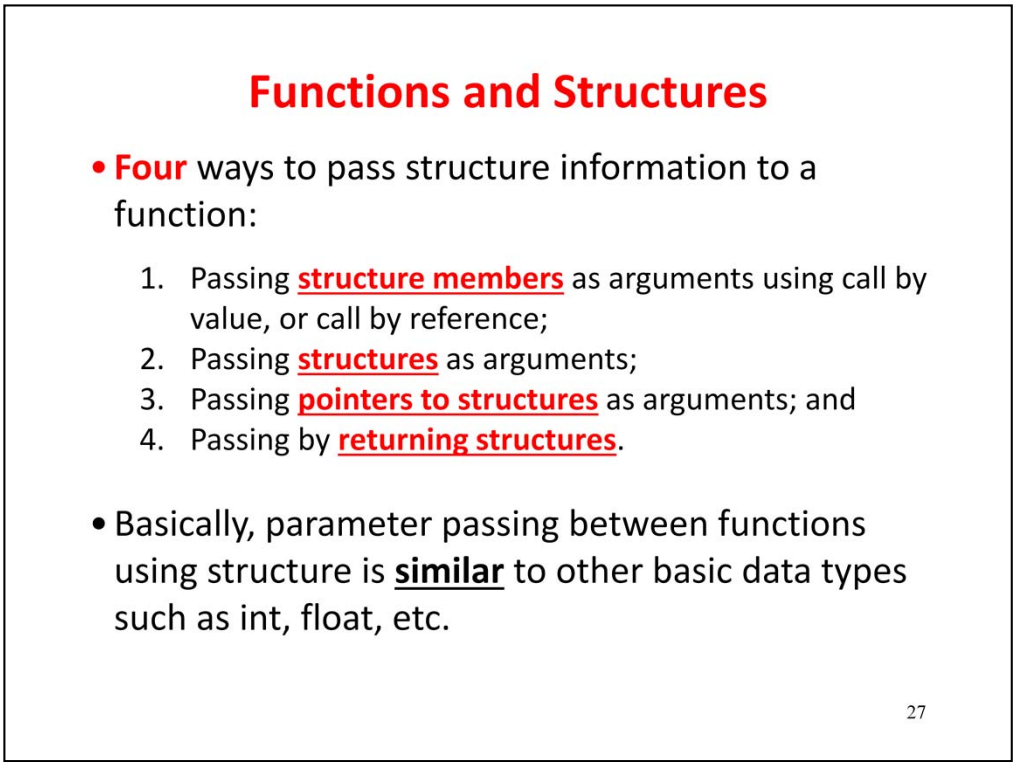
5.Therefore, the pointer variable contains the address of **bookRec**. As a result, we may access the members of **bookRec** via **ptr**.

6.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**.



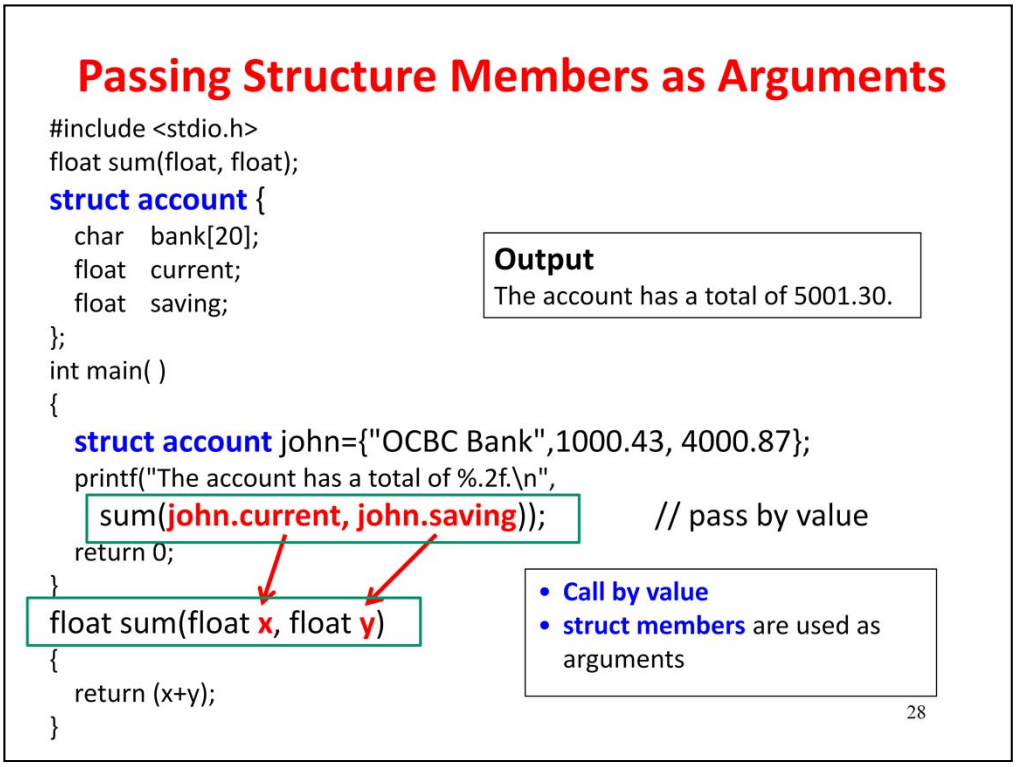
Structures

1. Here, we discuss functions and structures.



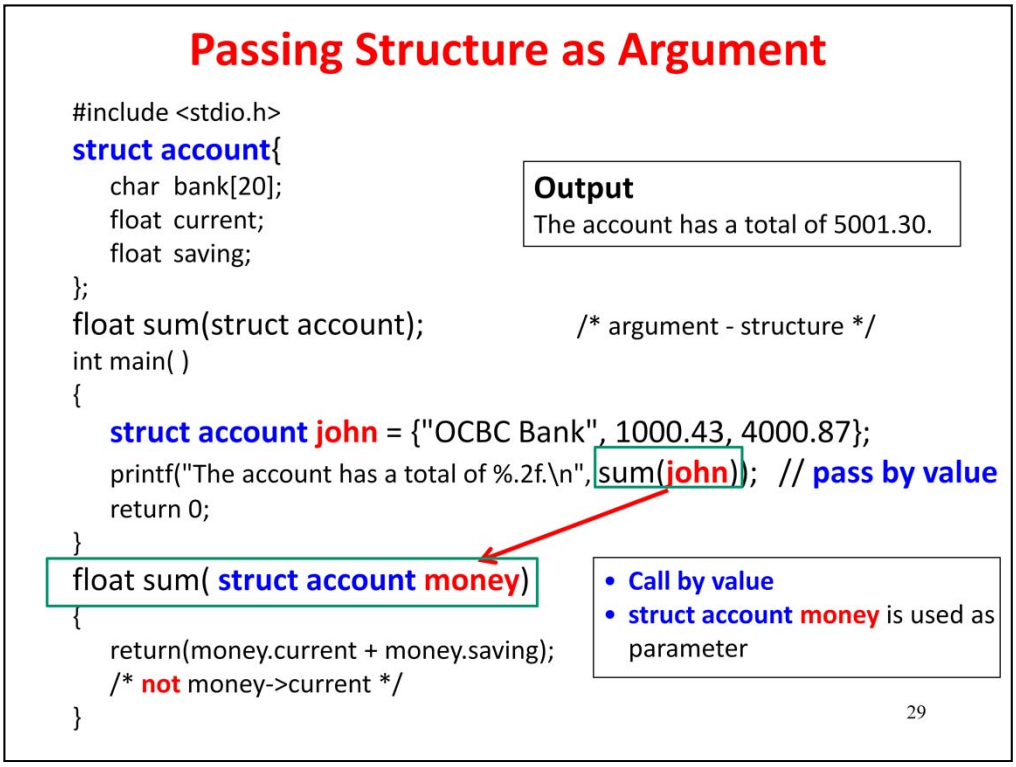
Functions and Structures

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.



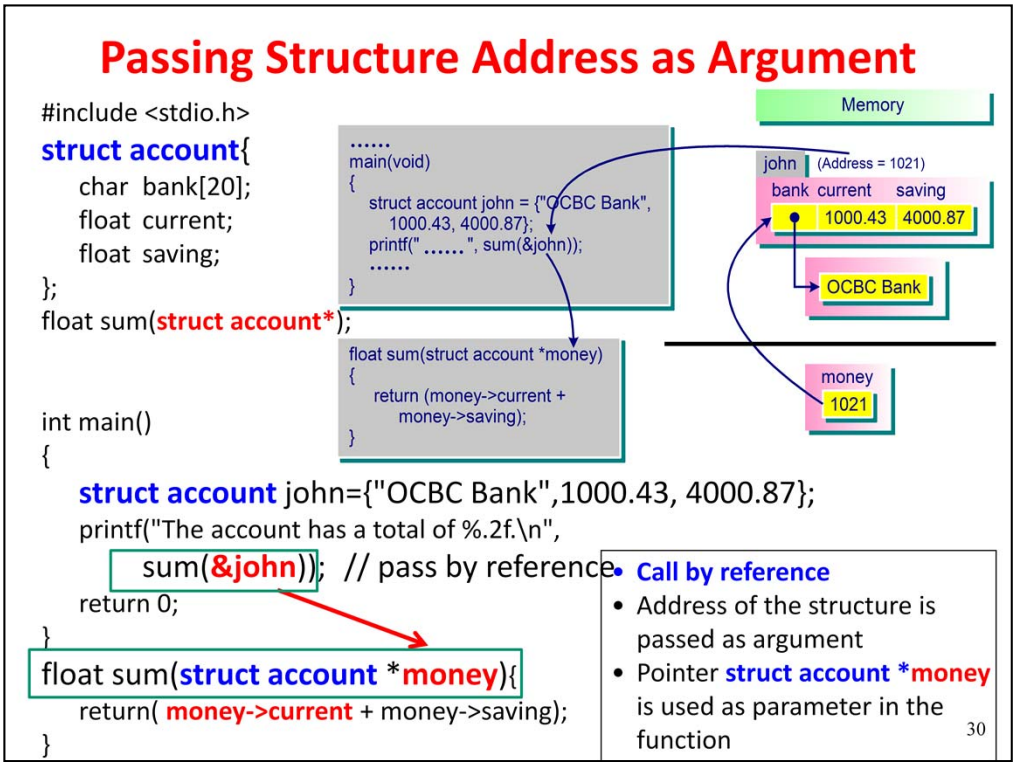
Passing Structure Members as Arguments

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 function **sum()** is used to compute the total amount from the saving and current accounts.
3. There are different ways to implement the function **sum()**.
4. The first approach is to pass individual members of a structure as arguments to a function.
5. In the program, the function **sum()** expects two arguments, **x** and **y**, of type **float**. The structure variable **john** is declared with **struct account** and the values of the members **current** and **saving** are passed to the function **sum()**. The structure members **john.current** and **john.saving** are of type **float**. As long as a structure member is a variable of a data type with a single value, we can pass the structure member as a function argument. The structure members **john.current** and **john.saving** are passed by value to the parameters **x** and **y** respectively.



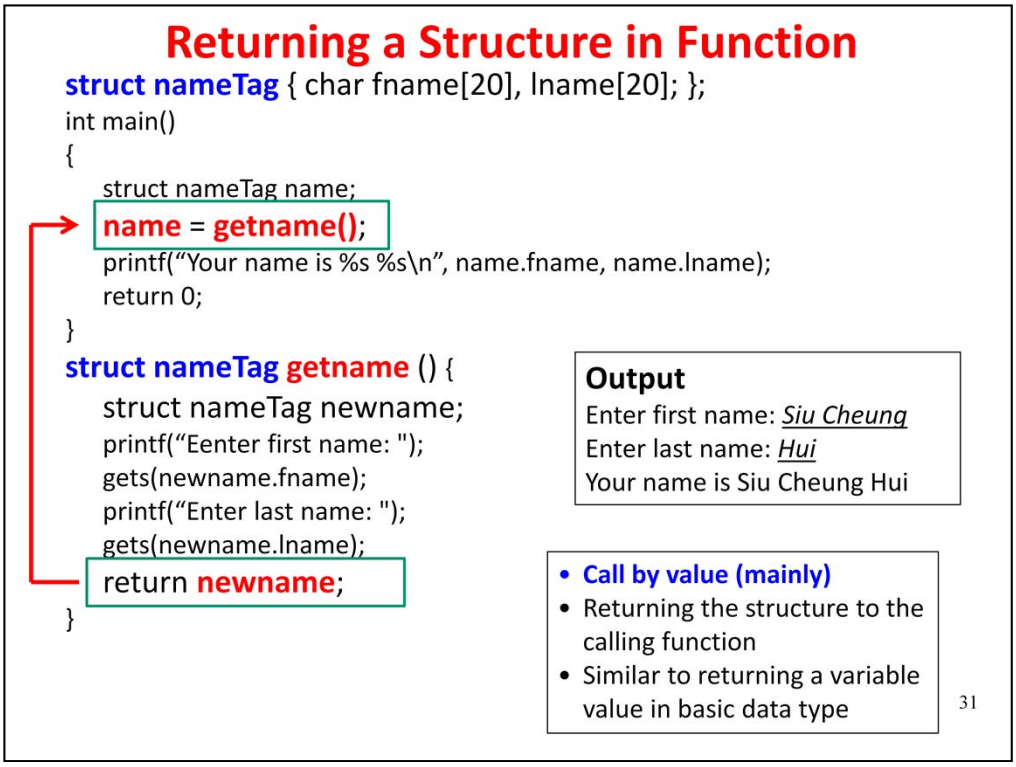
Passing Structures as Arguments

1. The second approach is to pass a structure to a function as an argument to a function. It uses the call by value method.
2. When a structure is passed as an argument to a function, it is passed using call by value. The members of this structure in the function **sum()** are initialized with local copies. The function can only modify the local copies. Notice that we simply use the member operator (**.**) to access the individual members of the structure variable as follows: **return(money.current + money.saving);**
3. 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.
4. 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.



Passing Structure Address

1. The third approach is to pass the address of the structure as an argument. It uses call by reference method.
2. Using the same structure template **account**, in the program, the **sum()** function uses a pointer to a structure account as its argument. The address of **john** is passed to the function that causes the pointer **money** to point to the structure **john**. The **->** operator is then used in the following statement: **return(money->current + money->saving);** to obtain the values of **john.current** and **john.saving**. This allows the function to access the structure variable and to modify its content.
3. This is a better approach than passing structures as arguments.

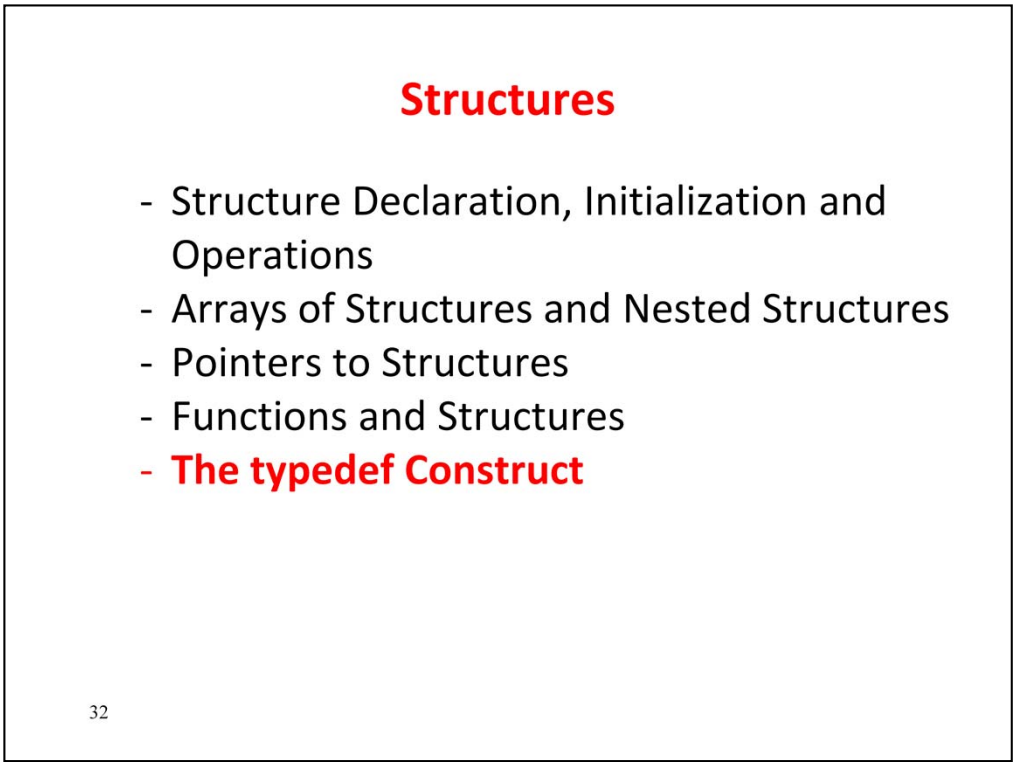


Passing by Returning Structures

1. The fourth approach is to return the structure in the function.
2. The function **struct nameTag getname()** returns a structure **nameTag**. To call this function, the calling function must declare a variable of type **struct nameTag** in order to receive the result from **getname()** as follows:

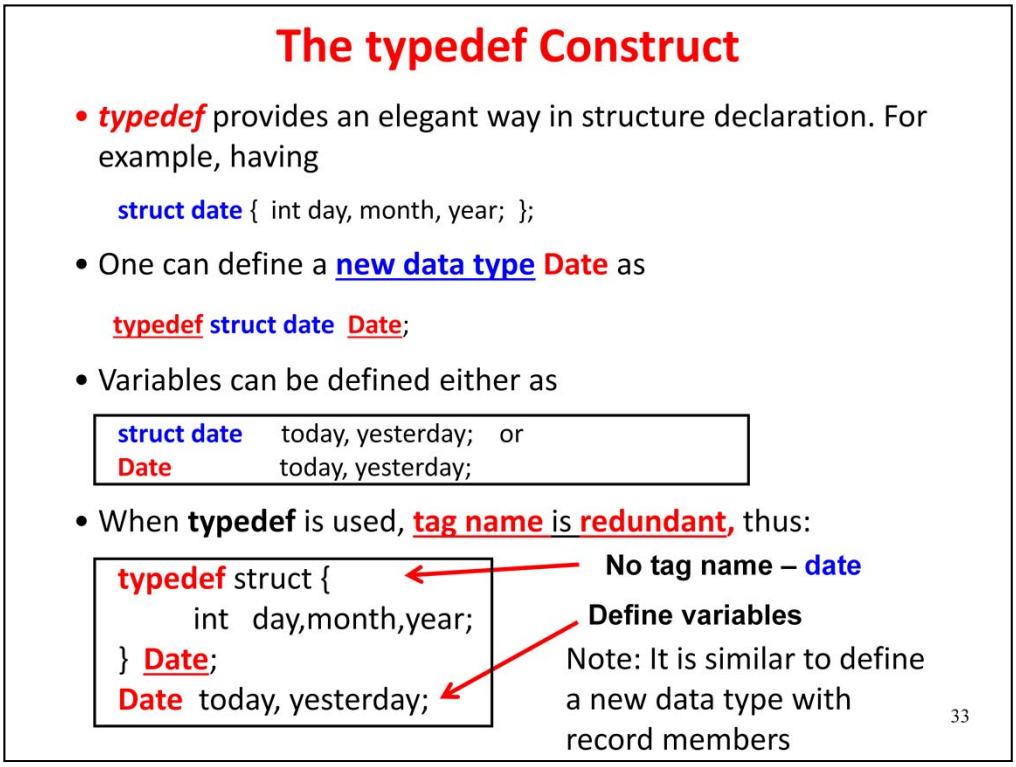
   struct nameTag name;

   name = getname();
3. The statement assigns the returned structure data to the variable **name**.



Structures

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



The typedef Construct

1. **typedef** provides an elegant way in structure declaration. The general syntax for the **typedef** statement is

   typedef dataType UserProvidedName;

2. The **typedef** keyword is followed by the data type and the user provided name for the data type. It is very useful for creating simple names for complex structures. For example, if we have defined the structure:

   struct date {

   int day, month, year;

   };

   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

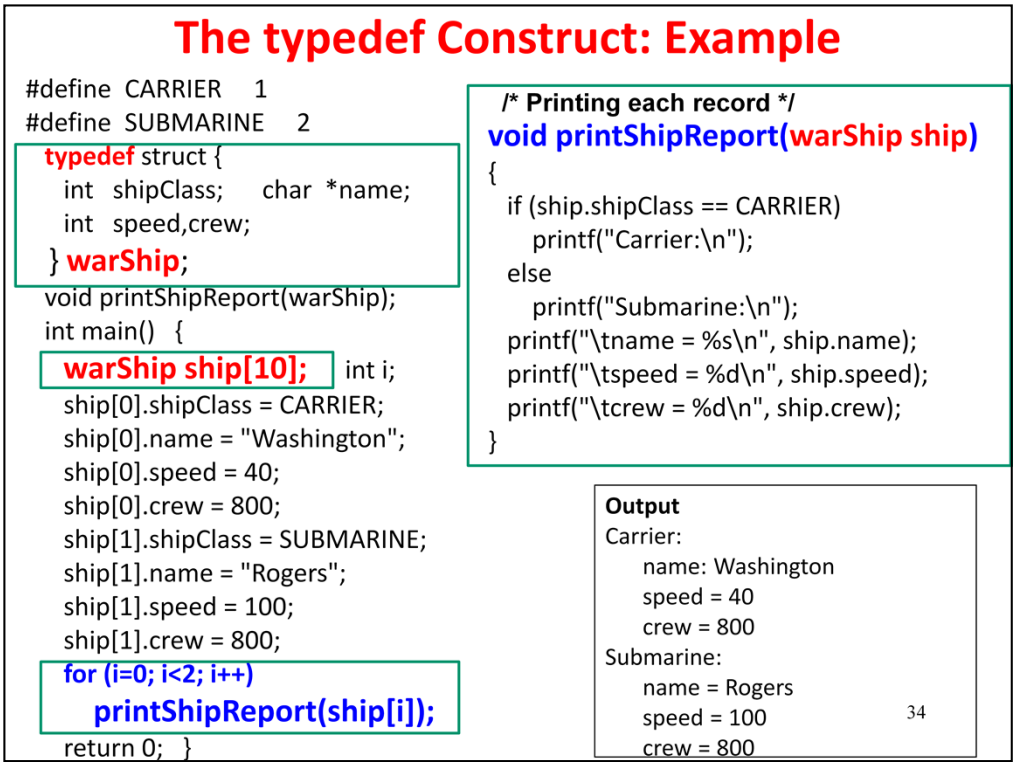
   Date today, yesterday;

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.

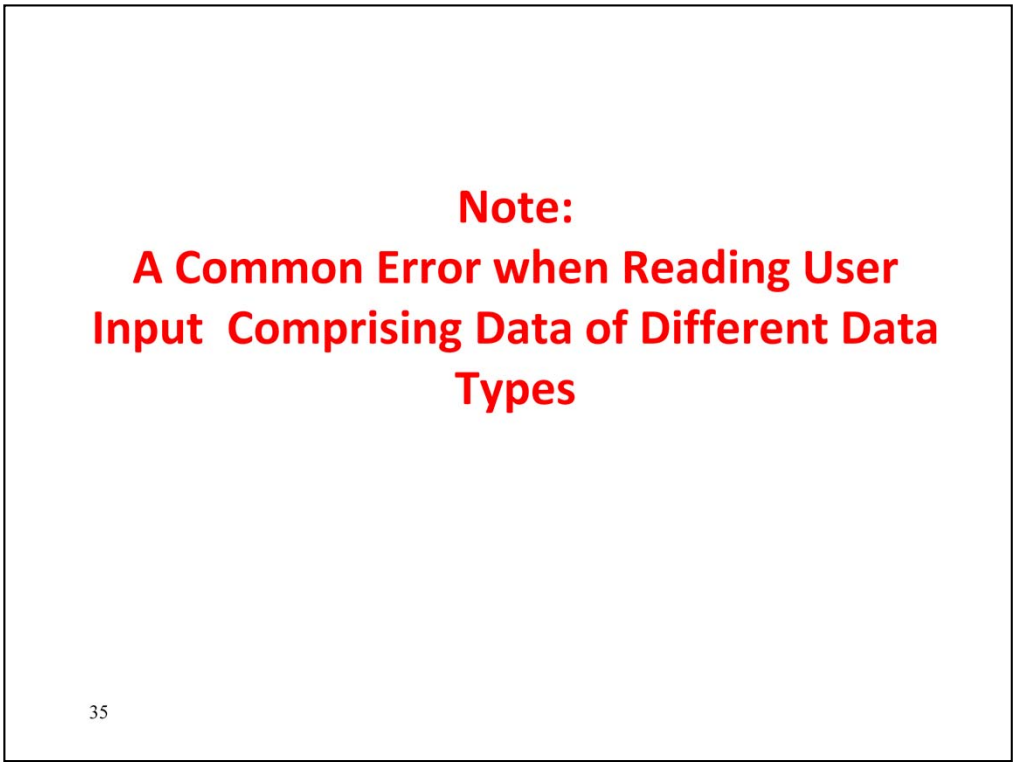


The typedef Construct: Example

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

```
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**. The function **printShipReport()** is used for printing the member information of the **warShip** structure. In the **main()** function, a **for** loop is used to print the member information of the **ship** variable using the **printShipReport()** function.



Common Errors in Reading User Input

1. Here, we discuss an example on a common error when reading user inputs.

Common Error: scanf()

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

      printf("Please 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 Output:

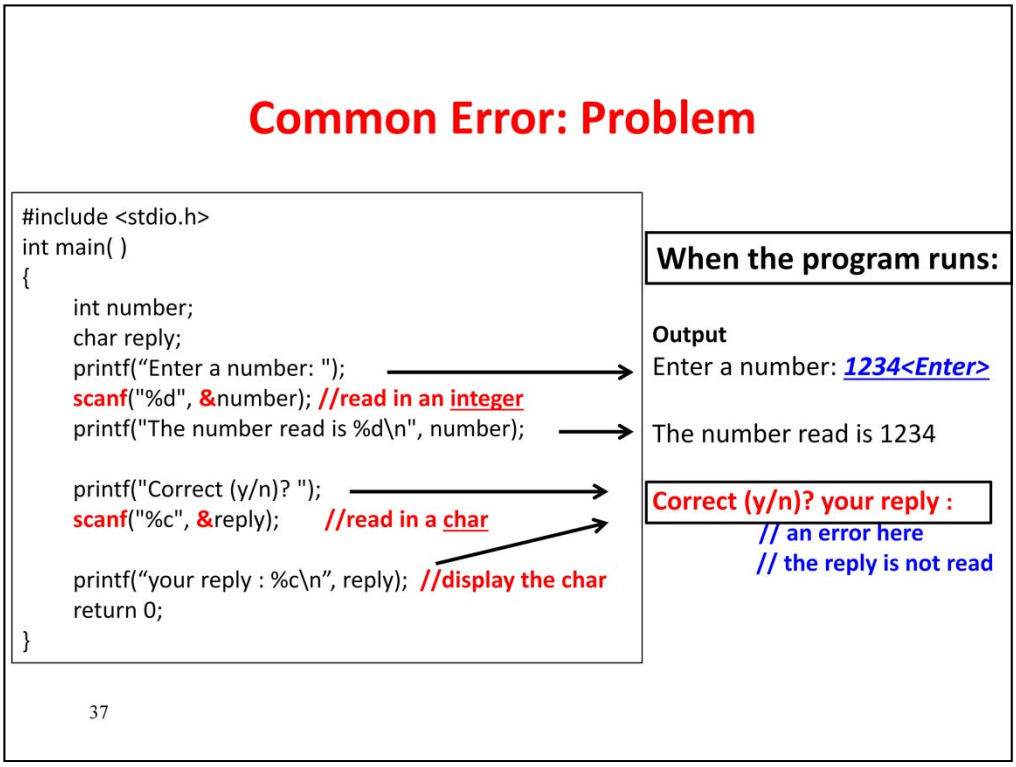
Please enter a number: *1234<Enter>*
The number read is 1234
Correct (y/n)? *y*
your reply : y

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

36

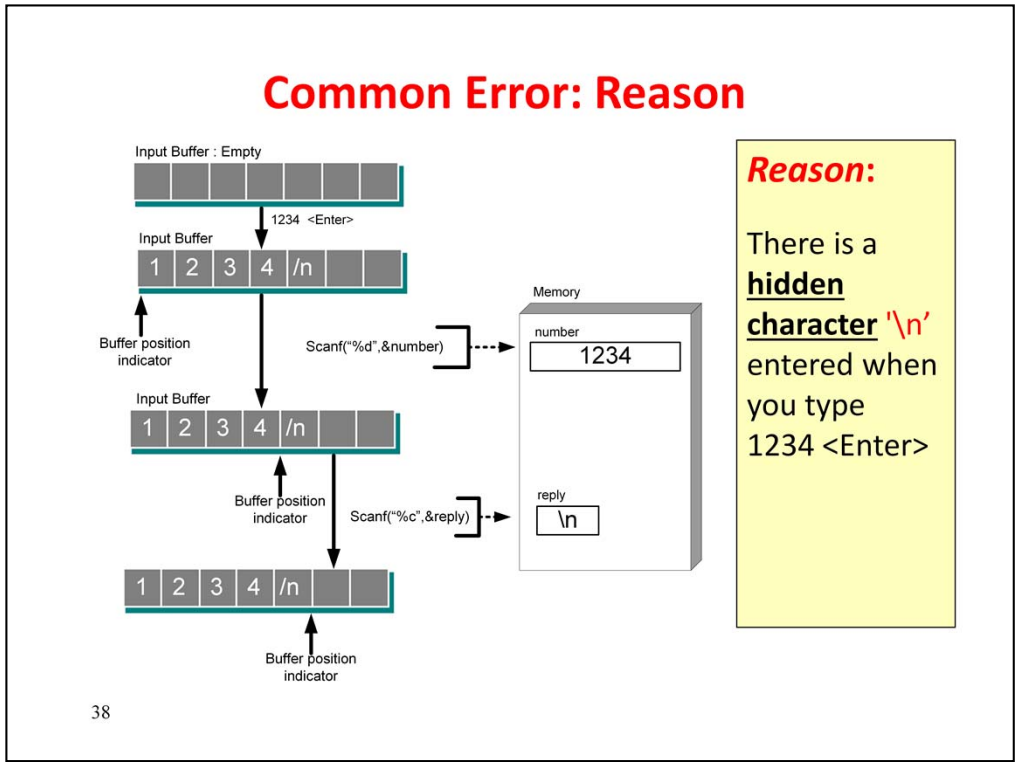
Common Error: scanf()

1. In the program, it implements the input operation using the **scanf()** statement.
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** (yes or no) using **scanf()**, and will be stored in the variable **reply**.
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 run as intended?



Common Error: 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 by the user.
4. As such, the program does not run correctly as intended. It means that the program does not read in user reply on **y** or **n**.



Common Error: Reason

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

Common Error: Suggested Solutions

- **Solution 1: read in '\n'**

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

OR (suggested approach)

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

- **Solution 2: using fflush() (Not Recommended)**

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

39

Common Error: Suggested Solutions

1. There are two ways to avoid this problem:
2. The first way is to modify the **scanf()** to read in a newline from the buffer as follows:

   scanf("\n%c", &reply);
3. Or we can declare a character variable **dummy** and then use the **scanf()** to read in the newline from the buffer as follows:

   scanf("%c", &dummy);
4. The second way is to place the **fflush(stdin)** function before **scanf()** as follows:

   fflush(stdin);

   scanf("%c", &reply);
5. As such, it will 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 suggested** to use **fflush()** as we found some C compilers do not support **fflush()**.
6. Note that this problem may occur when programs read in **numerical** data and **character/string** data one after another. For example, in this example, the program 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.

Thank you !!!

40