7
Structures

1

Review

- What have you learnt so far:
  - Basic Sequential Programming (data, operators, simple I/O)
  - Branching (if-else, switch, conditional operator)
  - Looping (for, while, do…while)
  - Functions (function definition and call by value)
  - Pointers (call by reference for function communication)
  - 1-D Arrays (+ using pointers for 1D arrays)
  - 2-D (Multi-dimensional) Arrays (+ using pointers for 2D arrays)
  - Character Strings (array of characters with null character)
- Arrays are list of data of the same data type.
- Character strings are arrays of characters.
- If you need to process data of different types as a record (structure), C provides you with Structures.
- In this lecture, we discuss about … **Structures**….

2

This Lecture

- At the end of this lecture, you will be able to understand the following:
  - Structure Declaration, Initialization and Operations
  - Arrays of Structures
  - Nested Structures
  - Pointers to Structures
  - Functions and Structures
  - The typedef Construct

3

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

Structures

- **Structure Declaration, Initialization and Operations**
- Arrays of Structures
- Nested Structures
- Pointers to Structures
- Functions and Structures
- The typedef Construct

4

Records

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

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

5

Records

Records are used to keep related information of an object together. There are many examples of records such as 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. 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 to use a structure:
  1. Setting up a structure template (similar to a data type);
  2. Defining a variable on the structure template.

6

Structures

Structure is an aggregate of values. Their components are distinct, and may possibly have different types, including arrays and other structures. To build our own data types using structures, we need to define the structure and declare variables of that type. A structure template is used to specify a structure definition. It tells the compiler the various components that make up the structure. Structure variables are then declared with the type of the structure.

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:

1.Set up a structure template (or data type);

2.Define 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.

7

Structure Template (Data Type)

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

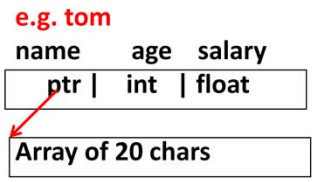
struct is a reserved keyword to introduce a structure. **book** is an optional tag name that follows the keyword **struct** to name the structure declared. The **title**, **author** and **value** are the *members* of the structure **book**. The members of a structure can be any of the valid C data types. A semicolon after the closing brace ends the definition of the structure definition.

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.

8

Structure Variable: with Tag Name

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 is created.

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

tom and **mary** are two structure variables which are declared using the structure **person**.

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.

9

Structure Variable: without Tag Name

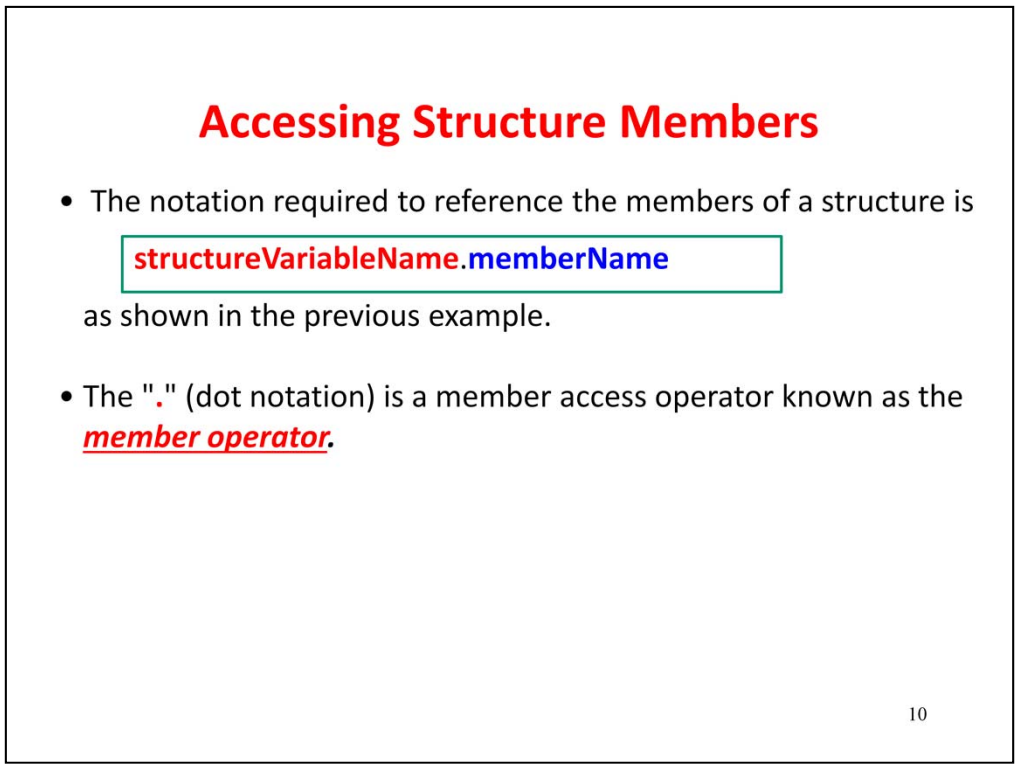
Without structure tag, the definition of structure template must be combined with that of structure variables.

In the following declaration, a structure template is created with the three components:

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

The variables **tom** and **mary** are then defined using this structure.

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

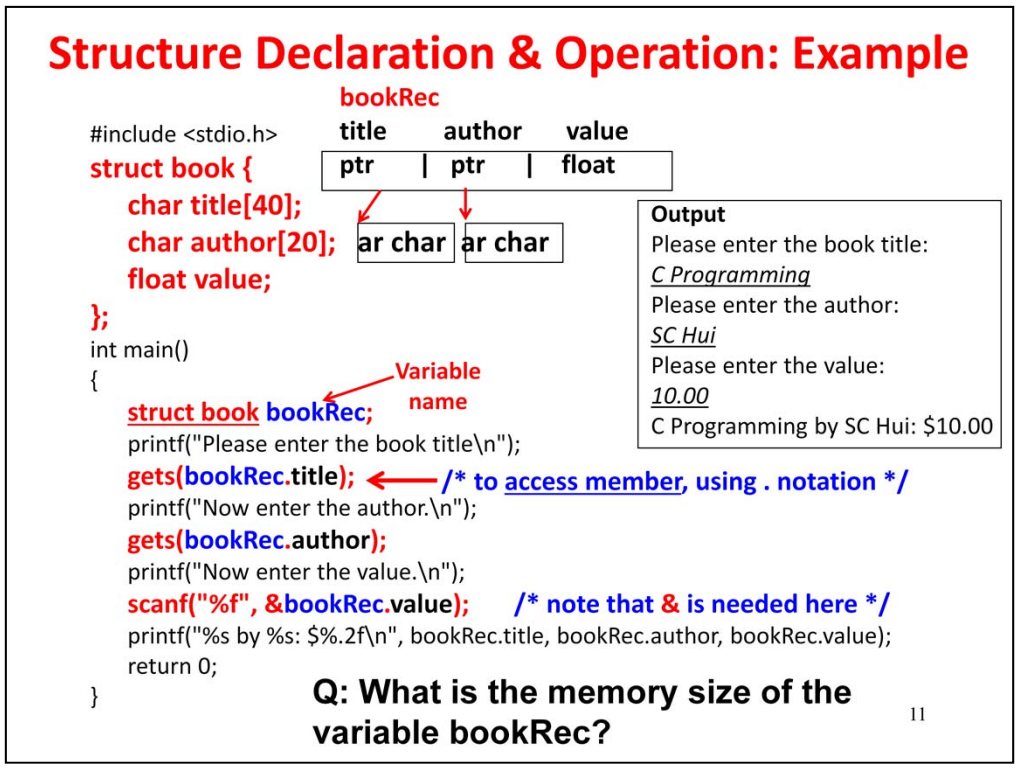
The notation required to access a member of a structure is

structureVariableName.memberName

For example, to access the member **id** of the variable **student**, we use

student.id

The "**.**" is an access operator known as the *member operator*. The member operator has the highest (or equal) priority among the operators.



Structure Declration and Operation: Example

In the program, it defines the structure template (data type) and the declaration of a structure variable.

After defining the structure template **struct book** outside the **main()** function, the following declaration

struct book bookRec;

declares a variable **bookRec** of type **struct book**. It also allocates storage for the variable.

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.

In the program, the statements

gets(bookRec.title);

gets(bookRec.author)

will 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 **bookRec.title** and **bookRec.author**.

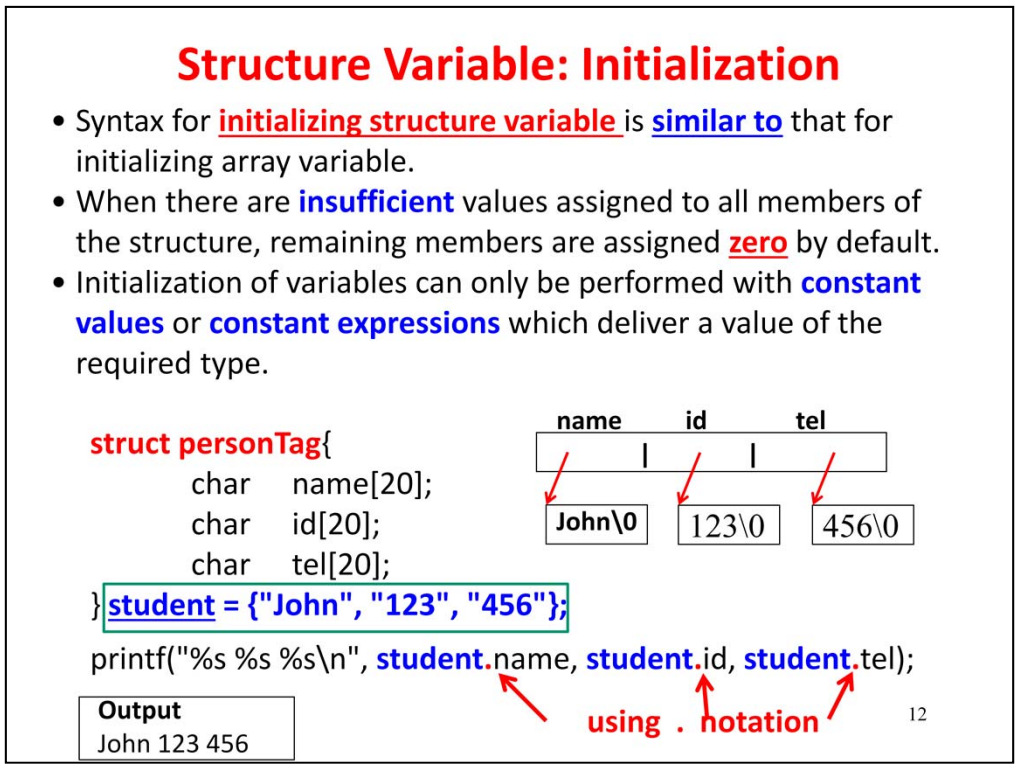
The statement

scanf("%f", &bookRec.value);

will read the user input on book value which is of data type **float**.

After reading the user input, the book title, author and book value will then be printed on the screen.

Question: In the structure definition **struct book**, what is the memory size of the variable **bookRec** after declaring the variable under the data type **struct book**? Hint: you may use the **sizeof** operator to print the size of the variable.



Structure Variable Initialization and Operation

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. The structure variable is followed by an assignment symbol and a list of values defined within braces:

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

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"**.

To print the data of the structure variable **student**, the statement

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

and the dot notation is used in order to access the member of the 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;
```

13

Structure Assignment

The value of one structure variable can be assigned to another structure variable of the same type using the assignment operator. First, we define a new variable **newmember** under the data type **struct personTag**:

```
struct personTag newmember;
```

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

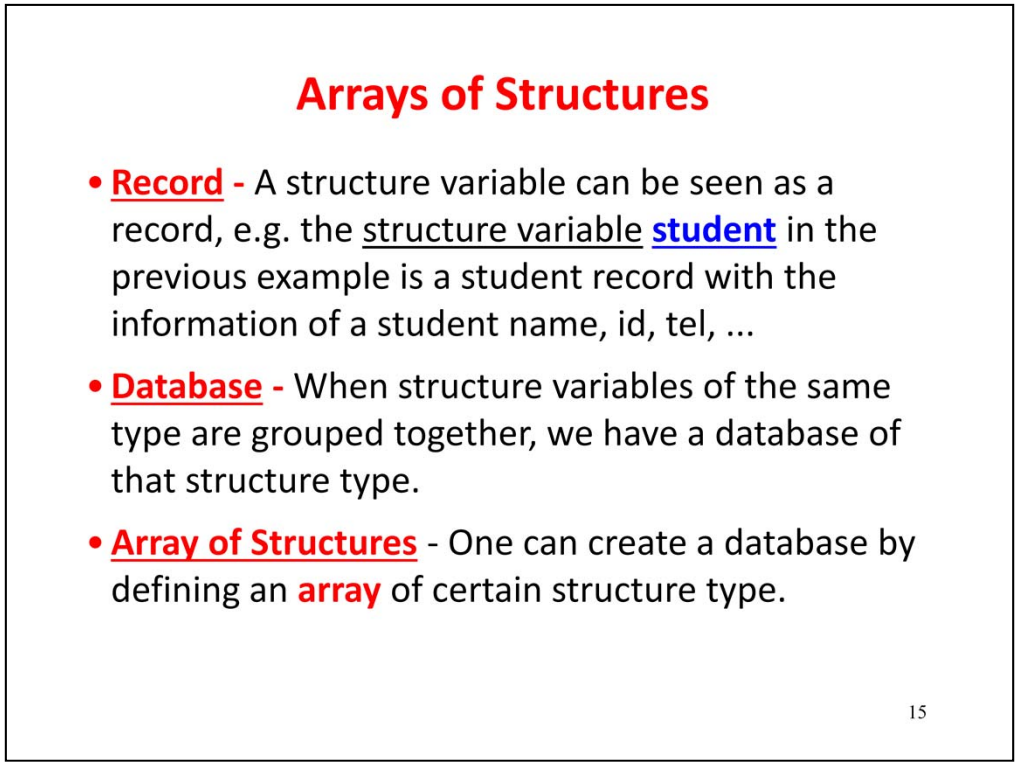
```
newmember = student;
```

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.

Structures

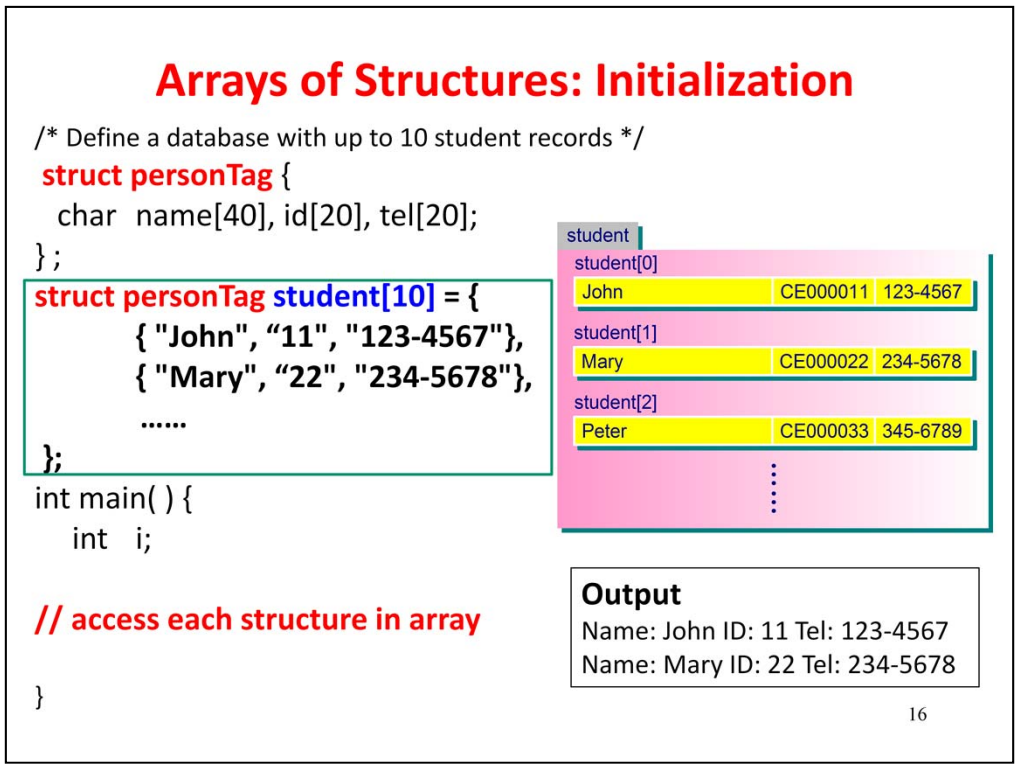
- Structure Declaration, Initialization and Operations
- **Arrays of Structures**
- Nested Structures
- Pointers to Structures
- Functions and Structures
- The typedef Construct

14



Arrays of Structures

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. When structure variables of the same type are grouped together, we can form a database of that structure type. Therefore, we can create a database by defining an array of structures.



Arrays of Structures: Initialization

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 **tel**, of the structure.

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.

Array index is used when accessing individual elements of an array of structures.

We use

student[i]

to denote the $(i+1)^{th}$ record. The first element starts with index 0.

To access a member of a specific element, we use

student[i].name

which denotes a member of the $(i+1)^{th}$ record.

We use

student[i].name[j]

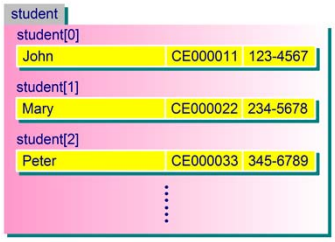
to denote a single character value of a member of the $(i+1)^{th}$ record.

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", "11", "123-4567"},
          { "Mary", "22", "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: 11 Tel: 123-4567
Name: Mary ID: 22 Tel: 234-5678

using array index and . operator

17

Arrays of Structures: Operation

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);
```

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.

Structures

- Structure Declaration, Initialization and Operations
- Arrays of Structures
- **Nested Structures**
- Pointers to Structures
- Functions and Structures
- The typedef Construct

18

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 – a complete database of student records
```

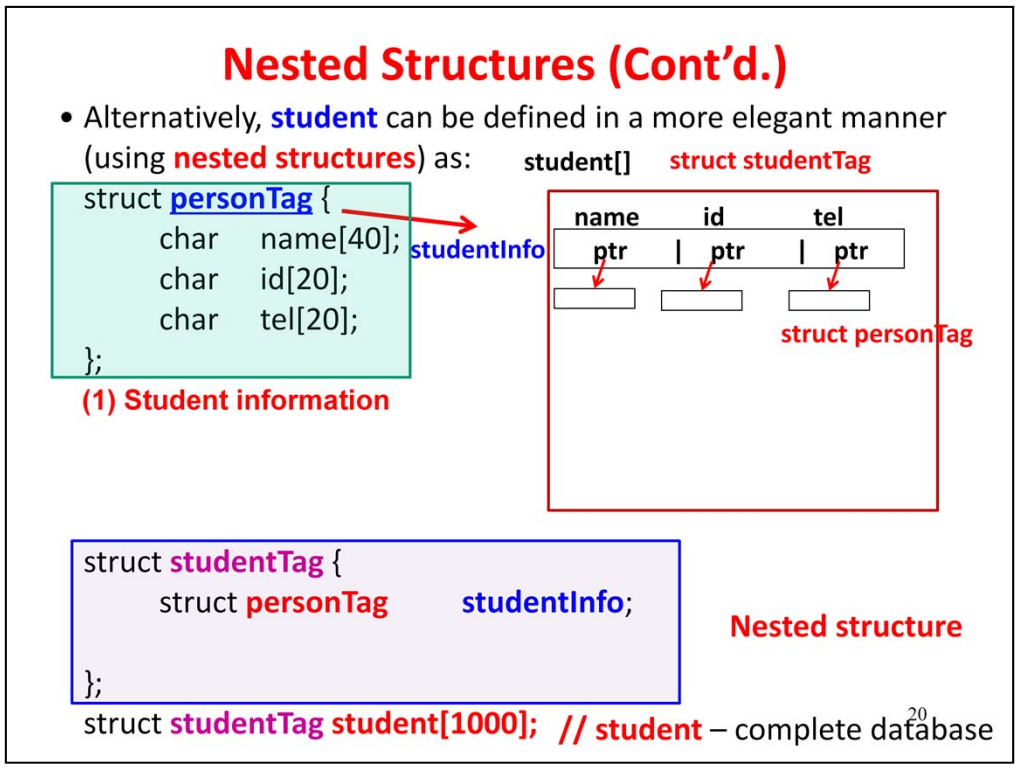
19

Nested Structures

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];
```

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

Alternatively, the variable **student** can be defined in a more elegant manner using nested structures.

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:

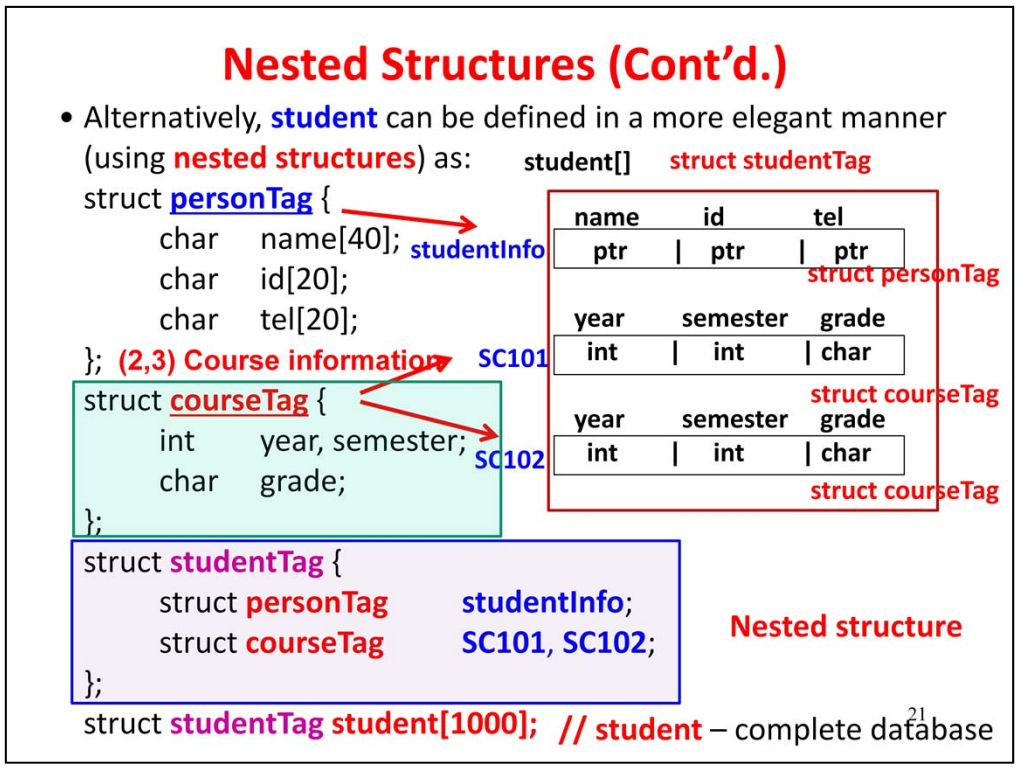
```
struct studentTag {
    struct personTag    studentInfo;
    struct courseTag    SC101, SC102;
};
```

The structure **studentTag** has three members.

- **studentInfo** which is a structure of **personTag**;
- **SC101** and **SC102** which are structures of **courseTag**.

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];
};
```



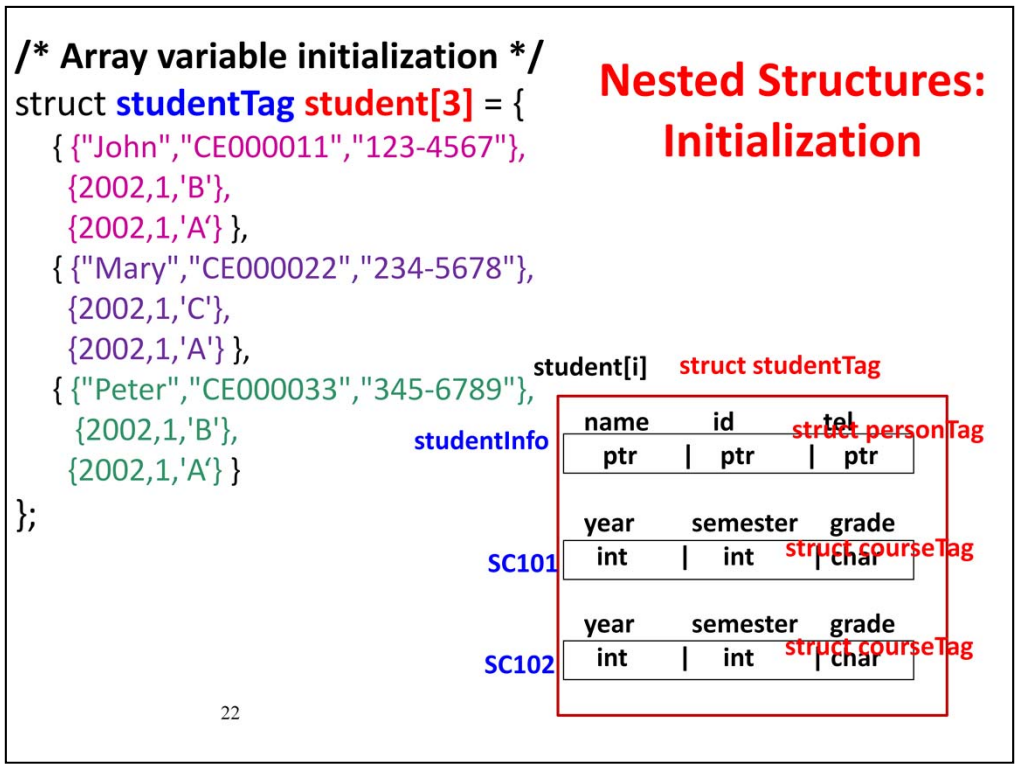
Nested Structures

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

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

The structure **courseTag** has three members, namely **year** and **semester** of type **int**, and **grade** of type **char**.

Note that the structure definition of **personTag** and **courseTag** must appear before the definition of structure **studentTag**.



Nested Structures: Initialization

In this program, after defining the nested structure **studentTag** and the array of structures variable **student**, we initialize the variable **student** with initial data.

```
/* 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);
   }
```

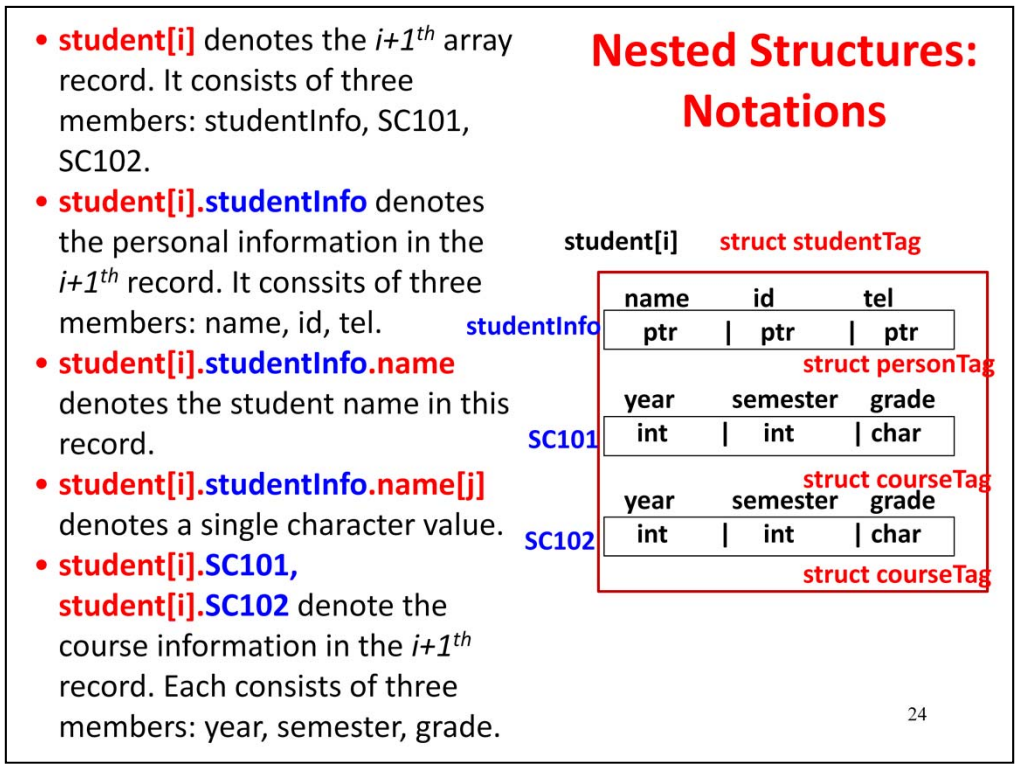
Nested Structures: Operation

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

23

Nested Structures: Example

To access each array element, we use a **for** loop to traverse the array. 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

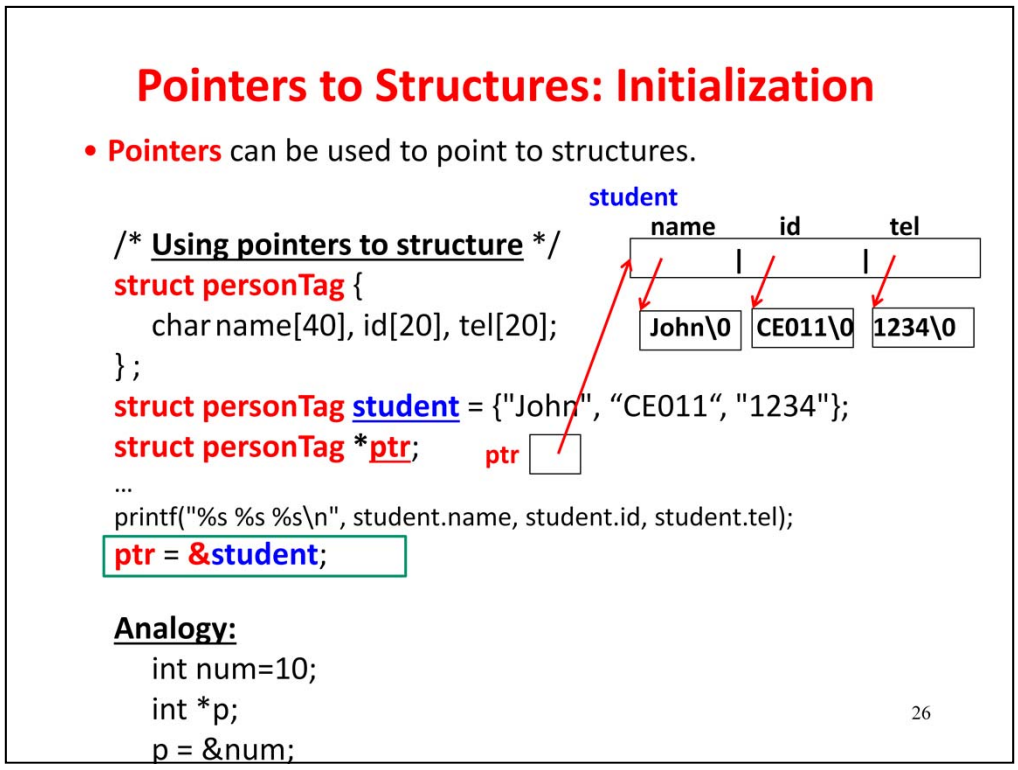
In the nested structure variable **student**, we note the following notations:

- **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
- Nested Structures
- **Pointers to Structures**
- Functions and Structures
- The typedef Construct

25



Pointers to Structures: Initialization

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

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

define a structure template **personTag** and create a pointer **ptr** to the structure **personTag**.

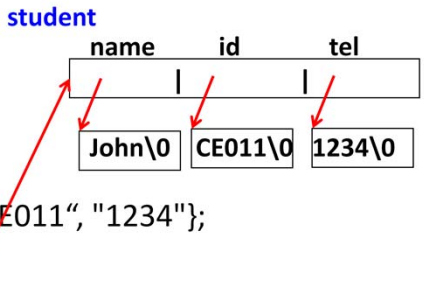
To initialize a pointer, we must use the address operator (**&**) to obtain the address of a structure variable, and then assign the address to the pointer as

```
ptr = &student;
```

The address of the structure variable **student** is assigned to the pointer variable **ptr**.

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", "CE011", "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 */
```

27

Pointers to Structures: Operation

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. 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**.

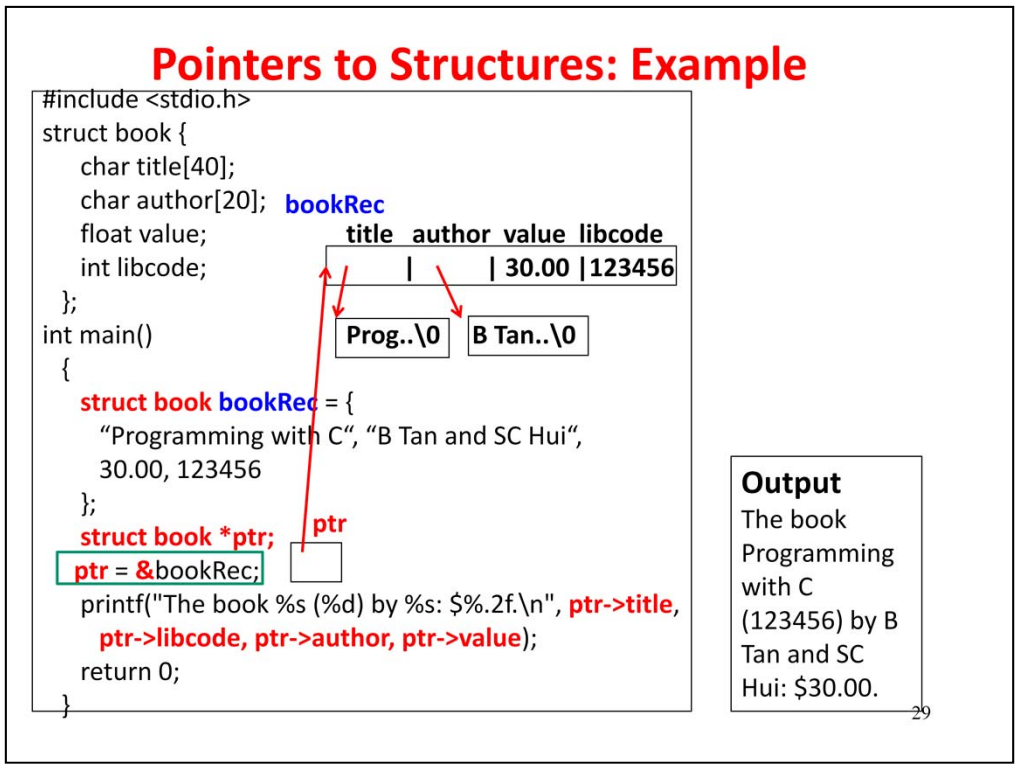
28

Pointers to Structures: Operation

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 (**>**). 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

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

struct book *ptr;

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**. As a result, we may access the members of **bookRec** via **ptr**.
The following statement

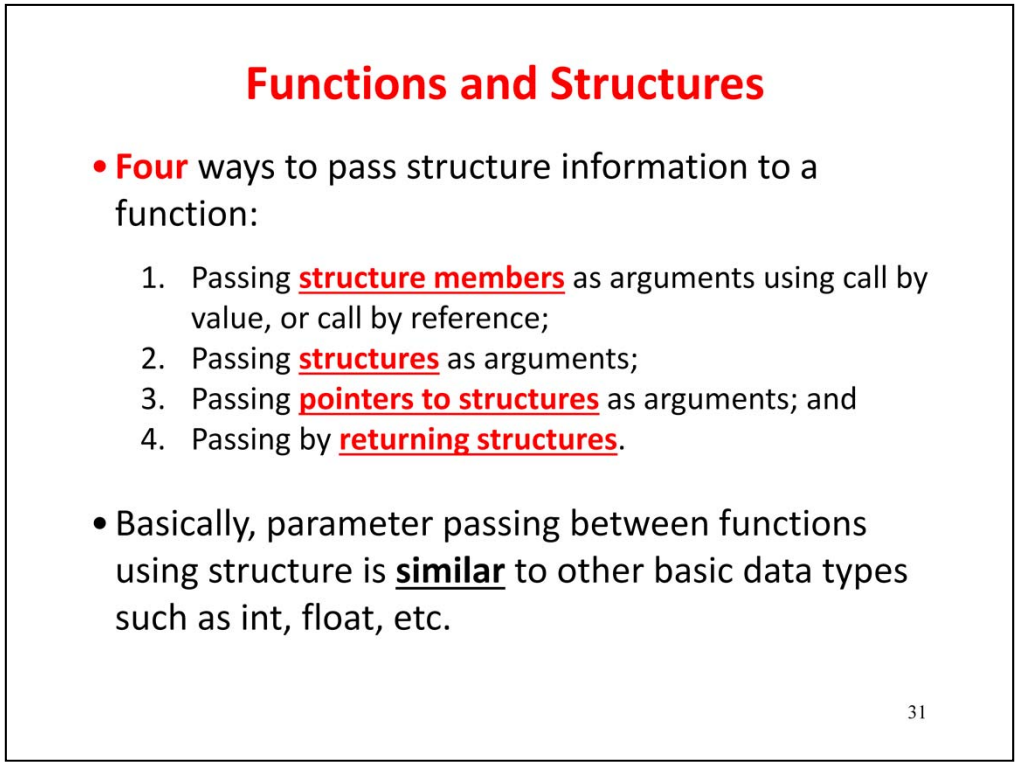
printf("The book %s (%d) by %s: $%.2f.\n", ptr->title, ptr->libcode,
ptr->author, ptr->value);

prints each member information of **bookRec**.

Structures

- Structures
- Arrays of Structures
- Nested Structures
- Pointers to Structures
- **Functions and Structures**
- The typedef Construct

30



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:

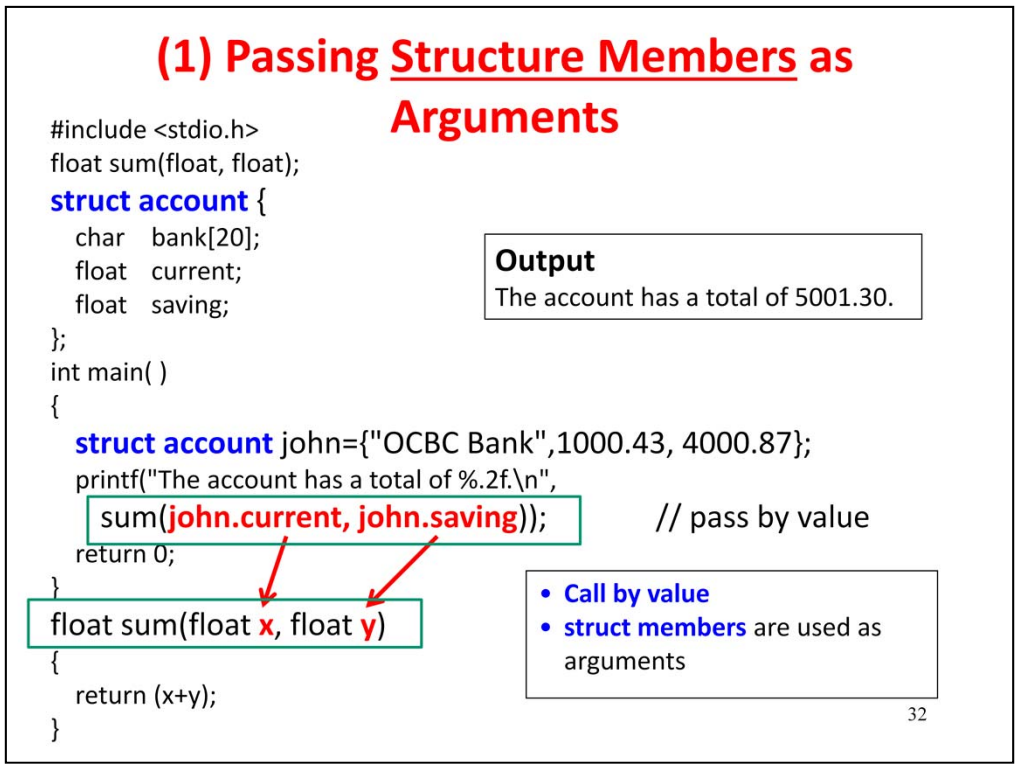
1.Passing structure members as arguments using call by value, or call by reference;

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 other basic data types such as **int**, **float**, etc.



Passing Structure Members as Arguments

In the program, a structure template **account** is defined with three members: **bank**, **current** and **saving**. 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. There are different ways to implement the function **sum()**.

The first approach is to pass individual members of a structure as arguments to a function. 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.

However, it is also possible to pass the structure members using call by reference. However, we need to pass the address of the members to the function **sum()**:

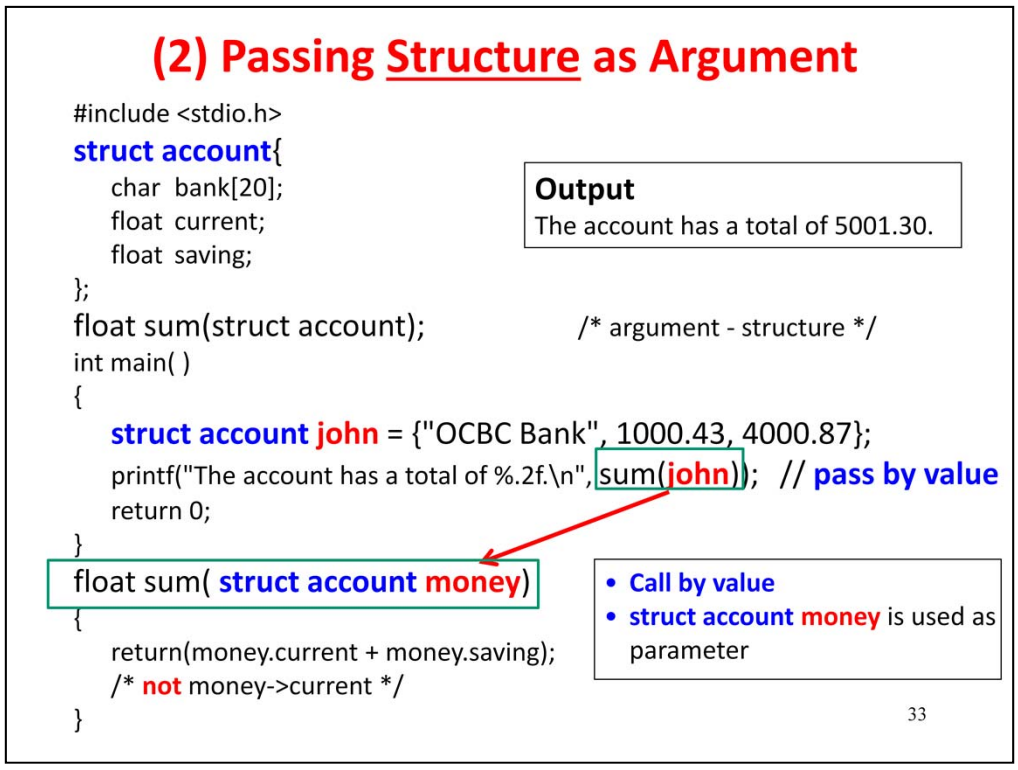
```
void sum(float *x, float *y, float *result)
{
    *result = *x + *y;
}
```

This new function receives the addresses of the variables **x** and **y** of type **float**. The sum of the content of the memory locations pointed to by the two variables is then calculated and

stored in the memory location pointed to by the pointer variable **result**. The function call using the address will be

sum(&john.current, &john.saving, &sum_value);

where the variables **john** and **sum_value** of types **struct account** and **float** respectively are defined respectively in the calling function.

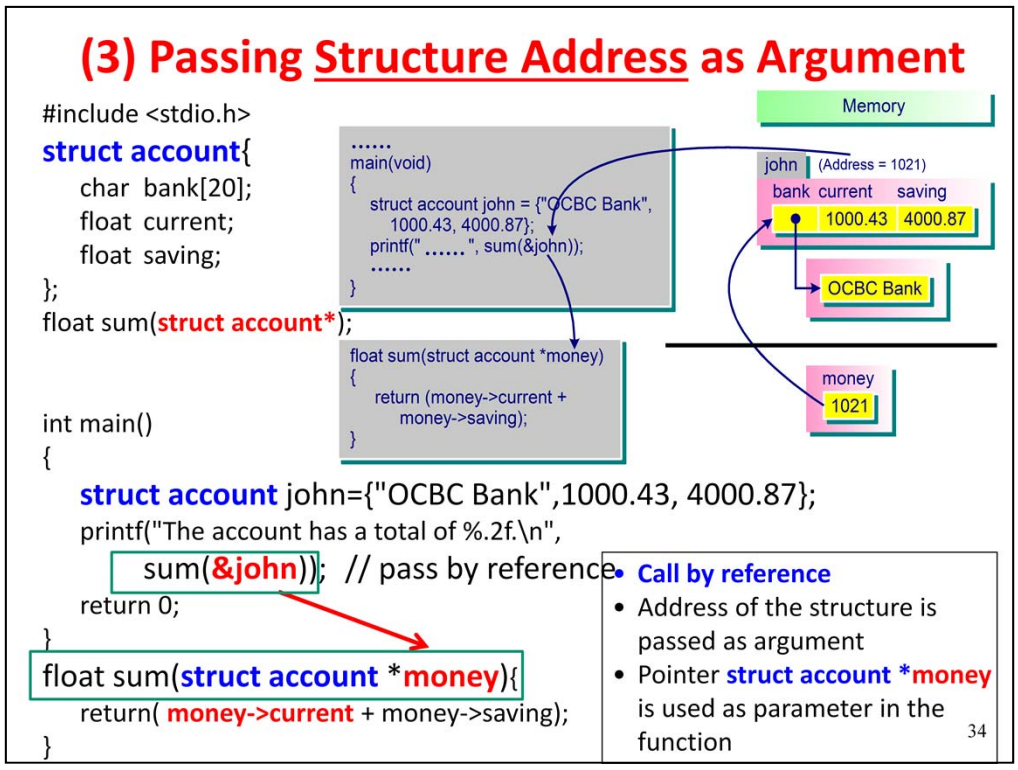


Passing Structures as Arguments

The second approach is to pass a structure to a function as an argument to a function. 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);

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. 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.

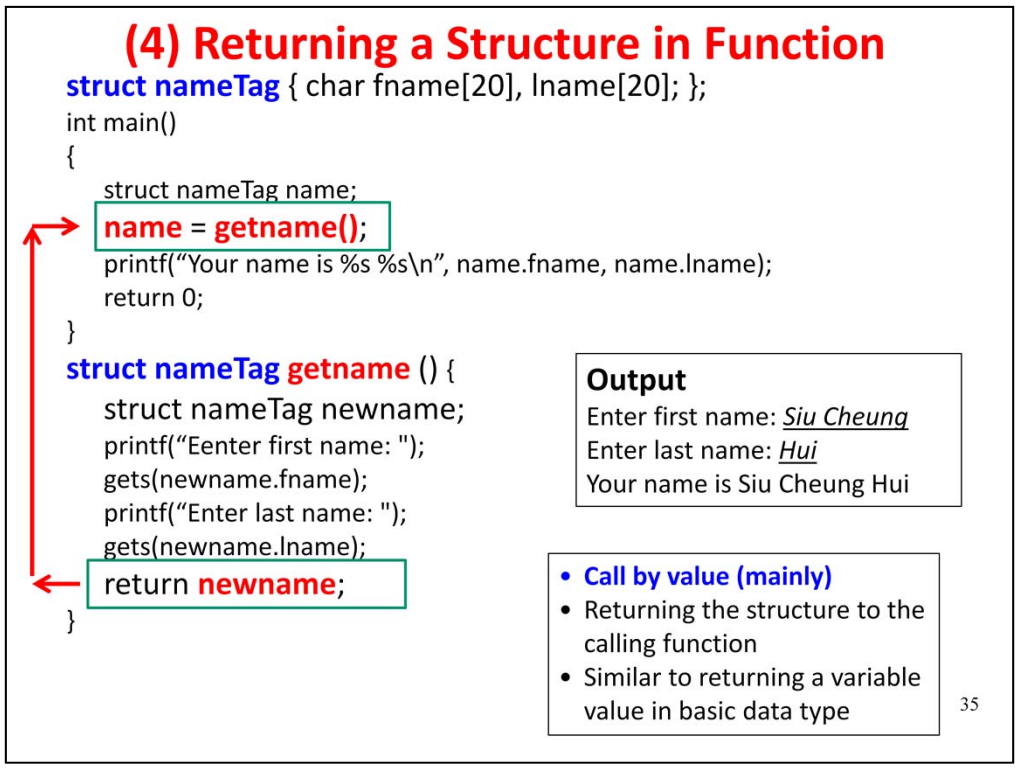


Passing Structure Address

The third approach is to pass the address of the structure as an argument. 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. This is a better approach than passing structures as arguments.



Passing by Returning Structures

The fouth approach is to return the structure in the function. 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;

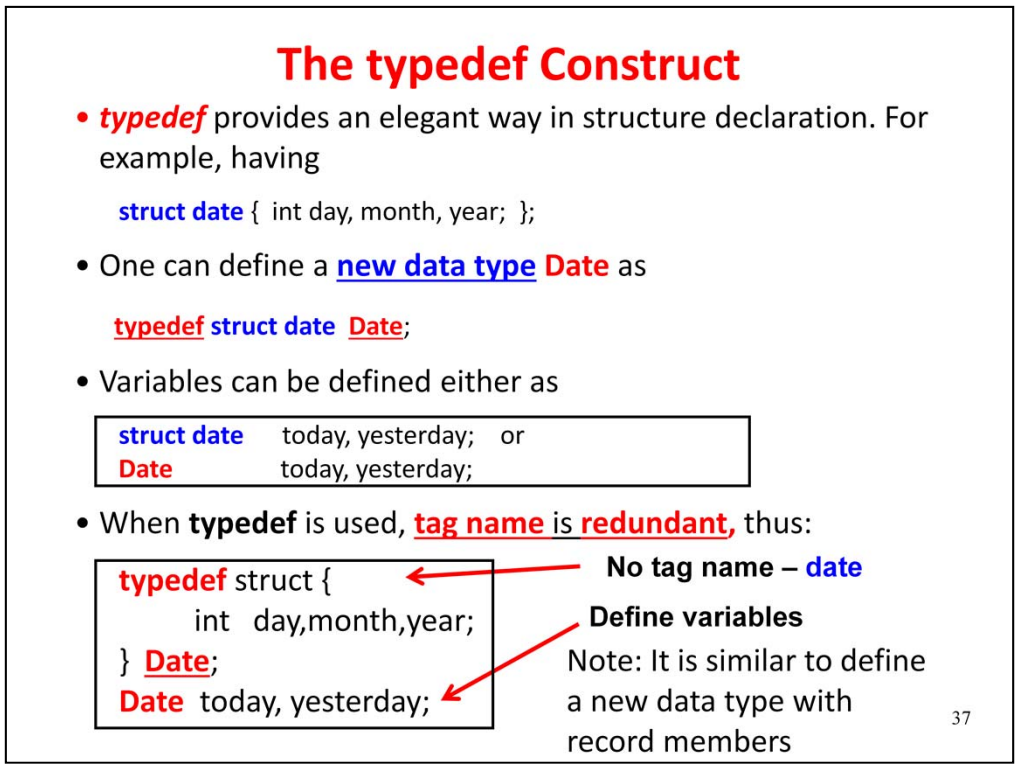
name = getname();

Which assigns the returned structure data to the variable **name**.

Structures

- Structures
- Arrays of Structures
- Nested Structures
- Pointers to Structures
- Functions and Structures
- **The typedef Construct**

36



The typedef Construct

typedef provides an elegant way in structure declaration. 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. 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;

Variables can then be declared either as

struct date today, yesterday;

or **Date today, yesterday;**

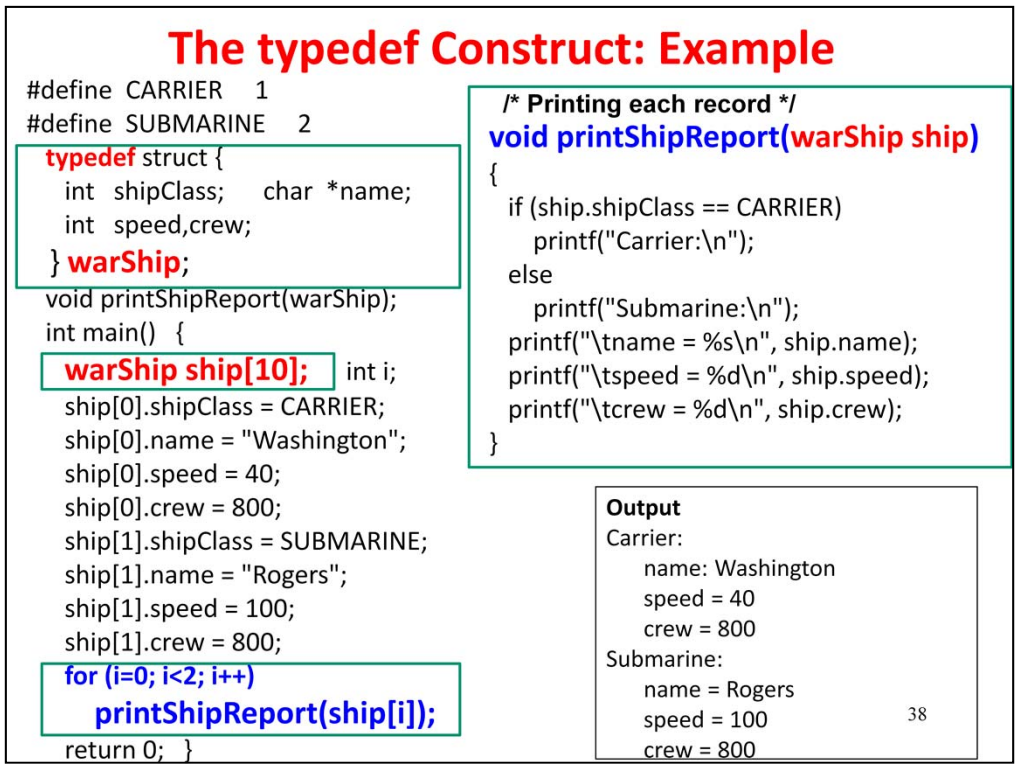
We can also use the type **Date** in function prototypes and function definitions. When **typedef** is used, tag name is redundant. Therefore, we can declare

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

Date today, yesterday;

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.

In addition, **typedef** is similar to the **#define** preprocessor directive. However, there are a number of differences. **typedef** is limited to giving names to data types only and is processed by the compiler, while **#define** is not limited to data types and is processed by the preprocessor.



The typedef Construct: Example

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

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

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.

Quiz: What is the output?

39

```
#include <stdio.h>
struct example {
   struct {
         int x;   int y;
   } in;
   int a;     int b;
};
int main()  {
   struct example e[2];
   struct example *p;

   e[0].a = 1;     e[0].b = 2;
   e[0].in.x = e[0].a * e[0].b;     e[0].in.y = e[0].a + e[0].b;
   printf("%d, %d\n", e[0].in.x, e[0].in.y);

   e[1].a = 3;     e[1].b = 4;
   e[1].in.x = e[1].a * e[1].b;     e[1].in.y = e[1].a + e[1].b;
   printf("%d, %d\n", e[1].in.x, e[1].in.y);


   return 0;}
```

• Nested Structures

Array of nested Structure

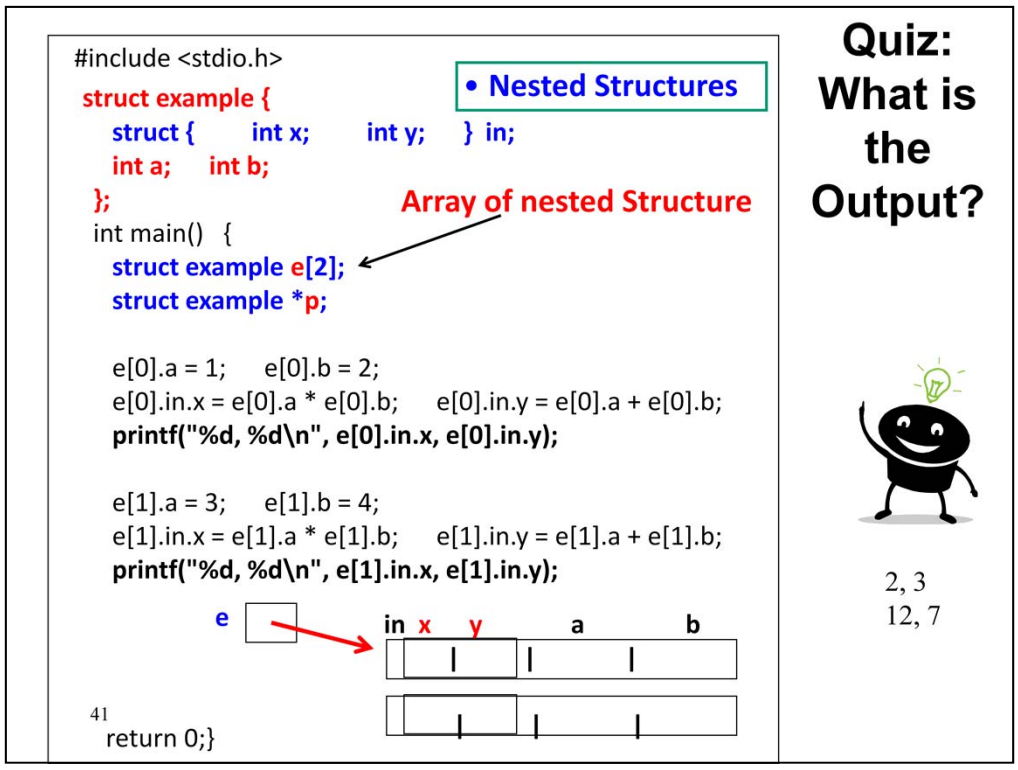
Pointer to Structure

Quiz: What is the Output?

40

Quiz: What is the Output?

Determine the output from the program.

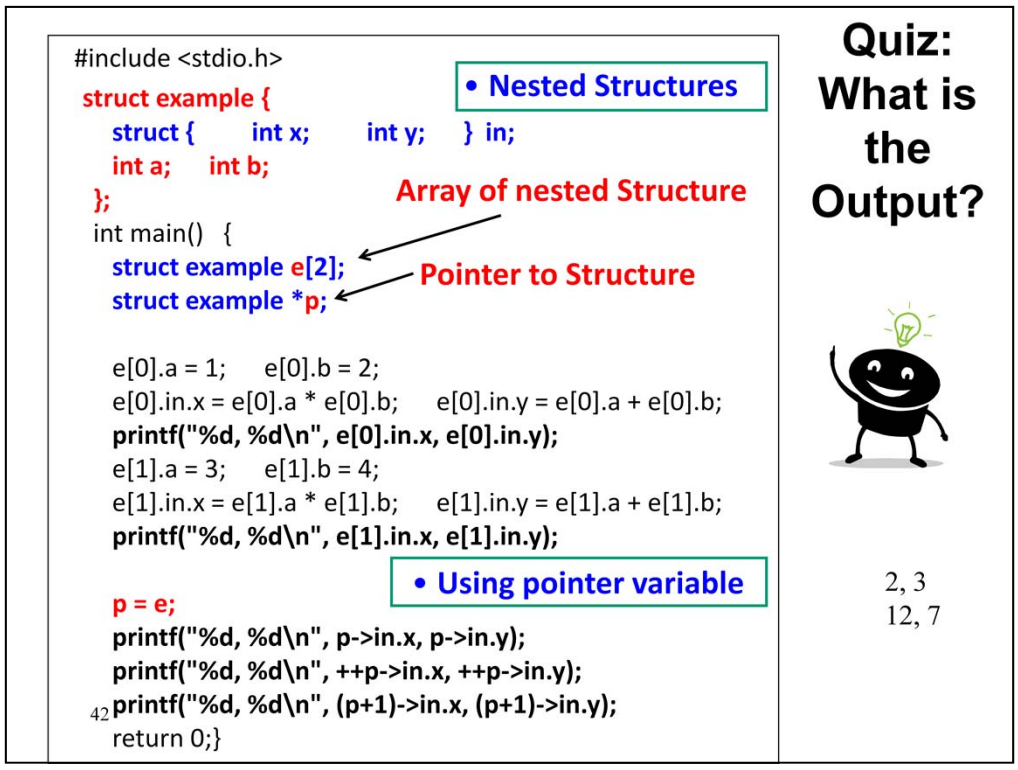


Quiz: What is the Output?

The output from the program is:

2, 3

12, 7



Quiz: What is the Output?

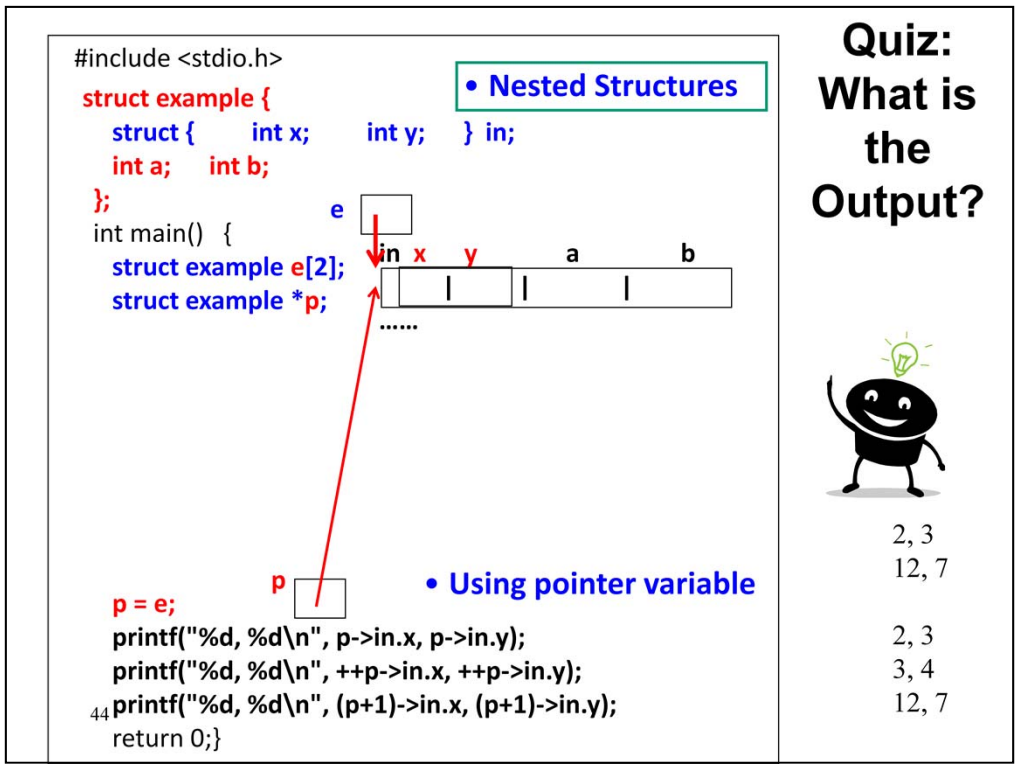
Determine the output from the program.

C Operator Precedence Table

Operator	Description	Associativity
( )	Parentheses (function call)	left-to-right
[ ]	Brackets (array subscript)	
.	**Member selection via object name**	
->	**Member selection via pointer**	
++ --	**Postfix increment/decrement**	
++ --	**Prefix increment/decrement**	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	**Dereference**	
&	**Address (of operand)**	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	

C Operator Precedence Table

You may need to refer to the precedence table to do this quiz.



44

Quiz: What is the Output?

The final output from the program is:

2, 3

12, 7

2, 3

3, 4

12, 7