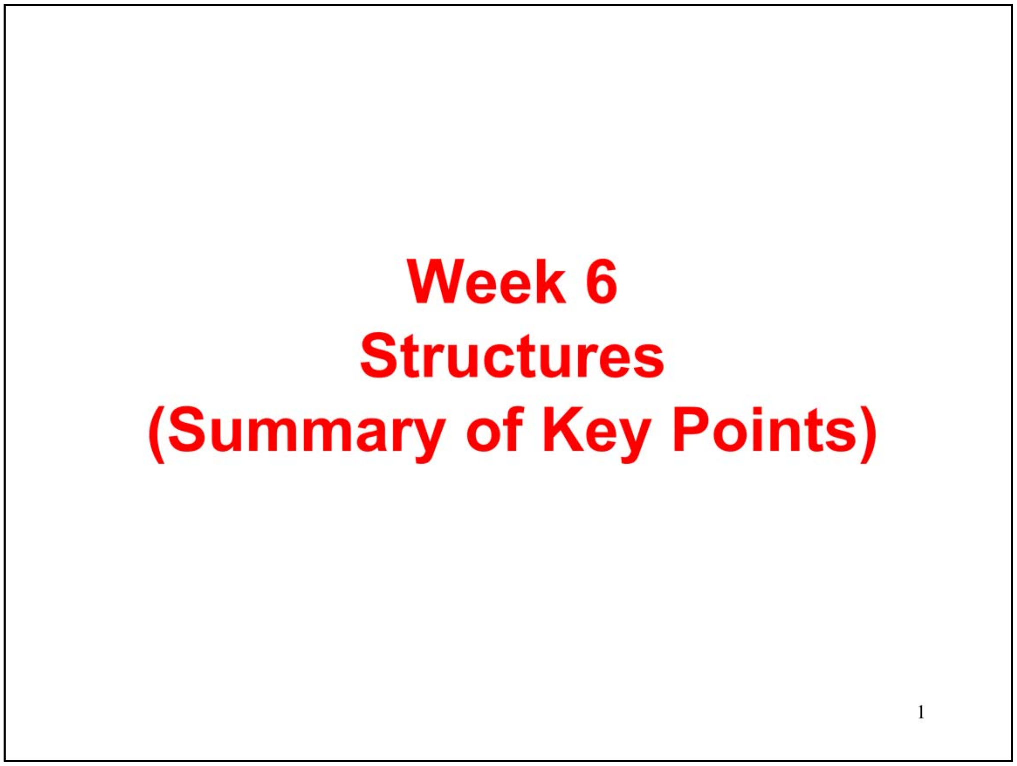
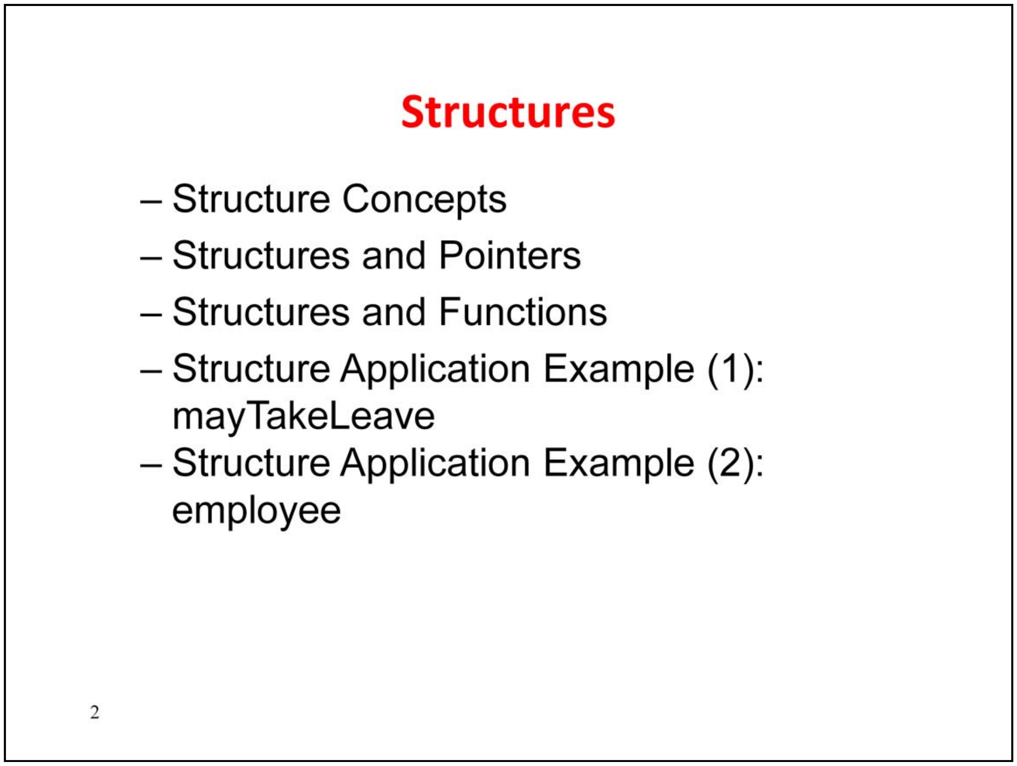
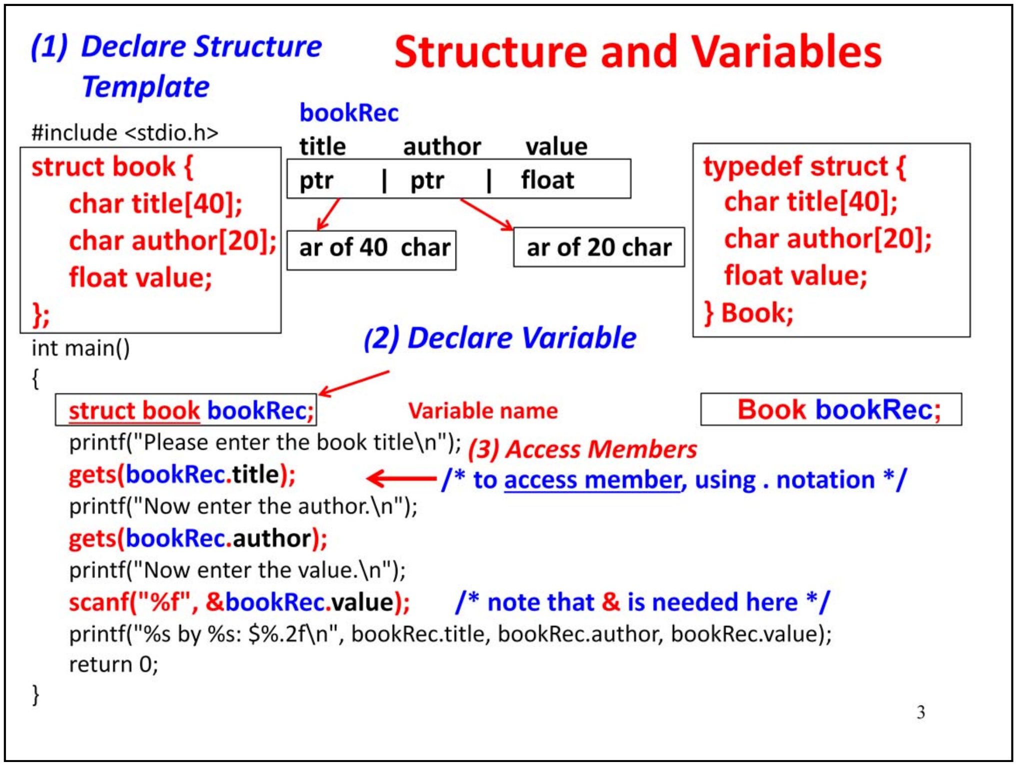


Week 6
Structures
(Summary of Key Points)



Structures

- Structure Concepts
- Structures and Pointers
- Structures and Functions
- Structure Application Example (1): mayTakeLeave
- Structure Application Example (2): employee



(1) Declare Structure Template

Structure and Variables

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

bookRec

title	author	value
ptr	ptr	float

ar of 40 char

ar of 20 char

```
typedef struct {
  char title[40];
  char author[20];
  float value;
} Book;
```

(2) Declare Variable

```
int main()
{
    struct book bookRec;
    printf("Please enter the book title\n");
    gets(bookRec.title);
    printf("Now enter the author.\n");
    gets(bookRec.author);
    printf("Now enter the value.\n");
    scanf("%f", &bookRec.value);      /* note that & is needed here */
    printf("%s by %s: $%.2f\n", bookRec.title, bookRec.author, bookRec.value);
    return 0;
}
```

Variable name

Book bookRec;

(3) Access Members

/* to access member, using . notation */

3

Structure and Variables

(1) 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 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.

(2) 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.

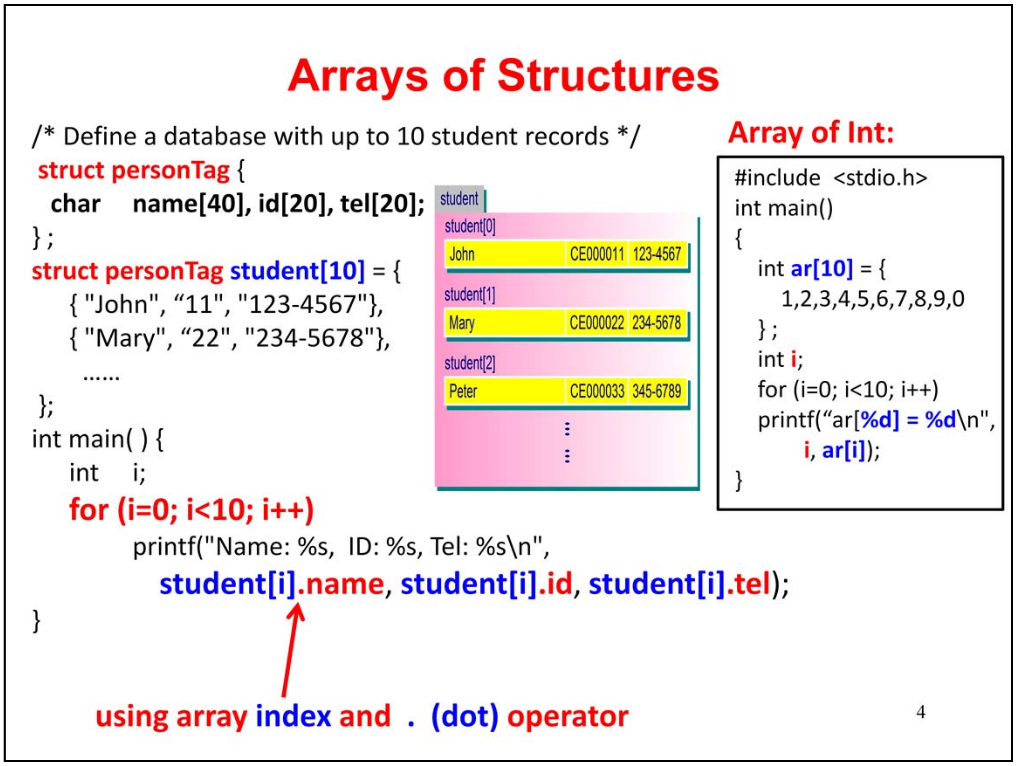
With tag name, we can use the structure data type subsequently in the program.

2. In the program, it defines the structure template (or data type) and the declaration of a structure variable.
3. 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.
4. 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.
5. 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.



Arrays of Structures

```
/* 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);
}
```

student
student[0]
John CE000011 123-4567
student[1]
Mary CE000022 234-5678
student[2]
Peter CE000033 345-6789

Array of Int:

```
#include  <stdio.h>
int main()
{
   int ar[10] = {
      1,2,3,4,5,6,7,8,9,0
   } ;
   int i;
   for (i=0; i<10; i++)
   printf(“ar[%d] = %d\n",
         i, ar[i]);
}
```

using array index and . (dot) operator

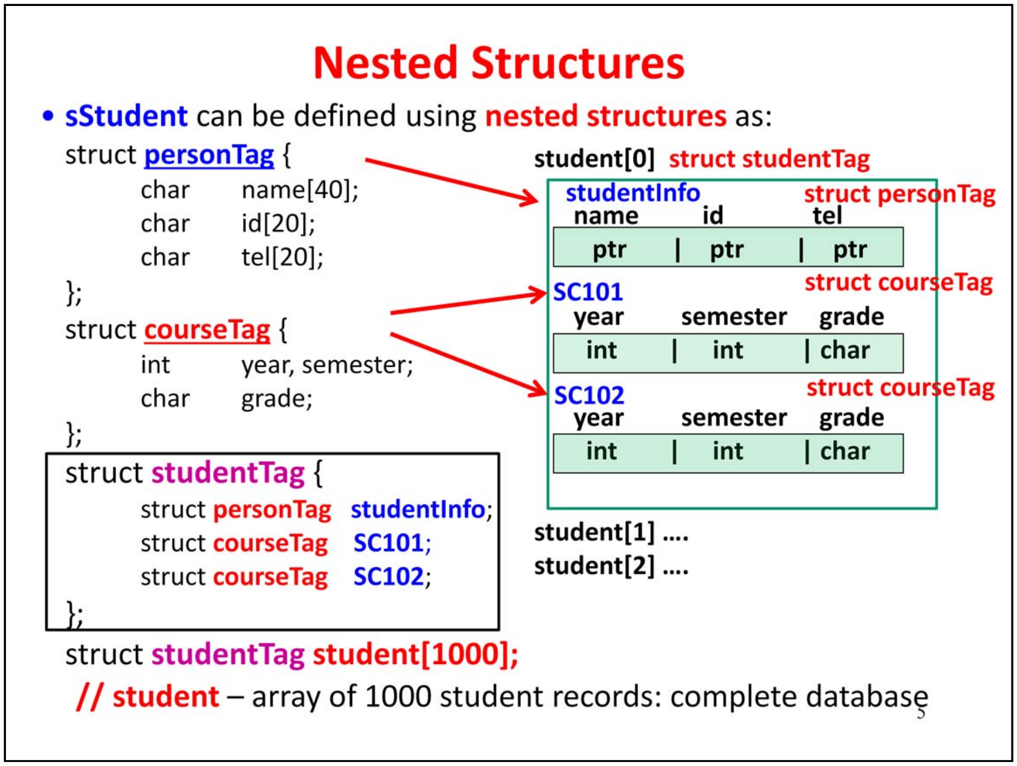
4

Arrays of Structures

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

- **sStudent** can be defined using **nested structures** as:

```
struct personTag {
        char       name[40];
        char       id[20];
        char       tel[20];
};
struct courseTag {
        int        year, semester;
        char       grade;
};
struct studentTag {
        struct personTag   studentInfo;
        struct courseTag   SC101;
        struct courseTag   SC102;
};
struct studentTag student[1000];
 // student – array of 1000 student records: complete database
```

Nested Structures

1. 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.

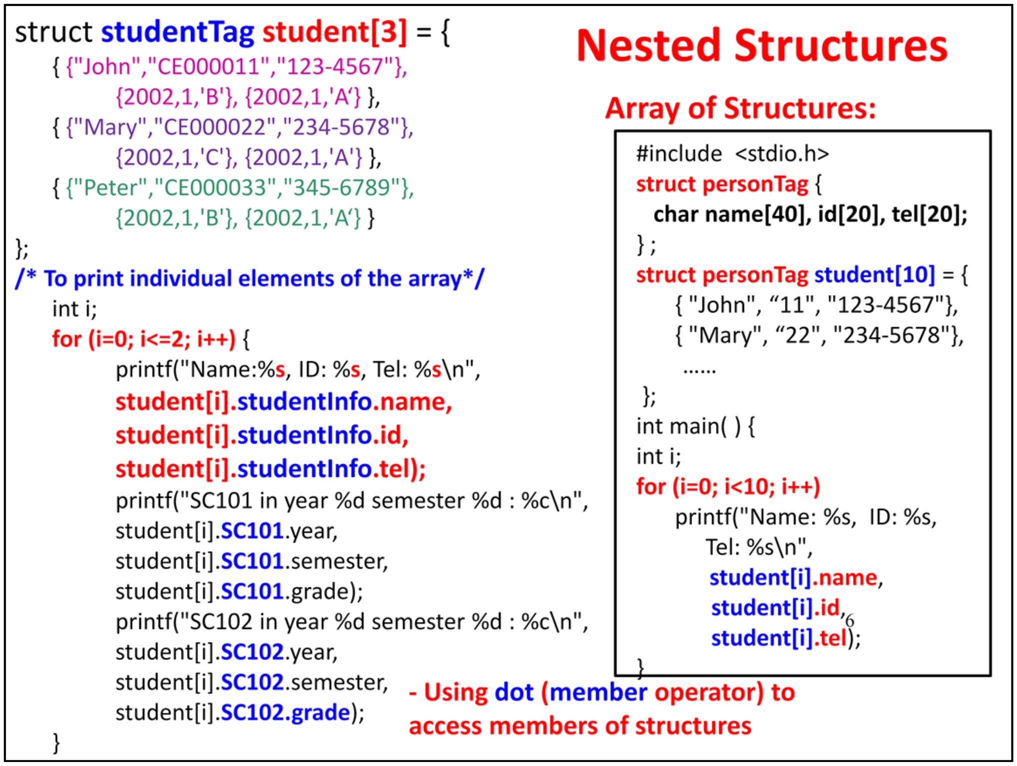
```
struct personTag {
     char       name[40];
     char       id[20];
     char       tel[20];
};
```

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

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

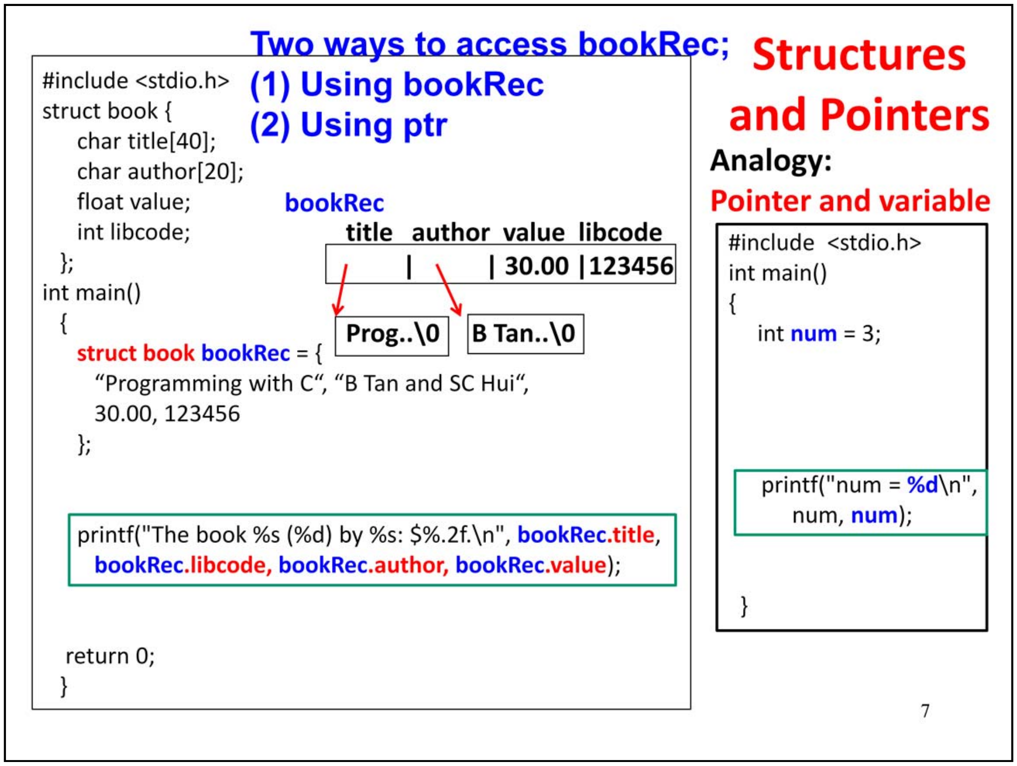
Array of Structures:

```
#include  <stdio.h>
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);
}
```

- 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.



Structures and Pointers

Two ways to access bookRec;
(1) Using bookRec
(2) Using ptr

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

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

    return 0;
  }
```

bookRec
title
author
value
libcode
30.00
123456
Prog..\0
B Tan..\0

Analogy:
Pointer and variable

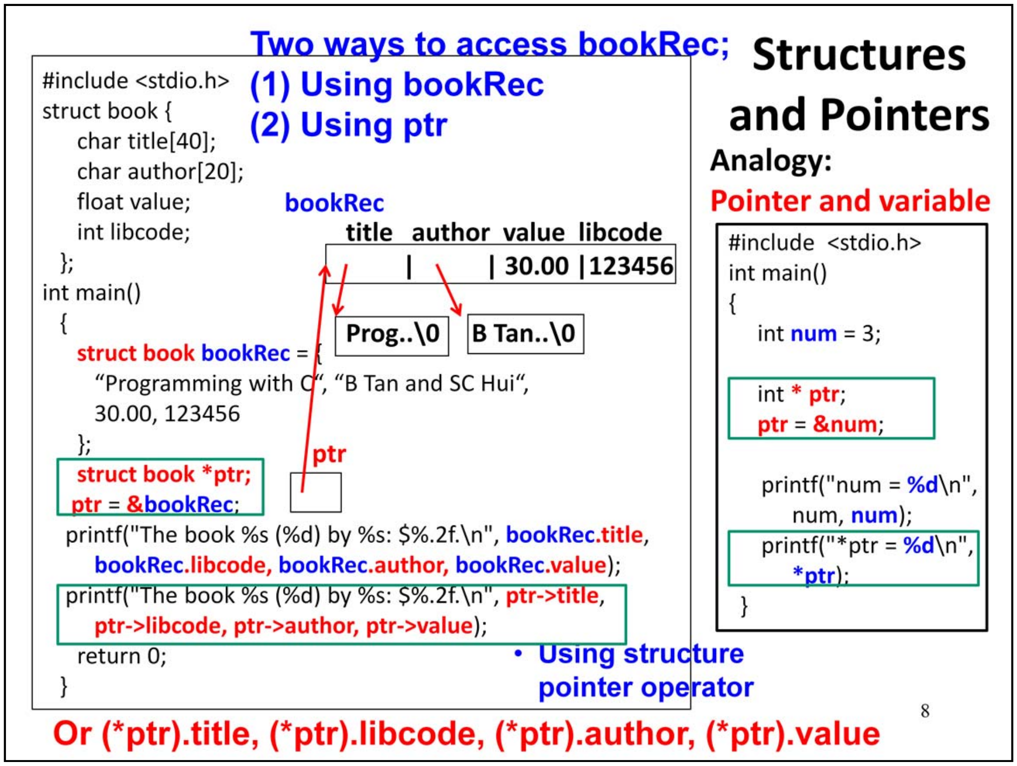
```
#include  <stdio.h>
int main()
{
   int num = 3;

    printf("num = %d\n",
        num, num);

 }
```

7

Structures and Pointers

1.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.

2.We can then use the structure variable **bookRec** to access each member of the structure.



Structures and Pointers

Two ways to access bookRec;
(1) Using bookRec
(2) Using ptr

```
#include <stdio.h>
struct book {
    char title[40];
    char author[20];
    float value;
    int libcode;
  };
int main()
  {
    struct book bookRec = {
      “Programming with C“, “B Tan and SC Hui“,
      30.00, 123456
    };
    struct book *ptr;
    ptr = &bookRec;
   printf("The book %s (%d) by %s: $%.2f.\n", bookRec.title,
      bookRec.libcode, bookRec.author, bookRec.value);
   printf("The book %s (%d) by %s: $%.2f.\n", ptr->title,
      ptr->libcode, ptr->author, ptr->value);
    return 0;
  }
```

- Using structure pointer operator

Or (*ptr).title, (*ptr).libcode, (*ptr).author, (*ptr).value

Analogy:
Pointer and variable

```
#include  <stdio.h>
int main()
{
   int num = 3;

   int * ptr;
   ptr = &num;

    printf("num = %d\n",
       num, num);
    printf("*ptr = %d\n",
       *ptr);
 }
```

8

Structures and Pointers

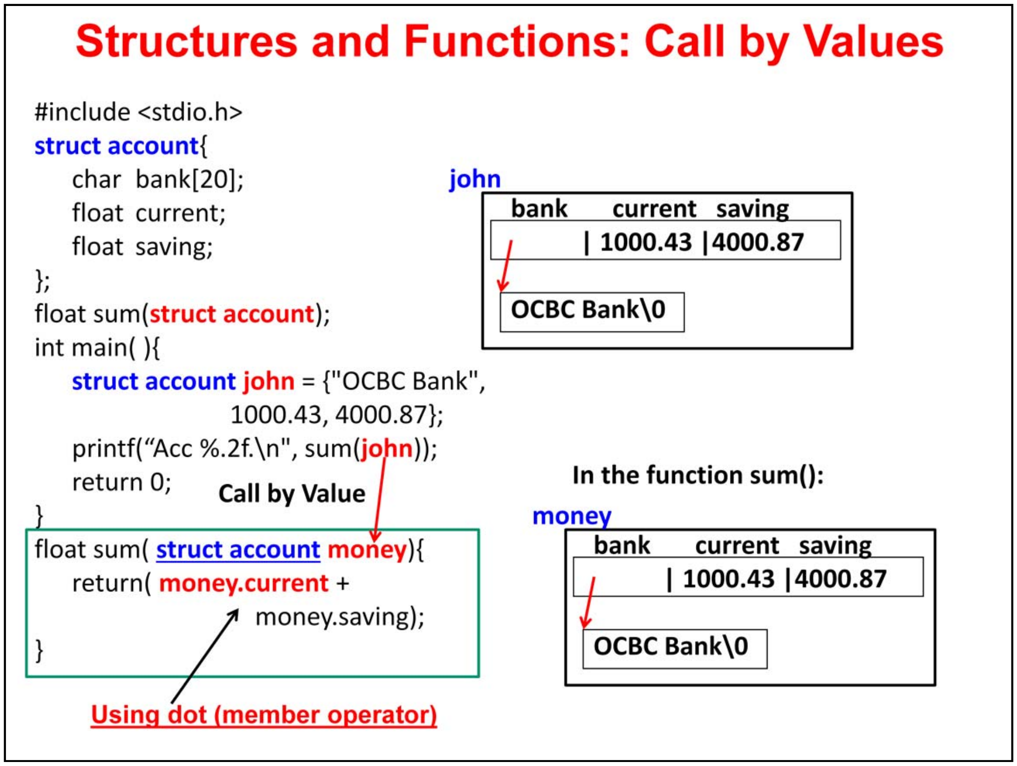
1.We can also use pointer variable to access each member of the structure.

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

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

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

5.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 and Functions: Call by Values

```
#include <stdio.h>
struct account{
    char  bank[20];
    float  current;
    float  saving;
};
float sum(struct account);
int main( ){
    struct account john = {"OCBC Bank",
                        1000.43, 4000.87};
    printf("Acc %.2f.\n", sum(john));
    return 0;
}
float sum( struct account money){
    return( money.current +
                    money.saving);
}
```

Call by Value

Using dot (member operator)

john

bank	current	saving
	1000.43	4000.87

OCBC Bank\0

In the function sum():

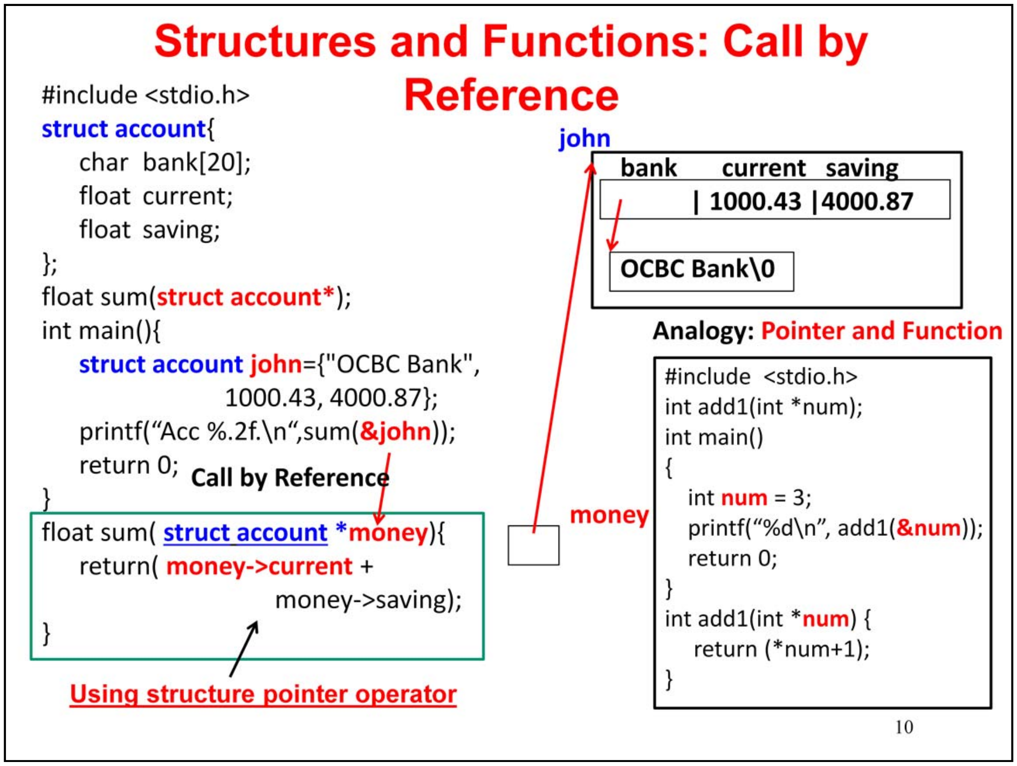
money

bank	current	saving
	1000.43	4000.87

OCBC Bank\0

Structures and Functions: Call by Value

1. The 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.



Structures and Functions: Call by Reference

```
#include <stdio.h>
struct account{
    char  bank[20];
    float  current;
    float  saving;
};
float sum(struct account*);
int main(){
    struct account john={"OCBC Bank",
                 1000.43, 4000.87};
    printf(“Acc %.2f.\n“,sum(&john));
    return 0;
}
float sum( struct account *money){
    return( money->current +
                    money->saving);
}
```

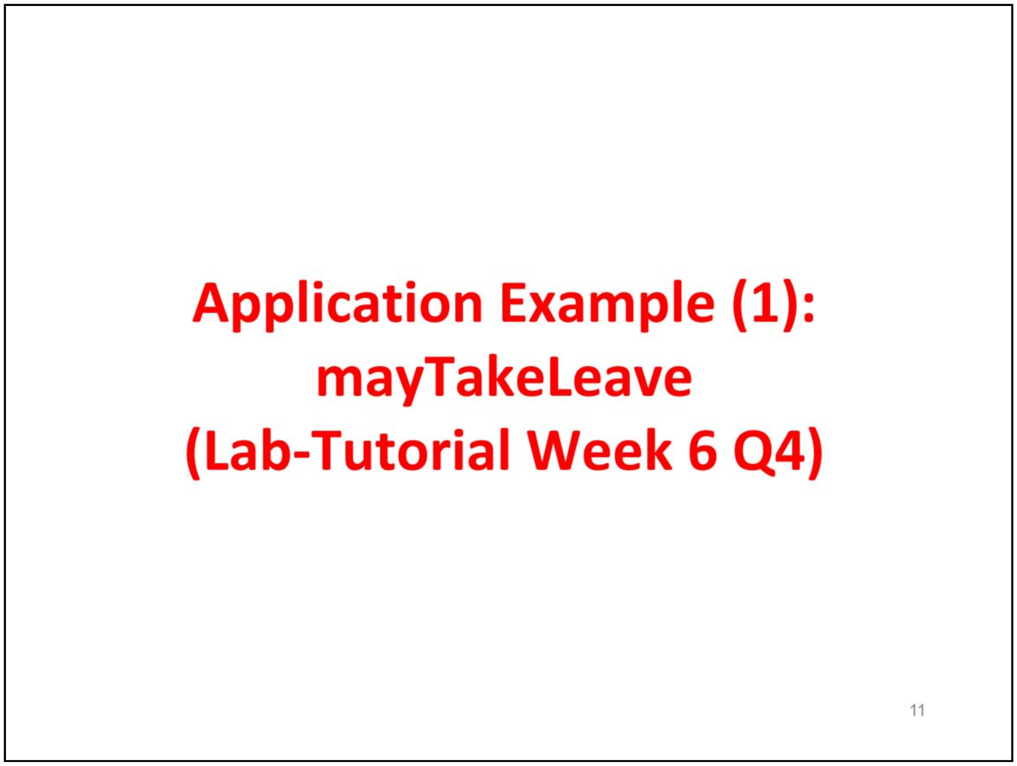
Analogy: Pointer and Function

```
#include  <stdio.h>
int add1(int *num);
int main()
{
   int num = 3;
   printf(“%d\n”, add1(&num));
   return 0;
}
int add1(int *num) {
    return (*num+1);
}
```

10

Structures and Functions: Call by Reference

1. The 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.

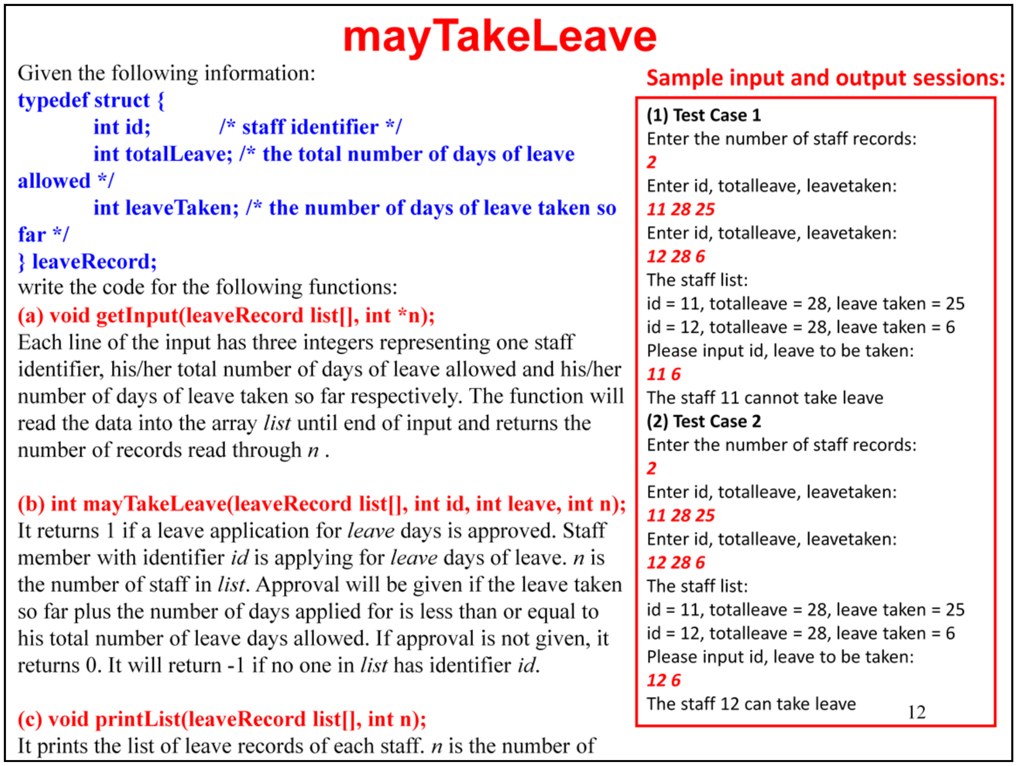


Application Example (1):
mayTakeLeave
(Lab-Tutorial Week 6 Q4)

11

Application Example (1): mayTakeLeave()

1. We use the application mayTakeLeave as an example to illustrate the use of functions in application development.



mayTakeLeave

Given the following information:

```
typedef struct {
        int id;         /* staff identifier */
        int totalLeave; /* the total number of days of leave allowed */
        int leaveTaken; /* the number of days of leave taken so far */
} leaveRecord;
```

write the code for the following functions:

(a) void getInput(leaveRecord list[], int *n);

Each line of the input has three integers representing one staff identifier, his/her total number of days of leave allowed and his/her number of days of leave taken so far respectively. The function will read the data into the array *list* until end of input and returns the number of records read through *n* .

(b) int mayTakeLeave(leaveRecord list[], int id, int leave, int n);

It returns 1 if a leave application for *leave* days is approved. Staff member with identifier *id* is applying for *leave* days of leave. *n* is the number of staff in *list*. Approval will be given if the leave taken so far plus the number of days applied for is less than or equal to his total number of leave days allowed. If approval is not given, it returns 0. It will return -1 if no one in *list* has identifier *id*.

(c) void printList(leaveRecord list[], int n);

It prints the list of leave records of each staff. *n* is the number of

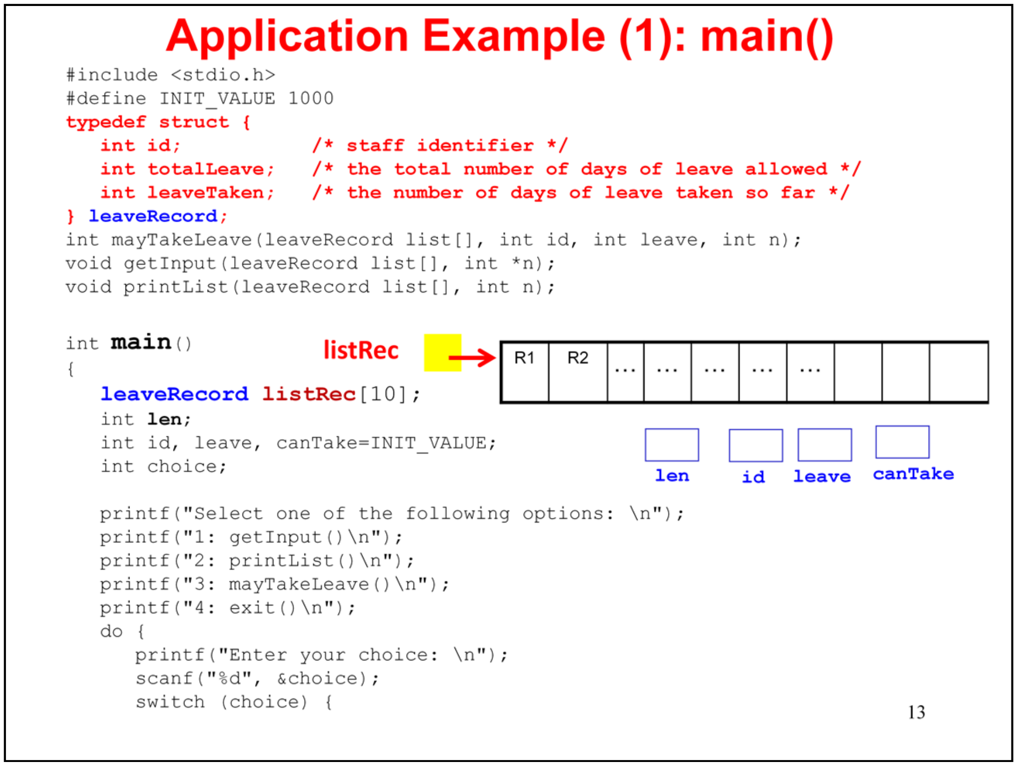
Sample input and output sessions:

```
(1) Test Case 1
Enter the number of staff records:
2
Enter id, totalleave, leavetaken:
11 28 25
Enter id, totalleave, leavetaken:
12 28 6
The staff list:
id = 11, totalleave = 28, leave taken = 25
id = 12, totalleave = 28, leave taken = 6
Please input id, leave to be taken:
11 6
The staff 11 cannot take leave
(2) Test Case 2
Enter the number of staff records:
2
Enter id, totalleave, leavetaken:
11 28 25
Enter id, totalleave, leavetaken:
12 28 6
The staff list:
id = 11, totalleave = 28, leave taken = 25
id = 12, totalleave = 28, leave taken = 6
Please input id, leave to be taken:
12 6
The staff 12 can take leave
```

12

Application Example: mayTakeLeave

1. The application problem specification is given here.



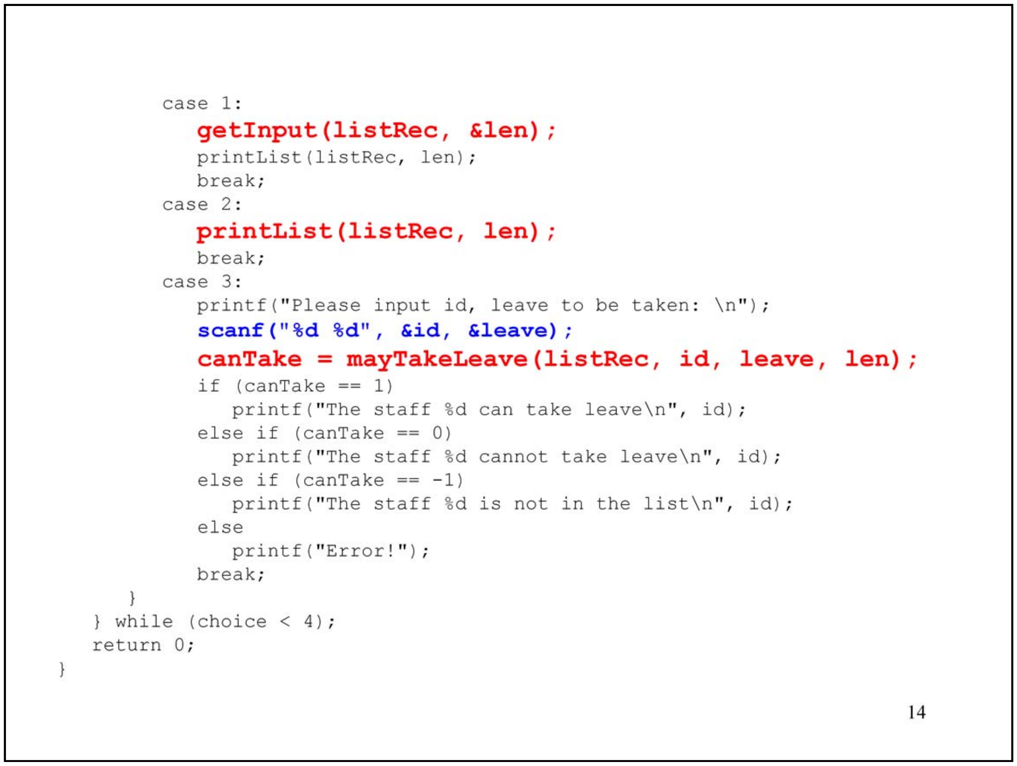
Application Example (1): main()

```
#include <stdio.h>
#define INIT_VALUE 1000
typedef struct {
   int id;           /* staff identifier */
   int totalLeave;   /* the total number of days of leave allowed */
   int leaveTaken;   /* the number of days of leave taken so far */
} leaveRecord;
int mayTakeLeave(leaveRecord list[], int id, int leave, int n);
void getInput(leaveRecord list[], int *n);
void printList(leaveRecord list[], int n);

int main()
{
   leaveRecord listRec[10];
   int len;
   int id, leave, canTake=INIT_VALUE;
   int choice;

   printf("Select one of the following options: \n");
   printf("1: getInput()\n");
   printf("2: printList()\n");
   printf("3: mayTakeLeave()\n");
   printf("4: exit()\n");
   do {
      printf("Enter your choice: \n");
      scanf("%d", &choice);
      switch (choice) {
```

13

Application Example: main()

1. The **main()** function is given here.

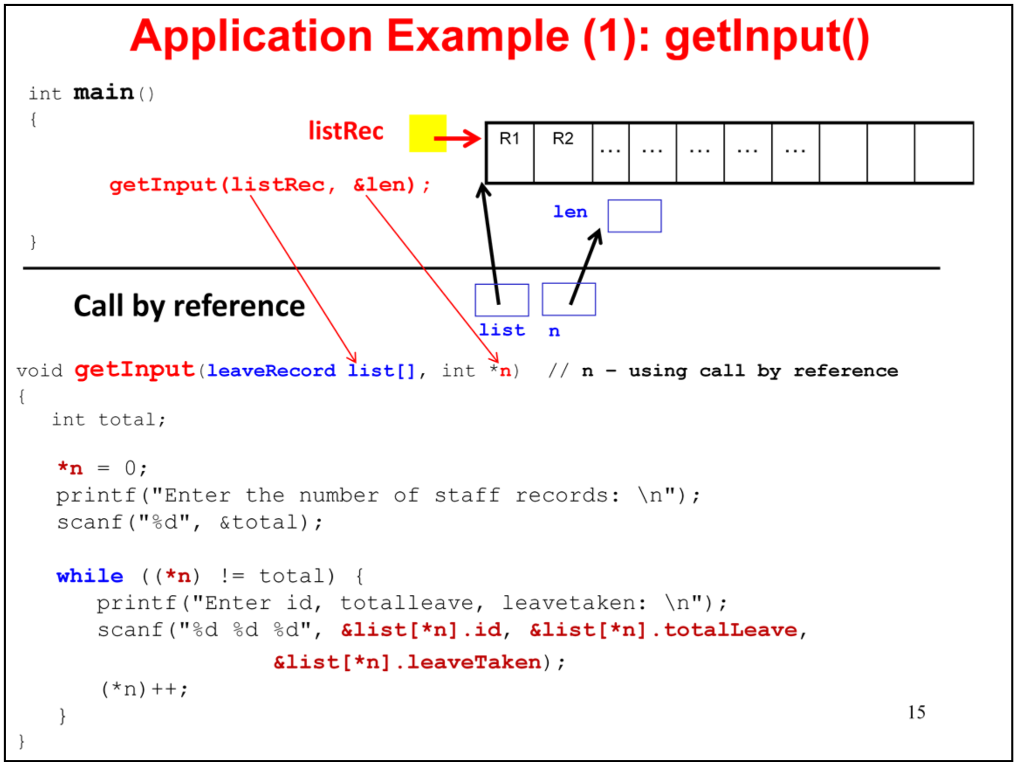


```
         case 1:
            getInput(listRec, &len);
            printList(listRec, len);
            break;
         case 2:
            printList(listRec, len);
            break;
         case 3:
            printf("Please input id, leave to be taken: \n");
            scanf("%d %d", &id, &leave);
            canTake = mayTakeLeave(listRec, id, leave, len);
            if (canTake == 1)
               printf("The staff %d can take leave\n", id);
            else if (canTake == 0)
               printf("The staff %d cannot take leave\n", id);
            else if (canTake == -1)
               printf("The staff %d is not in the list\n", id);
            else
               printf("Error!");
            break;
      }
   } while (choice < 4);
   return 0;
}
```

14

Application Example: main()

1. The **main()** function is given here.



Application Example (1): getInput()

```
int main()
{
      getInput(listRec, &len);

}
```

listRec
R1
R2
…
…
…
…
…
len
list
n

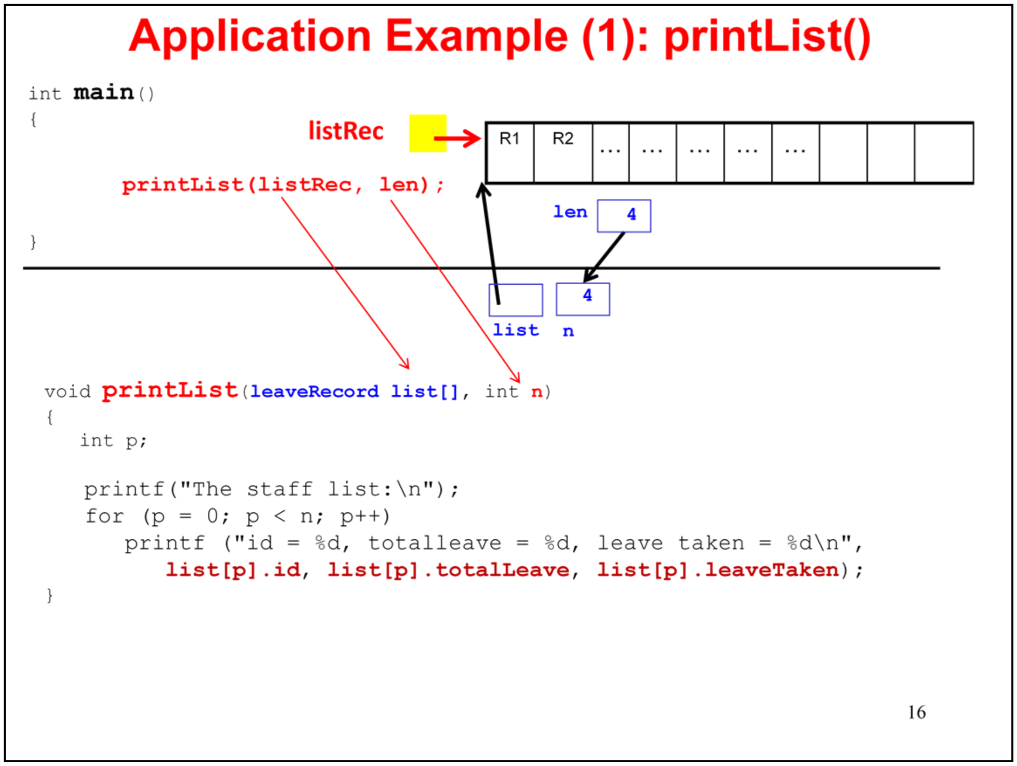
Call by reference

```
void getInput(leaveRecord list[], int *n)  // n – using call by reference
{
   int total;

   *n = 0;
   printf("Enter the number of staff records: \n");
   scanf("%d", &total);

   while ((*n) != total) {
      printf("Enter id, totalleave, leavetaken: \n");
      scanf("%d %d %d", &list[*n].id, &list[*n].totalLeave,
                     &list[*n].leaveTaken);
      (*n)++;
   }
}
```

15

Application Example: getInput()

1. The **getInput()** function is given here.



Application Example (1): printList()

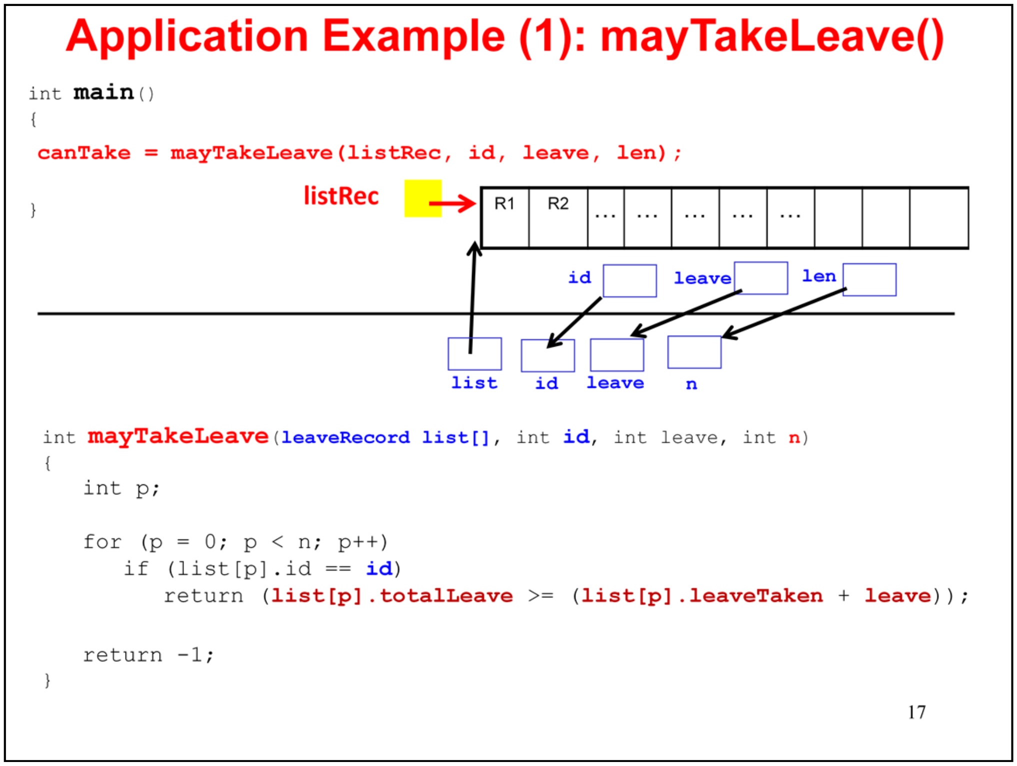
```
int main()
{

      printList(listRec, len);

}
```

```
 void printList(leaveRecord list[], int n)
 {
    int p;

    printf("The staff list:\n");
    for (p = 0; p < n; p++)
       printf ("id = %d, totalleave = %d, leave taken = %d\n",
          list[p].id, list[p].totalLeave, list[p].leaveTaken);
 }
```

16

Application Example: printList()

1. The **printList()** function is given here.



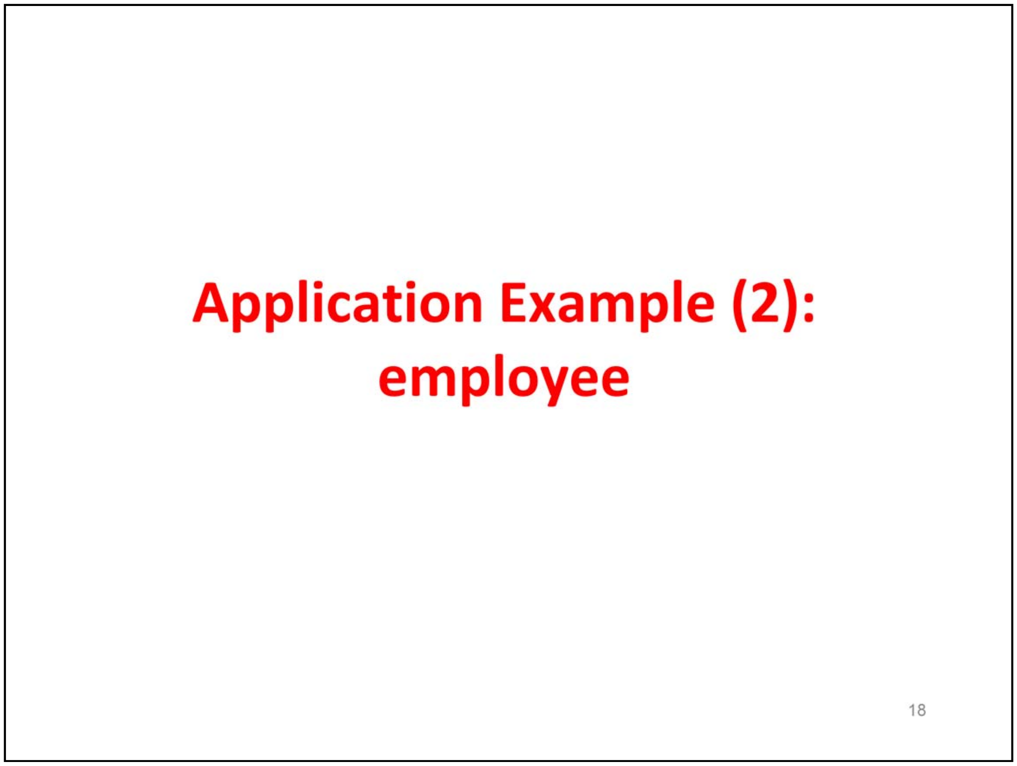
Application Example (1): mayTakeLeave()

```
int main()
{
 canTake = mayTakeLeave(listRec, id, leave, len);

}
```

```
int mayTakeLeave(leaveRecord list[], int id, int leave, int n)
{
   int p;

   for (p = 0; p < n; p++)
      if (list[p].id == id)
         return (list[p].totalLeave >= (list[p].leaveTaken + leave));

   return -1;
}
```

Application Example: mayTakeLeave()

1. The **mayTakeLeave()** function is given here.

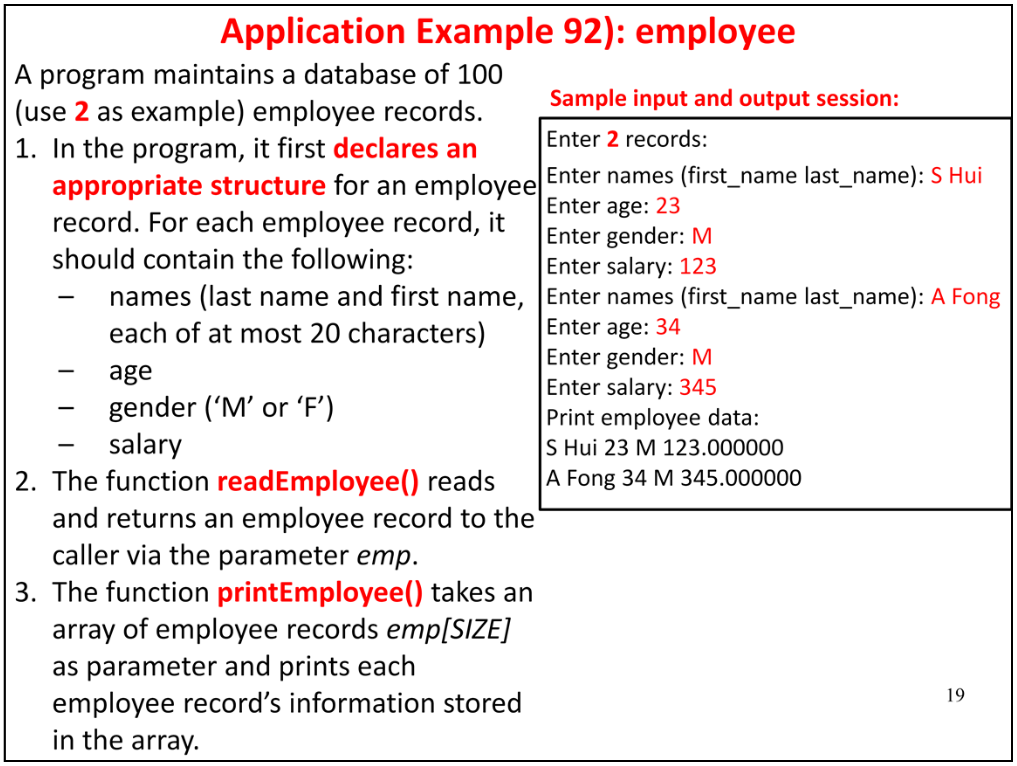


Application Example (2): employee

18

Application Example

1. We use the application mayTakeLeave as an example to illustrate the use of functions in application development.



Application Example 92): employee

A program maintains a database of 100 (use **2** as example) employee records.

1. In the program, it first **declares an appropriate structure** for an employee record. For each employee record, it should contain the following:
   - names (last name and first name, each of at most 20 characters)
   - age
   - gender (‘M’ or ‘F’)
   - salary
2. The function **readEmployee()** reads and returns an employee record to the caller via the parameter *emp*.
3. The function **printEmployee()** takes an array of employee records *emp[SIZE]* as parameter and prints each employee record’s information stored in the array.

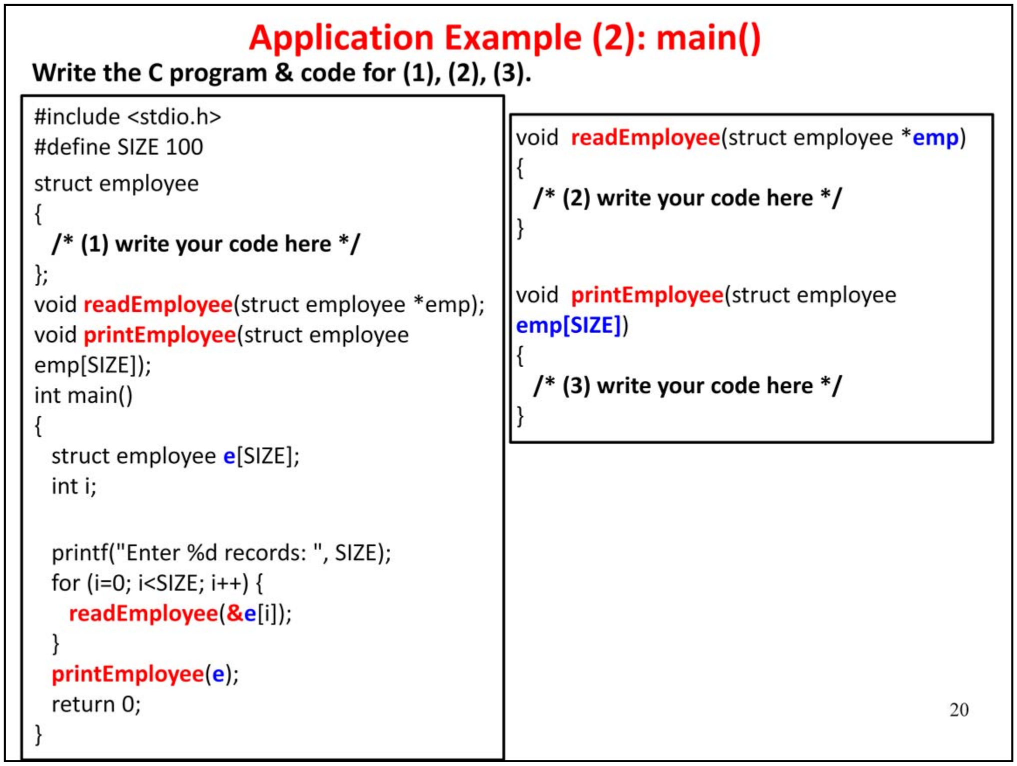
Sample input and output session:

```
Enter 2 records:
Enter names (first_name last_name): S Hui
Enter age: 23
Enter gender: M
Enter salary: 123
Enter names (first_name last_name): A Fong
Enter age: 34
Enter gender: M
Enter salary: 345
Print employee data:
S Hui 23 M 123.000000
A Fong 34 M 345.000000
```

Application Example (2): Employee

The application is specified as follows:

1. A program maintains a database of 100 (using **2** as an example) employee records.
2. In the program, it first **declares an appropriate structure** for an employee record. For each employee record, it should contain the following:
   - names (last name and first name, each of at most 20 characters)
   - age
   - gender (‘M’ or ‘F’)
   - salary
3. The function **readEmployee()** reads and returns an employee record to the caller via the parameter **emp**. The function **printEmployee()** takes an array of employee records **emp[SIZE]** as parameter and prints each employee record’s information stored in the array.
4. In this application, it is required to implement two functions: **readEmployee()** and **printEmployee()**.



Application Example (2): main()

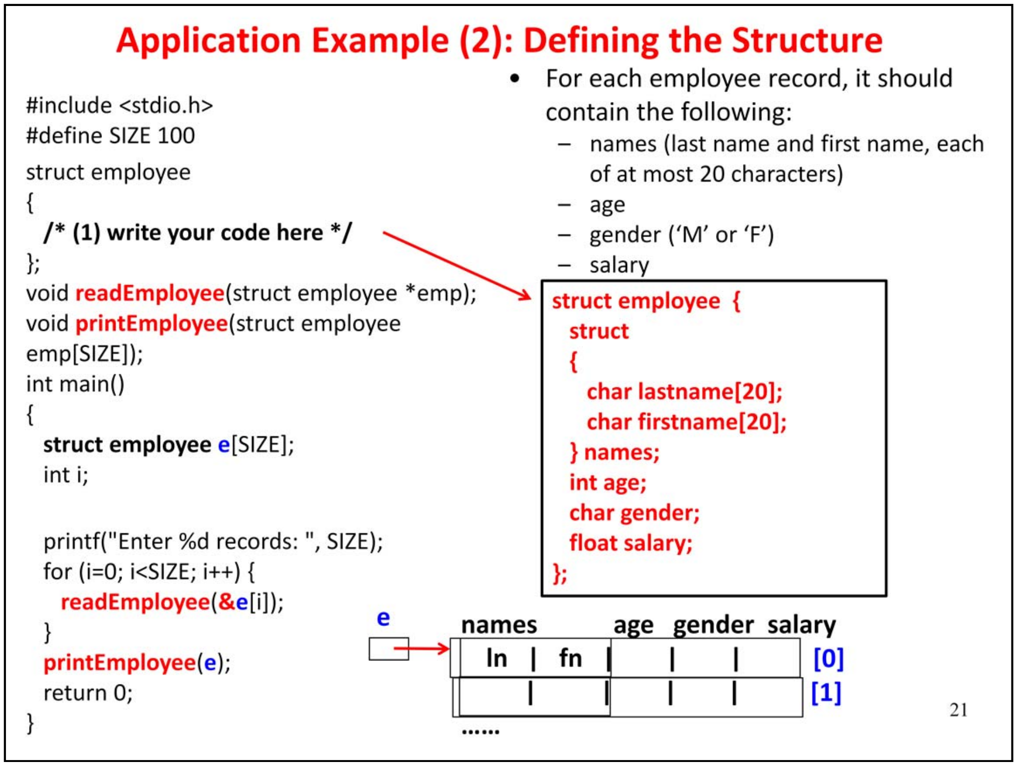
Write the C program & code for (1), (2), (3).

```
#include <stdio.h>
#define SIZE 100
struct employee
{
  /* (1) write your code here */
};
void readEmployee(struct employee *emp);
void printEmployee(struct employee
emp[SIZE]);
int main()
{
  struct employee e[SIZE];
  int i;

  printf("Enter %d records: ", SIZE);
  for (i=0; i<SIZE; i++) {
    readEmployee(&e[i]);
  }
  printEmployee(e);
  return 0;
}
```

```
void  readEmployee(struct employee *emp)
{
  /* (2) write your code here */
}

void  printEmployee(struct employee
emp[SIZE])
{
  /* (3) write your code here */
}
```

Application Example: Program Template

1. The program template for the application is given.



Application Example (2): Defining the Structure

```
#include <stdio.h>
#define SIZE 100
struct employee
{
  /* (1) write your code here */
};
void readEmployee(struct employee *emp);
void printEmployee(struct employee
emp[SIZE]);
int main()
{
  struct employee e[SIZE];
  int i;

  printf("Enter %d records: ", SIZE);
  for (i=0; i<SIZE; i++) {
    readEmployee(&e[i]);
  }
  printEmployee(e);
  return 0;
}
```

- For each employee record, it should contain the following:
  - names (last name and first name, each of at most 20 characters)
  - age
  - gender (‘M’ or ‘F’)
  - salary

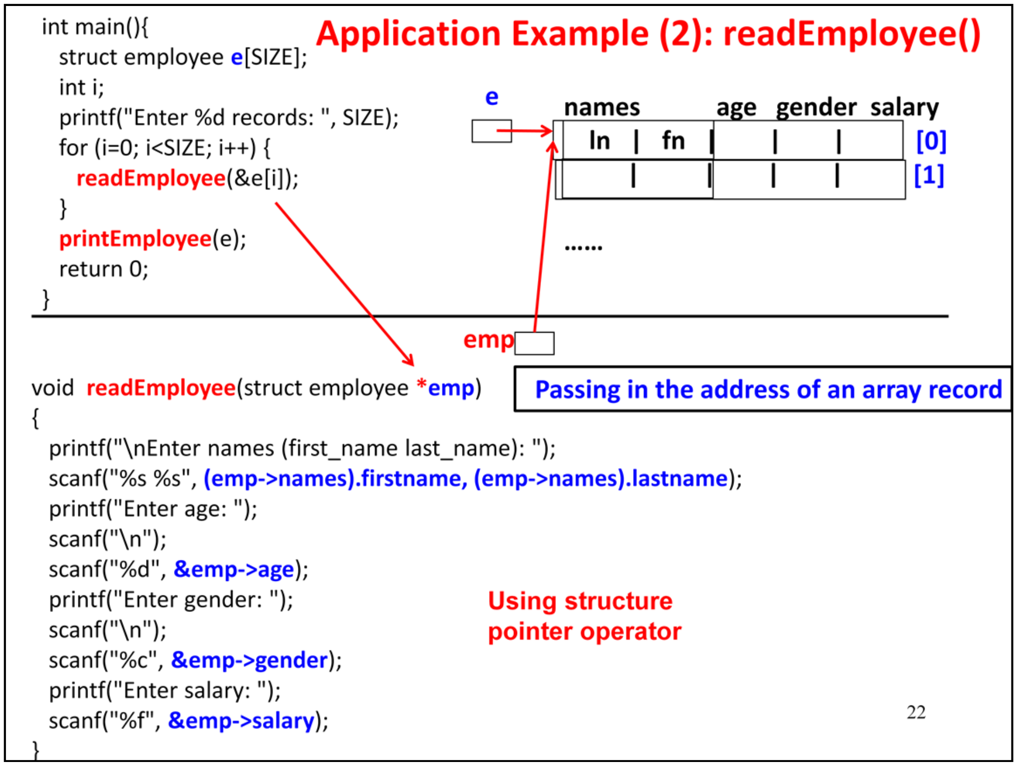
```
struct employee  {
  struct
  {
    char lastname[20];
    char firstname[20];
  } names;
  int age;
  char gender;
  float salary;
};
```

21

Application Example: Defining the Structure

1. The structure for this application is given as follows:

```
struct employee  {
      struct
      {
            char lastname[20];
            char firstname[20];
      } names;
      int age;
      char gender;
      float salary;
};
```



Application Example (2): readEmployee()

```
int main(){
  struct employee e[SIZE];
  int i;
  printf("Enter %d records: ", SIZE);
  for (i=0; i<SIZE; i++) {
    readEmployee(&e[i]);
  }
  printEmployee(e);
  return 0;
}
```

Passing in the address of an array record

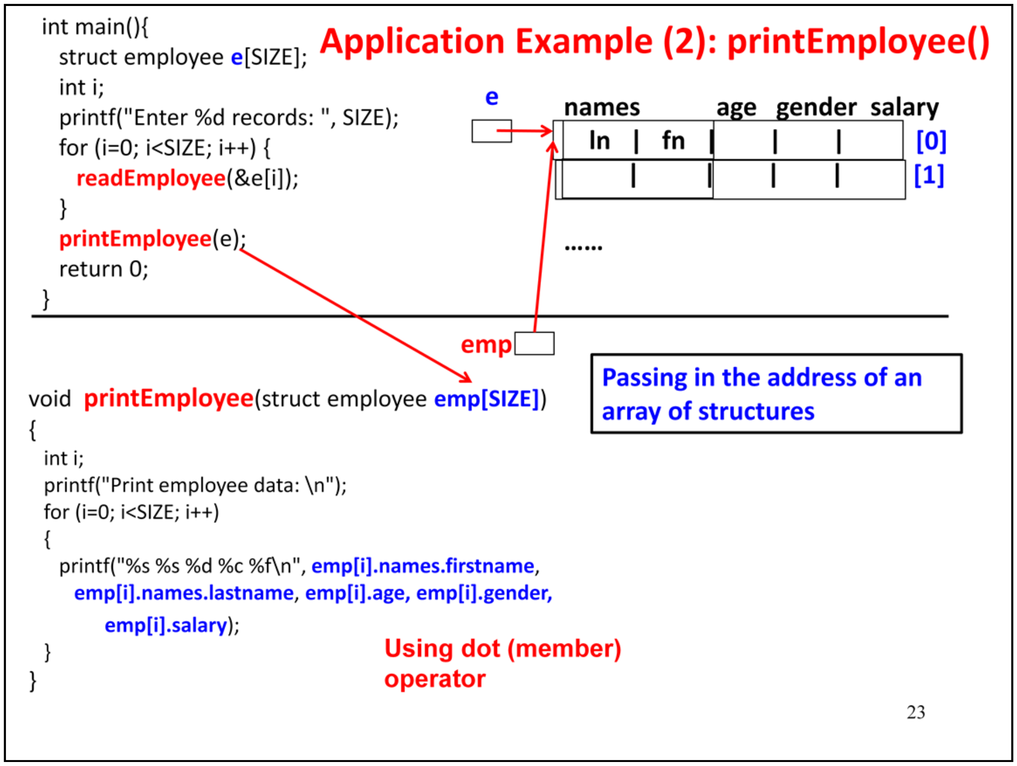
```
void  readEmployee(struct employee *emp)
{
  printf("\nEnter names (first_name last_name): ");
  scanf("%s %s", (emp->names).firstname, (emp->names).lastname);
  printf("Enter age: ");
  scanf("\n");
  scanf("%d", &emp->age);
  printf("Enter gender: ");
  scanf("\n");
  scanf("%c", &emp->gender);
  printf("Enter salary: ");
  scanf("%f", &emp->salary);
}
```

Using structure pointer operator

Application Example: readEmployee()

1. The function readEmployee() is given.

```
void  readEmployee(struct employee *emp)
 {
     printf("\nEnter names (first_name last_name): ");
     scanf("%s %s", (emp->names).firstname, (emp->names).lastname);
     printf("Enter age: ");
     scanf("\n");
     scanf("%d", &emp->age);
     printf("Enter gender: ");
     scanf("\n");
     scanf("%c", &emp->gender);
     printf("Enter salary: ");
     scanf("%f", &emp->salary);
 }
```



Application Example (2): printEmployee()

```
int main(){
  struct employee e[SIZE];
  int i;
  printf("Enter %d records: ", SIZE);
  for (i=0; i<SIZE; i++) {
    readEmployee(&e[i]);
  }
  printEmployee(e);
  return 0;
}
```

```
void  printEmployee(struct employee emp[SIZE])
{
  int i;
  printf("Print employee data: \n");
  for (i=0; i<SIZE; i++)
  {
    printf("%s %s %d %c %f\n", emp[i].names.firstname,
      emp[i].names.lastname, emp[i].age, emp[i].gender,
        emp[i].salary);
  }
}
```

Passing in the address of an array of structures

Using dot (member) operator

23

Application Example: printEmployee()

1. The function printEmployee() is given.

```
void  printEmployee(struct employee emp[SIZE])
 {
       int i;
       printf("Print employee data: \n");
       for (i=0; i<SIZE; i++)
       {
              printf("%s %s %d %c %f\n", emp[i].names.firstname,
              emp[i].names.lastname, emp[i].age, emp[i].gender, emp[i].salary);
       }
 }
```