# 7
# Structures

# Why Learning Structures?

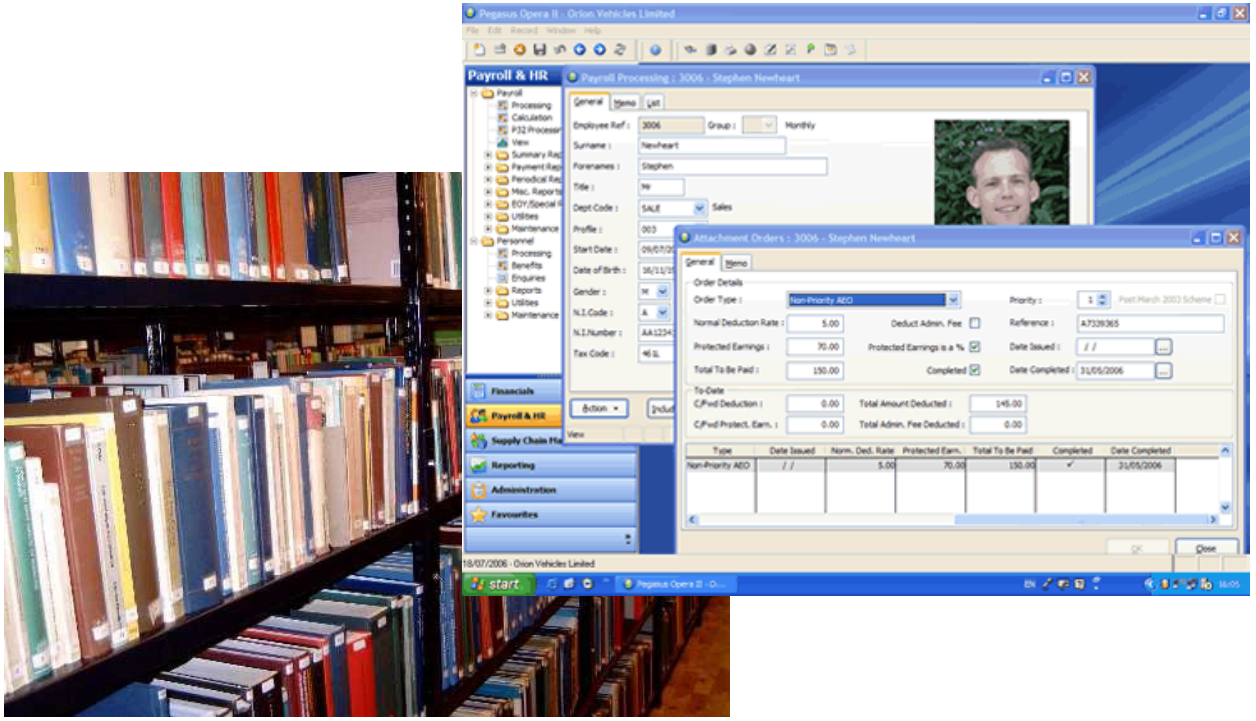
- Arrays are used to store a collection of unrelated data items of the **same** data type.
- C also provides a **data type** called ***structure*** that stores a collection of data items of different data types as a group.
- The individual components of a structure can be of any valid data types.
- In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

# Structures

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

# Records

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




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

# Structures

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

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

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

# 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;
};
```

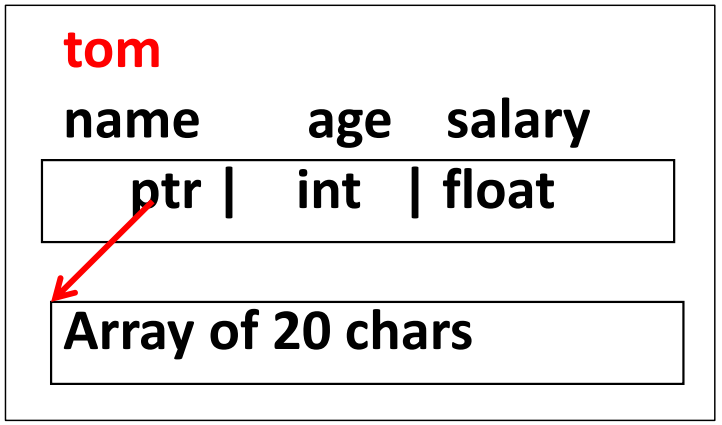
  - **struct**: reserved keyword to introduce a structure
  - **book**: an **optional** **tag name** which follows the keyword struct to name the structure declared.
  - **title, author, value**: the **member** of the structure book.

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

# 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.

# 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.

# Accessing Structure Members

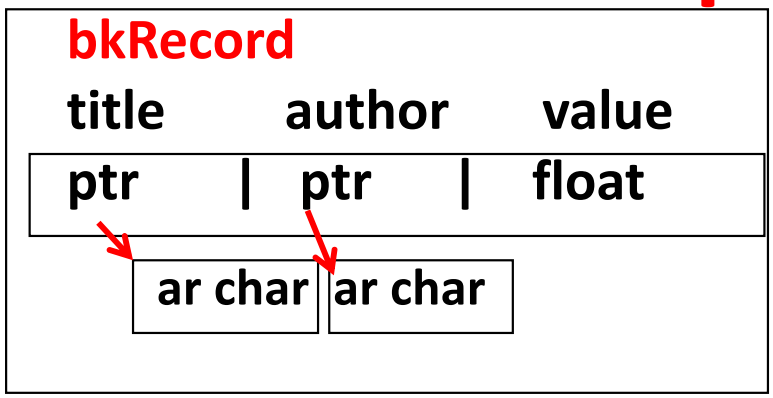
- The notation required to reference the members of a structure is

**structureVariableName.memberName**

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

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

# Structure Declaration & Operation: Example




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

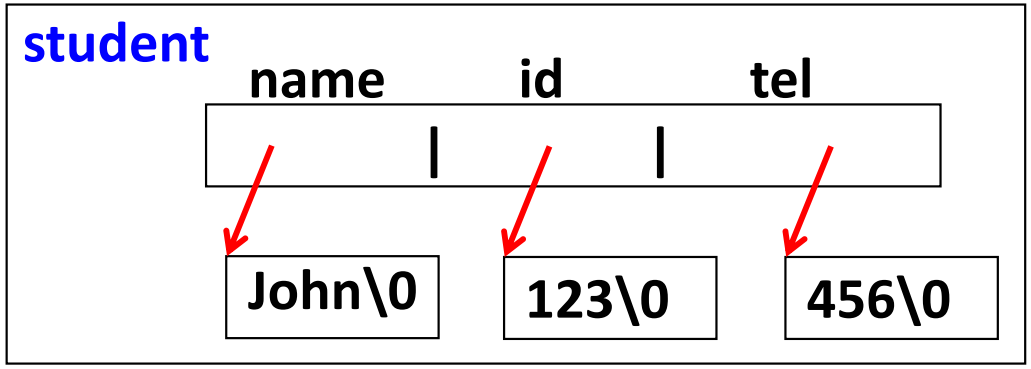
Variable name

**Output**

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

# Structure Variable: Initialization

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




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

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

using . notation

**Output**
John 123 456

# 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;
```

# Structures

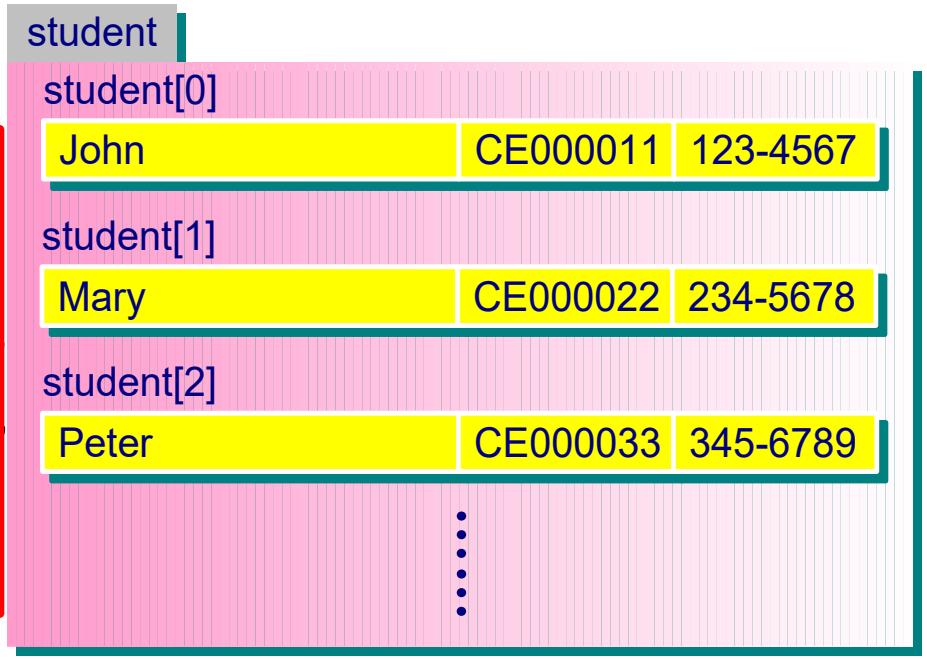
- Structure Declaration, Initialization and Operations
- **Arrays of Structures and Nested Structures**
- Pointers to Structures
- Functions and Structures
- The typedef Construct
- Reading Inputs from Mixed Data Types

# Arrays of Structures

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

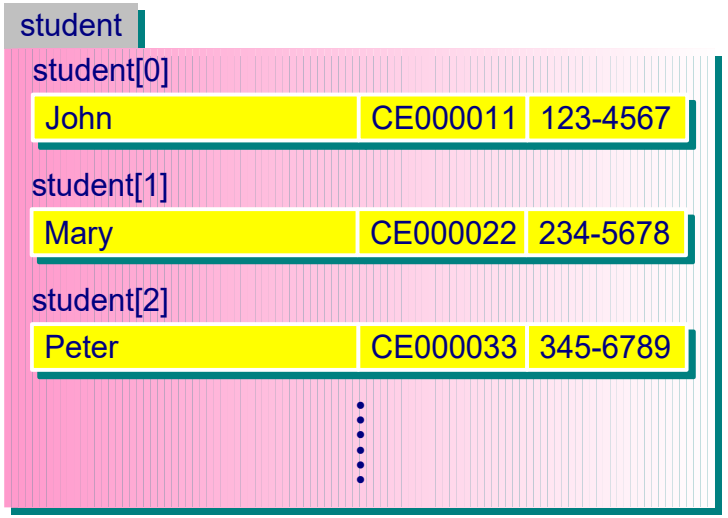
# Arrays of Structures: Declaration & Initialization

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

    // access each structure in array

}
```



# 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

# Nested Structures

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

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

(1) Student info

(2) Course info: SC101

(3) Course info: SC102

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

# Nested Structures

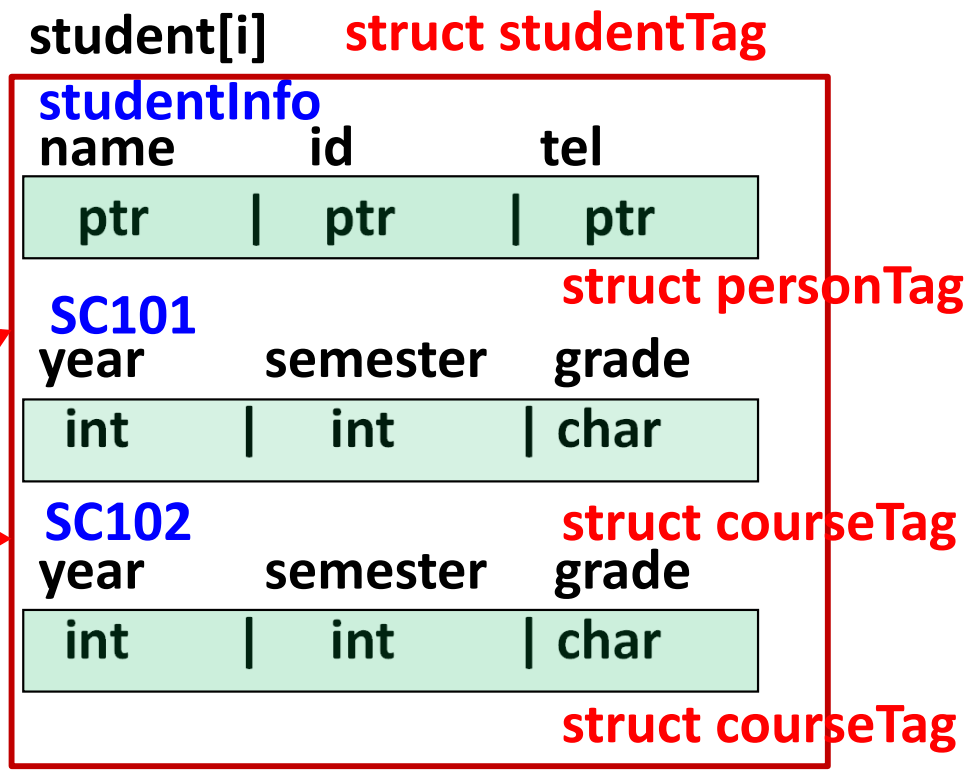
- Alternatively, struct **studentTag** can be defined in a more elegant manner using **nested structures**:

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

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

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

```
struct studentTag student[1000];
```



# Nested Structures: Initialization

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

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

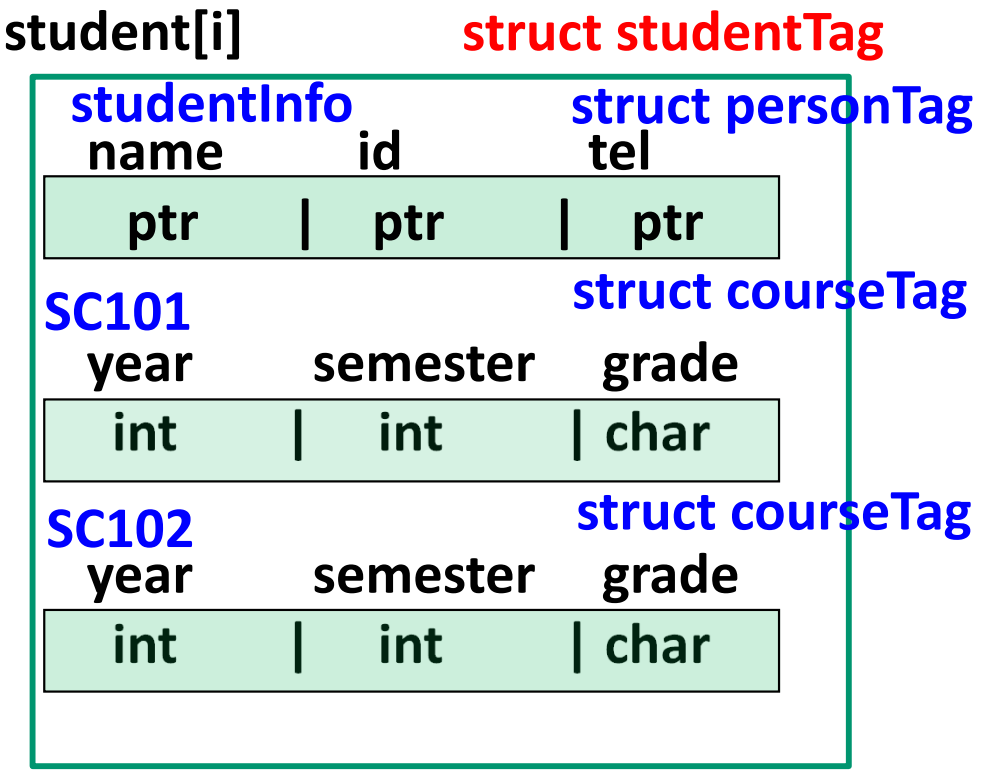
# Nested Structures: Operation

```
/* To print individual elements of the array of structures*/
int i;
for (i=0; i<=2; i++) {
        printf("Name:%s, ID: %s, Tel: %s\n",
        student[i].studentInfo.name,
        student[i].studentInfo.id,
        student[i].studentInfo.tel);

        printf("SC101 in year %d semester %d : %c\n",
        student[i].SC101.year,
        student[i].SC101.semester,
        student[i].SC101.grade);
        printf("SC102 in year %d semester %d : %c\n",
        student[i].SC102.year,
        student[i].SC102.semester,
        student[i].SC102.grade);
}
```

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

# Nested Structures: Notations

- **student[i]** denotes the $i+1^{th}$ array record. It consists of three members: studentInfo, SC101, SC102.
- **student[i].studentInfo** denotes the personal information in the $i+1^{th}$ 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 $i+1^{th}$ record. Each consists of three members: year, semester, grade.



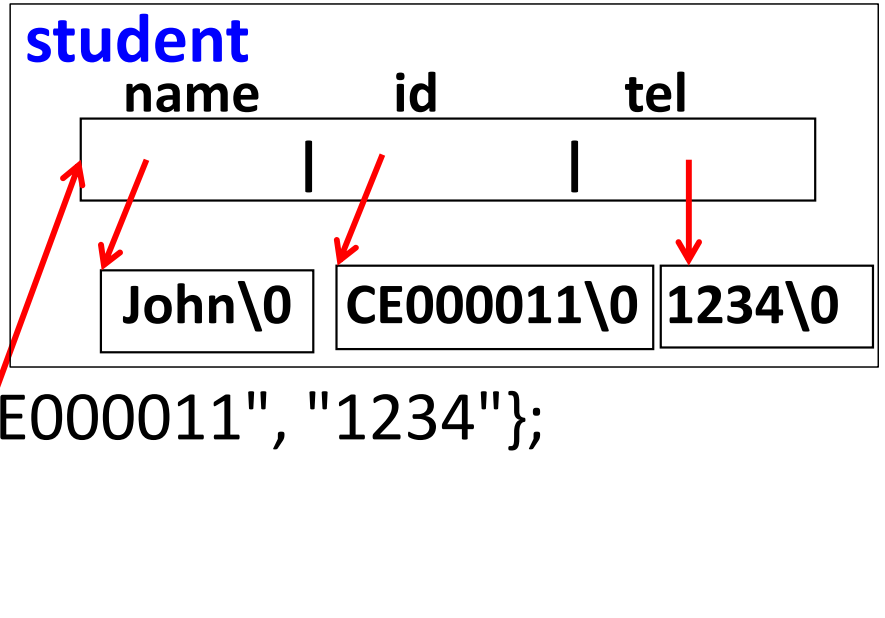
# Structures

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

# Pointers to Structures

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

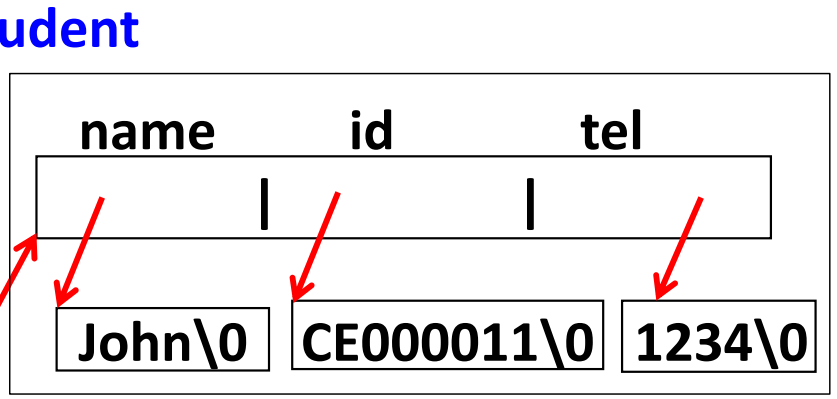
```
/* Using pointer to structure */
struct personTag {
    char name[40], id[20], tel[20];
} ;
struct personTag student = {"John", "CE000011", "1234"};

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




**Analogy:**

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

# 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 */
```

# Pointers to Structures: Operation

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

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

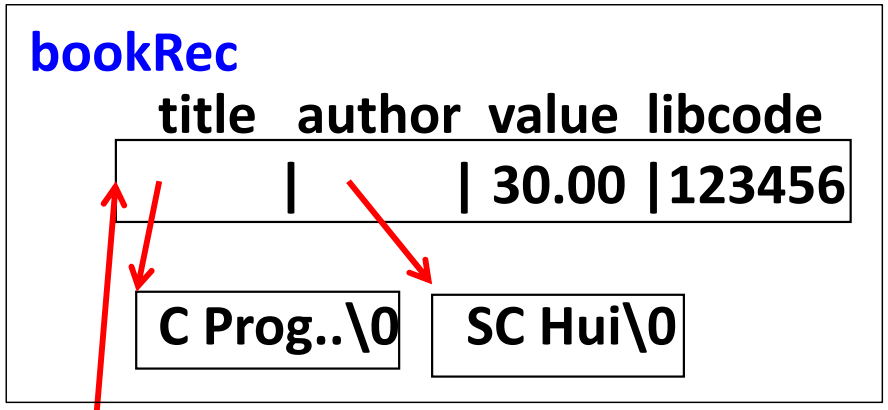
- Instead, we can use the **structure pointer operator (->)** for a pointer pointing to a structure:

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

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

# Pointers to Structures: Example

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




**Output**

The book C Programming (123456) by SC Hui: $30.00.

# Structures

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

# Functions and Structures

- It is often necessary to pass structure information to a function. In C, there are **four** ways to pass structure information to a function:

  1. Passing **structure members** as arguments using call by value, or call by reference;
  2. Passing **structures** as arguments;
  3. Passing **pointers to structures** as arguments; and
  4. Passing by **returning structures**.

- Basically, parameter passing between functions using structure is **similar** to passing data of other basic data types such as int, float, etc.

# Passing Structure Members as Arguments

```
#include <stdio.h>
float sum(float, float);
struct account {
   char    bank[20];
   float   current;
   float   saving;
};
int main( )
{
   struct account john={"OCBC Bank",1000.43, 4000.87};
   printf("The account has a total of %.2f.\n",
            sum(john.current, john.saving));      // pass by value
   return 0;
}
float sum(float x, float y)
{
   return (x+y);
}
```

**Output**

The account has a total of 5001.30.

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

# Passing Structure as Argument: Call by Value

```
#include <stdio.h>
struct account{
    char  bank[20];
    float  current;
    float  saving;
};
float sum(struct account);          /* argument - structure */
int main( )
{
    struct account john = {"OCBC Bank", 1000.43, 4000.87};
    printf("The account has a total of %.2f.\n", sum(john));   // pass by value
    return 0;
}
float sum( struct account money)
{
    return(money.current + money.saving);
    /* not money->current */
}
```

**Output**

The account has a total of 5001.30.

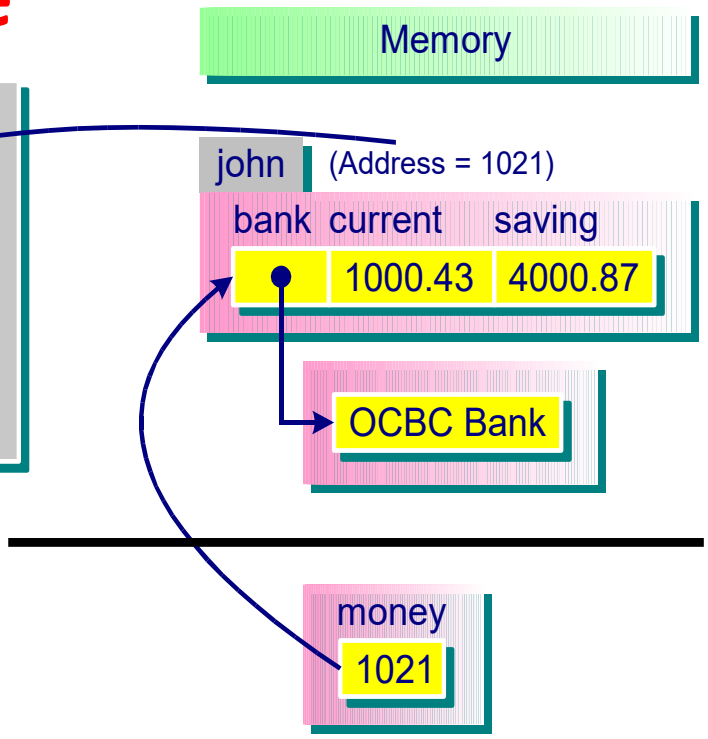
- **Call by value**
- **struct account money** is used as parameter

# Passing Structure Address as Argument: Call by Reference

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

```
.....
main(void)
{
   struct account john = {"OCBC Bank",
      1000.43, 4000.87};
   printf(" ..... ", sum(&john));
   .....
}
```

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




```
int main()
{
    struct account john={"OCBC Bank",1000.43, 4000.87};
    printf("The account has a total of %.2f.\n",
            sum(&john));
    return 0;
}
```

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

- **Call by reference**

# Passing by Returning a Structure

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

**Output**

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

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

# Structures

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

# The typedef Construct

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

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

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

```
typedef struct date Date;
```

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

- Then, variables can be declared either as

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

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

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

**No tag name – date**

**Define variables**

# The typedef Construct: Example

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

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

**Output**

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

# Structures

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

# Common Error on Reading Input Data

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

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

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

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

**Intended Input/Output:**

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

The number read is 1234

Correct (y/n)? ***y***

your reply : y

**Can the program compile correctly?**

**Can the program run as intended?**

# Common Error on Reading Input Data: Problem

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

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

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

**When the program runs:**

**Output**

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

The number read is 1234

**Correct (y/n)? your reply :**

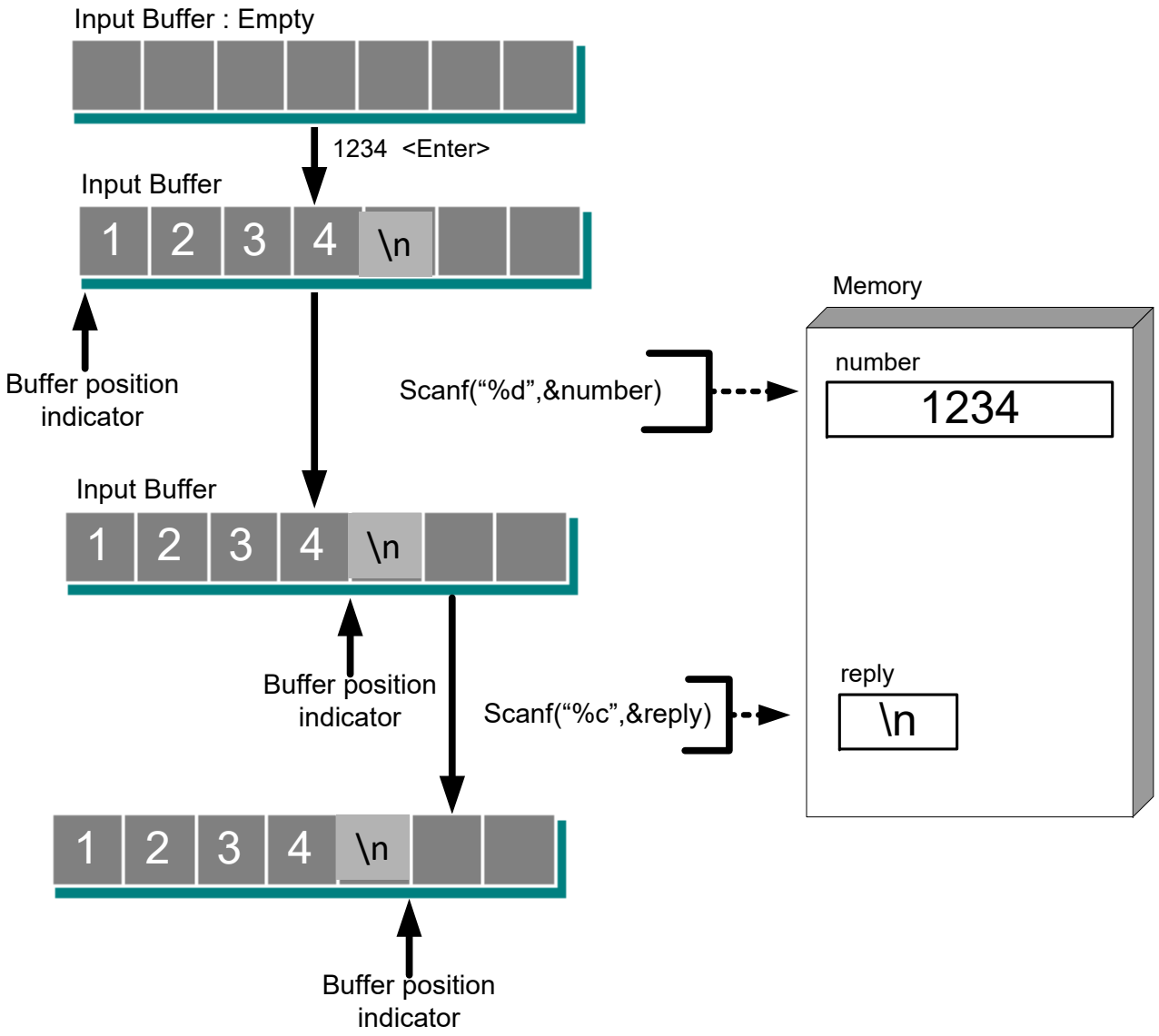
**// an error here**

**// the reply is not read**

**Can the program compile correctly?**

**Can the program run as intended?**

# Common Error on Reading Input Data: Reason




***Reason*:**

There is a **hidden character** '\n' entered when you type 1234 <Enter>

# Reading Mixed Data Input

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

# Reading Mixed Data Input: Solution

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

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

**2:**

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

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

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

# Reading Mixed Data Input in APAS

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

# Thank You!