Developing Technical Software

Week 3:
User data types, Self-referential structures

3.1 Structure definition

A structure is a generalized array with different data types.

- Arrays are able to manage the variables with the same data type. However, a structure is a *generalized* array, designed to manage variables of different data types. In the following structure definition:

```
struct  employee  {
                  char  name[30];
                  int  day,  month,  year;
                  char  street[30];
                  char  city[20];
                  char  state[7];
                  int  zip[4];
                  int  phone_num[12];
                };
```

The ***structure type*** is struct employee, which has 7 **members**:

char name[30], **int** day, month, year, **char** street[30], **char** city[20], **char** state[7], **int** zip[4] and int phone_num[12].

- The above structure definition has created a *new data type*, which can be used to declare structure variables. For example, the instruction

```
struct employee a;   /* int a; */
```

has declared that a is a variable of type struct employee.

- The general form of a structure definition is

```
struct  employee {
                  char  name[30];
                  int  day,  month,  year;
                  char  street[30];
                  char  city[20];
                  char  state[7];
                  int  zip[4];
                  int  phone_num[12];
                } a;
```

- The following structure variable declaration:

```
struct  employee  a,  dept[52],  *ePtr;
```

defines (i)   a *variable*, a, of type struct employee;

(ii) an array, dept[52], with 52 elements of type struct employee;

(iii) a pointer, ePtr, pointing to a structure variable of type struct employee.

- The equivalent definition is of the form:

```
struct  employee  {
                    char  name[30];
                    int  day,  month,  year;
                    char  street[30];
                    char  city[20];
                    char  state[7];
                    int  zip[4];
                    int  phone_num[12];
               }  a,  dept[52],  *ePtr;
```

3.2 Accessing structure members

For the following defined `struct card`:

```
struct card {
               char *face;
               char *suit;
             } aCard, *aPtr;
```

Two operators may be used to access its members:

(i) **the dot operator**

```
printf(“%s”, aCard.suit);
```

(ii) **the structure pointer operator**

```
printf(“%s”, aPtr->suit);
```

Remark: The expression

```
aPtr->suit
```

is equivalent to

```
(*aPtr).suit
```

Structure initialisation

```
struct personal_data {
        char surname[20];
        char initials[4];
        char address1[20];
        char town[20];
        char post_code[8];
        char phone[12];
  };
int main()
{    struct personal_data a = {
    “Camberwell”,
    “N.D.”,
    “2 North Road”,
    “Treebridge”,
    “TD9 12XT”,
    “0567 2237”};
    return 0;
   }
```

```
printf(“%s”, a.surname);
```

Example 3.2: Assign the values to the structure members

```
#include<stdio.h>
struct  personal_data  {
               int  day;
               int  month;
               int  year;
                 }  a;
int main()
{
     a.day  =  11;
     a.month  =  7;
     a.year  =  1956;
     printf(“%4d\n%4d\n%4d\n”,  a.day,  a.month,  a.year);
      return 0;
}
```

Using the structure members

```
#include  <stdio.h>
struct  card  {      char *face;
                     char *suit;
                    };

int  main()
{        struct  card  a;
       struct  card    *aPtr;
       a.face  =  “Ace  of  ”;
       a.suit  =  “Spades”;
       aPtr  =  &a;
       printf(“%s%s\n”,  a.face,  a.suit);
       printf(“%s%s\n”,  aPtr->face,  (*aPtr).suit);
       return  0;
}
```

```
Ace of Spades
Ace of Spades
```

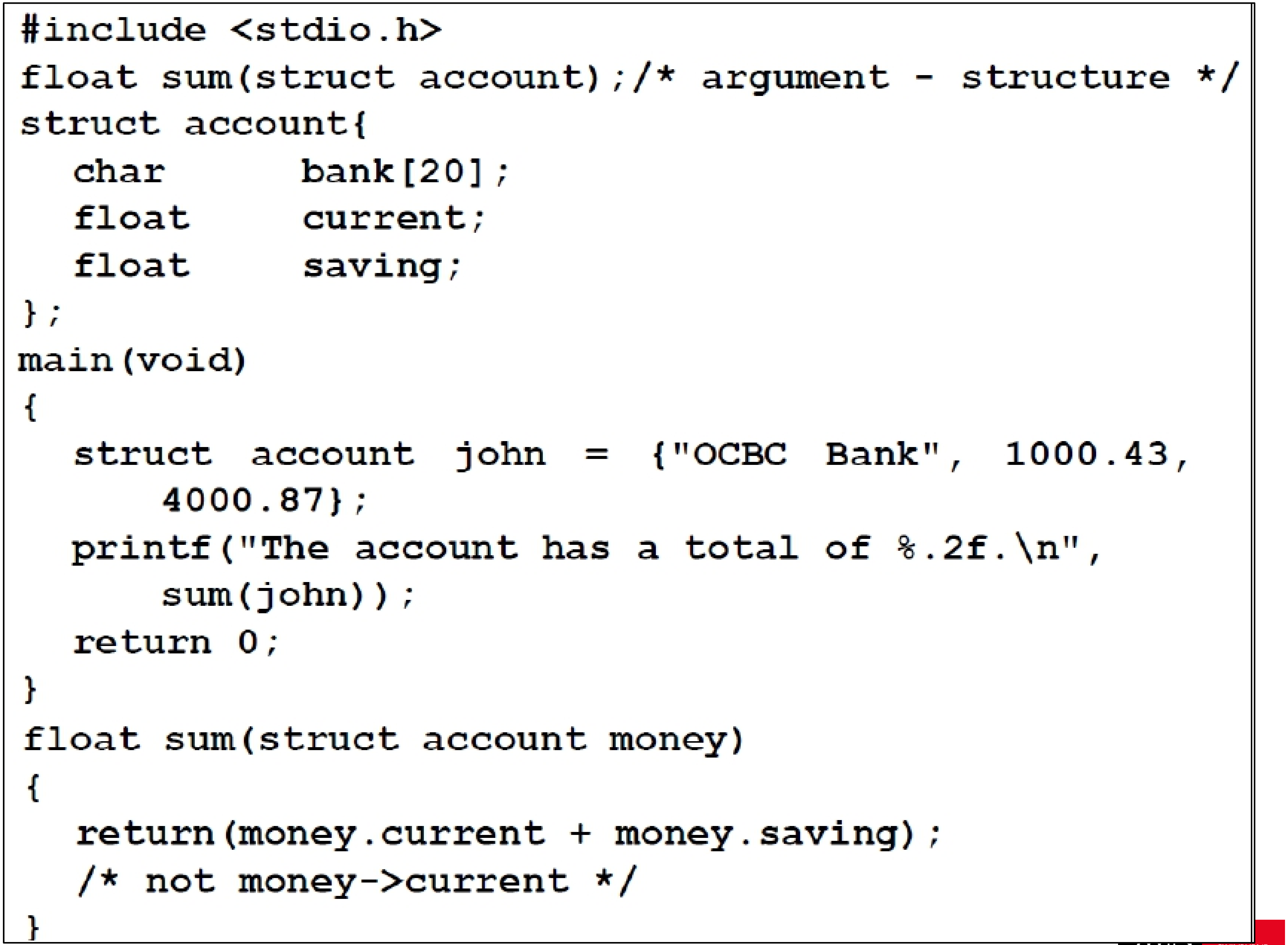
Passing structures as parameters

```
#include<stdio.h>
struct personal_data {
        int day;
        int month;
        int year;
         }a;
int abc(struct personal_data);
int main()
{ int e;
  a.day = 11;
  a.month = 7;
  a.year = 1956;
  printf(“%4d\n%4d\n%4d\n”, a.day, a.month, a.year);
  e=abc(a);
  printf(“%4d”, e);
  return 0;
}
int abc(struct personal_data b)
{
  int g;
  return g=(b.day);
}
```

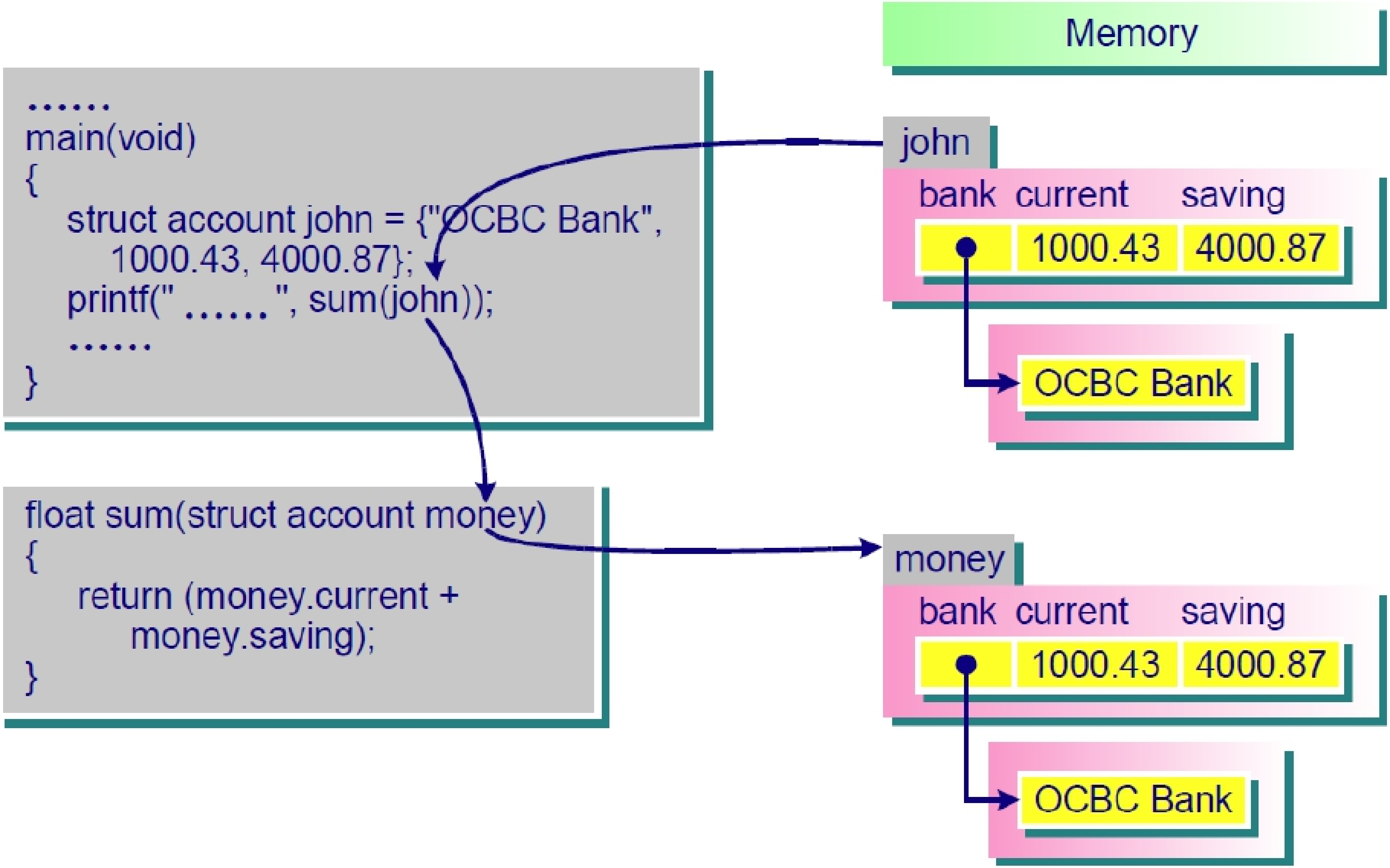
```
  11
   7
1956
  11
```

Using structures with functions

- Functions and structures
- Passing a structure as an argument
- Passing individual members of a structure as argument
- Passing the structure address



```
#include <stdio.h>
float sum(struct account);/* argument - structure */
struct account{
   char       bank[20];
   float      current;
   float      saving;
};
main(void)
{
   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);
   /* not money->current */
}
```



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

```
float sum(struct account money)
{
     return (money.current +
         money.saving);
}
```

Memory

john

bank	current	saving
●	1000.43	4000.87

OCBC Bank

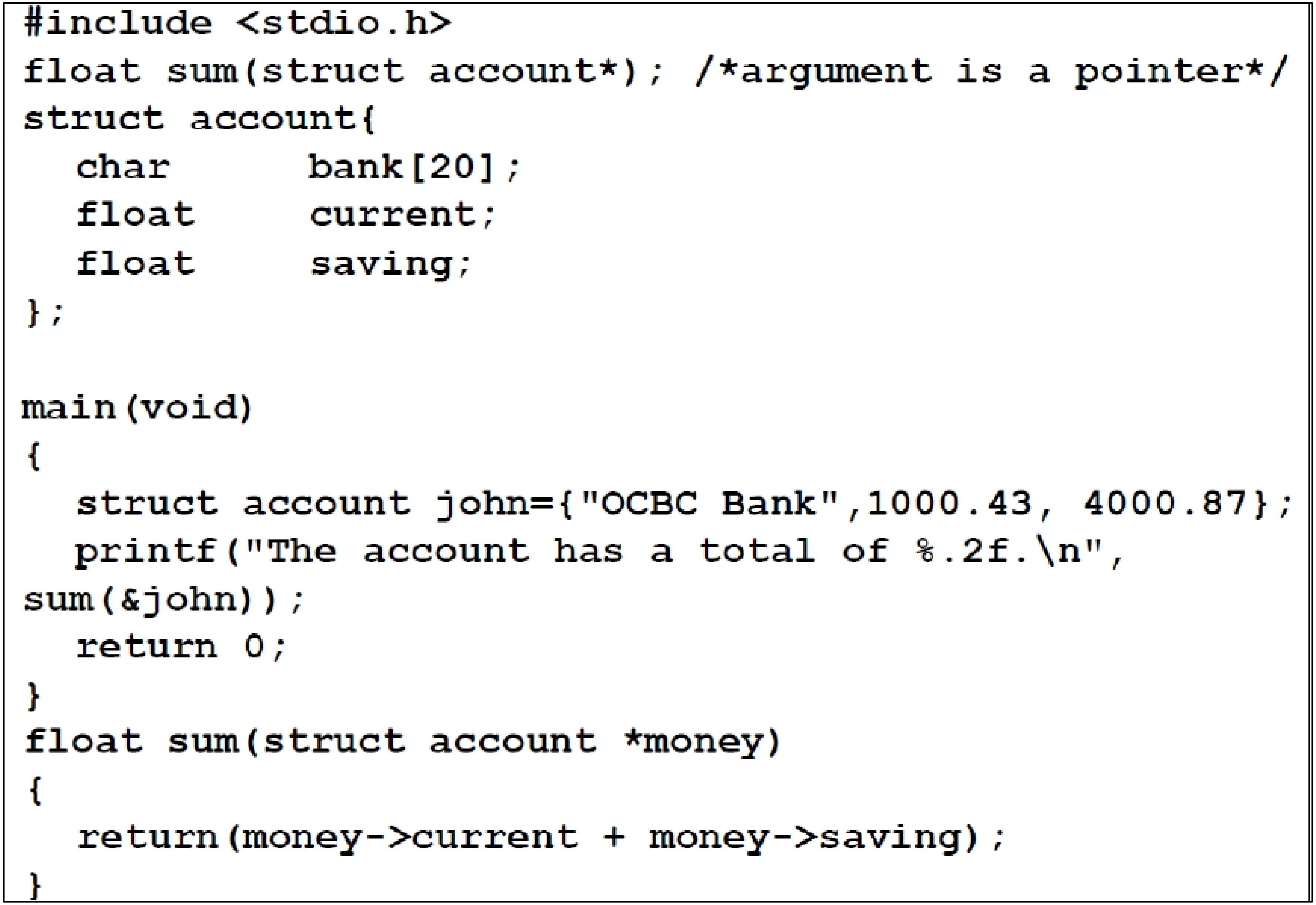
money

bank	current	saving
●	1000.43	4000.87

OCBC Bank

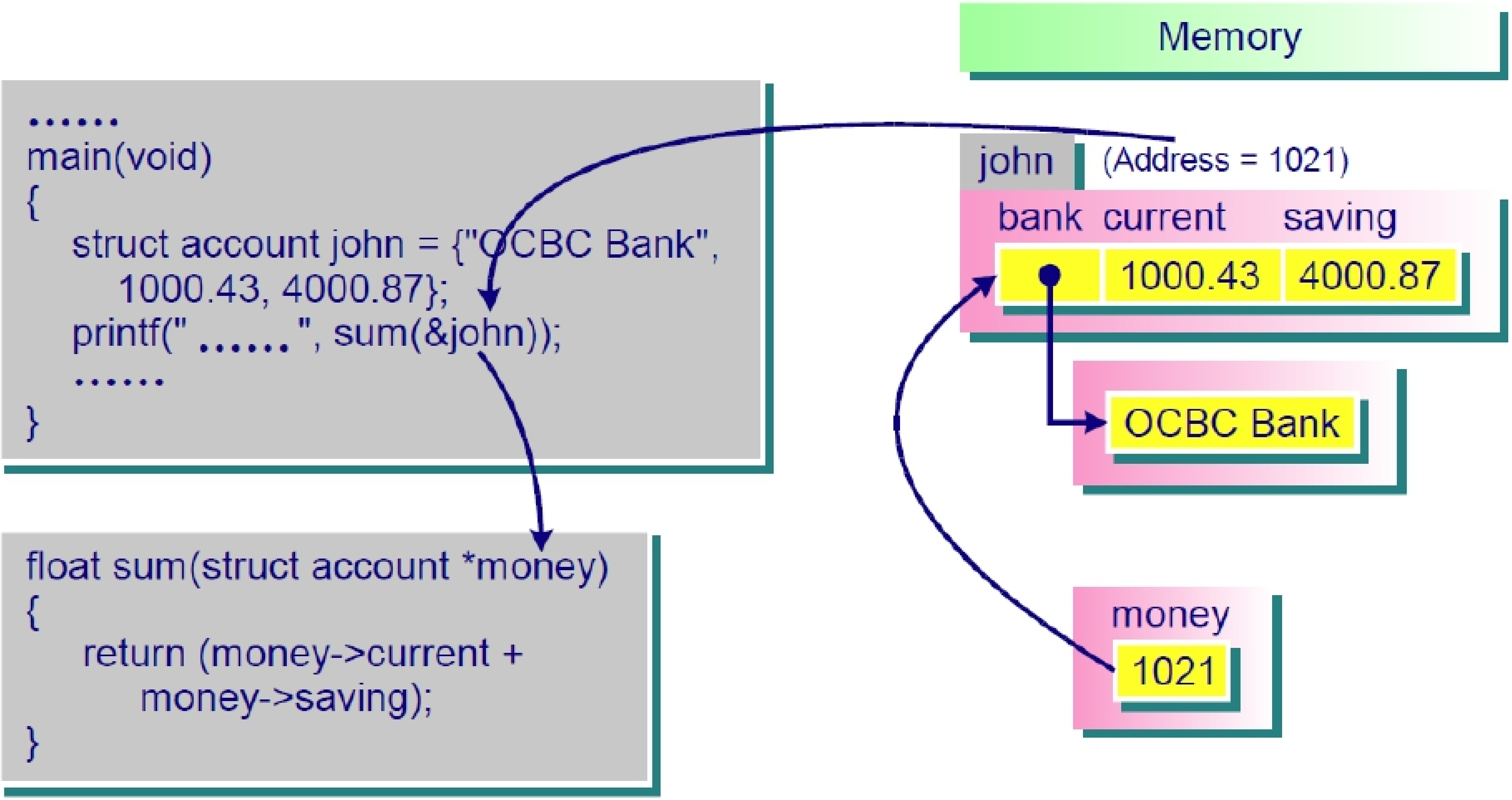
Using pointers to structures

- Pointers to structures are easier to manipulate than structures themselves
- In older C implementation, structure is passed as an argument to a function using pointer to structure
- Many advanced data structures require pointers to structures



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

main(void)
{
   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);
}
```

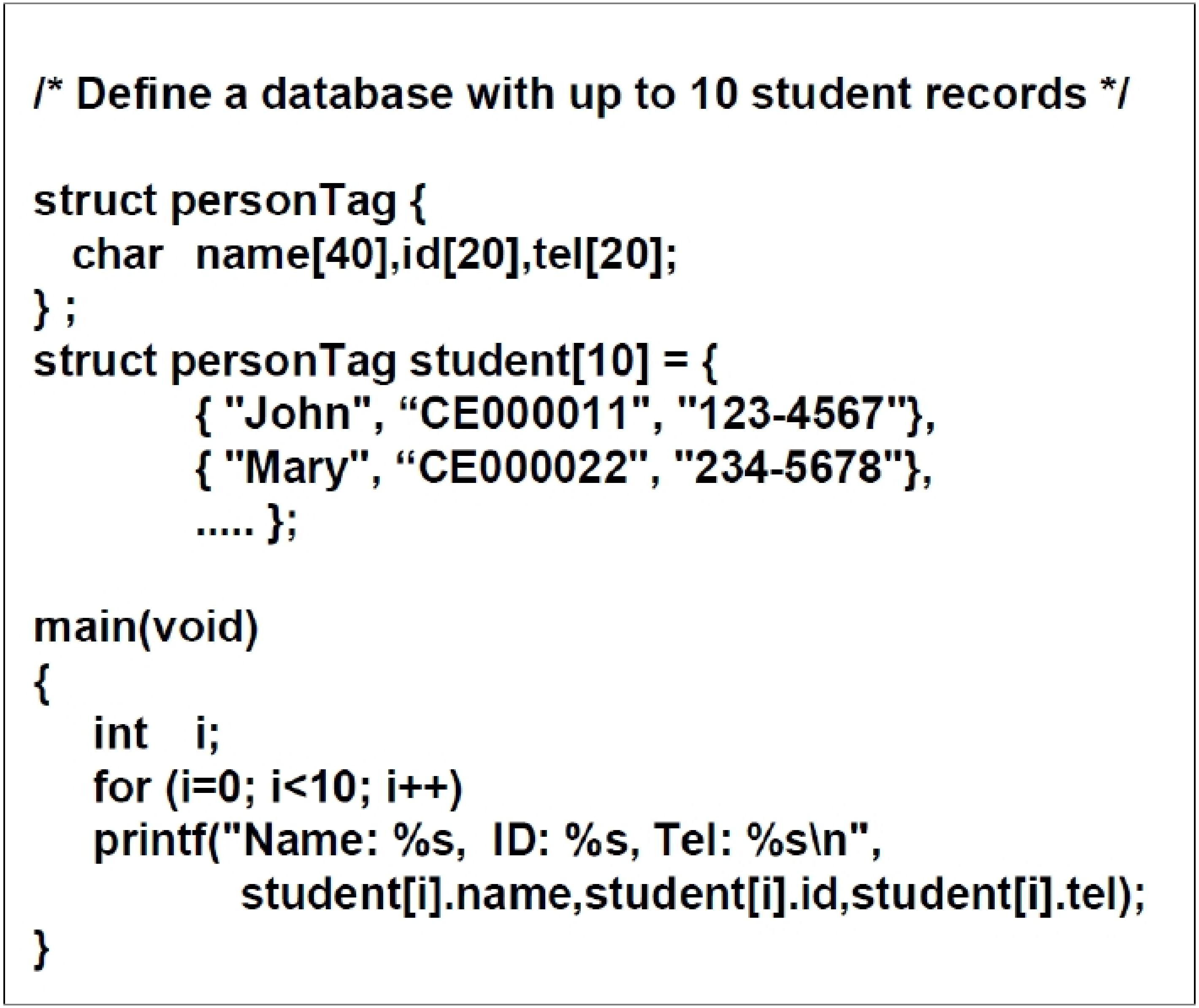


```
......
main(void)
{
    struct account john = {"OCBC Bank",
        1000.43, 4000.87};
    printf(" ......", sum(&john));
    ......
}
```

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

3.4 Arrays of structures

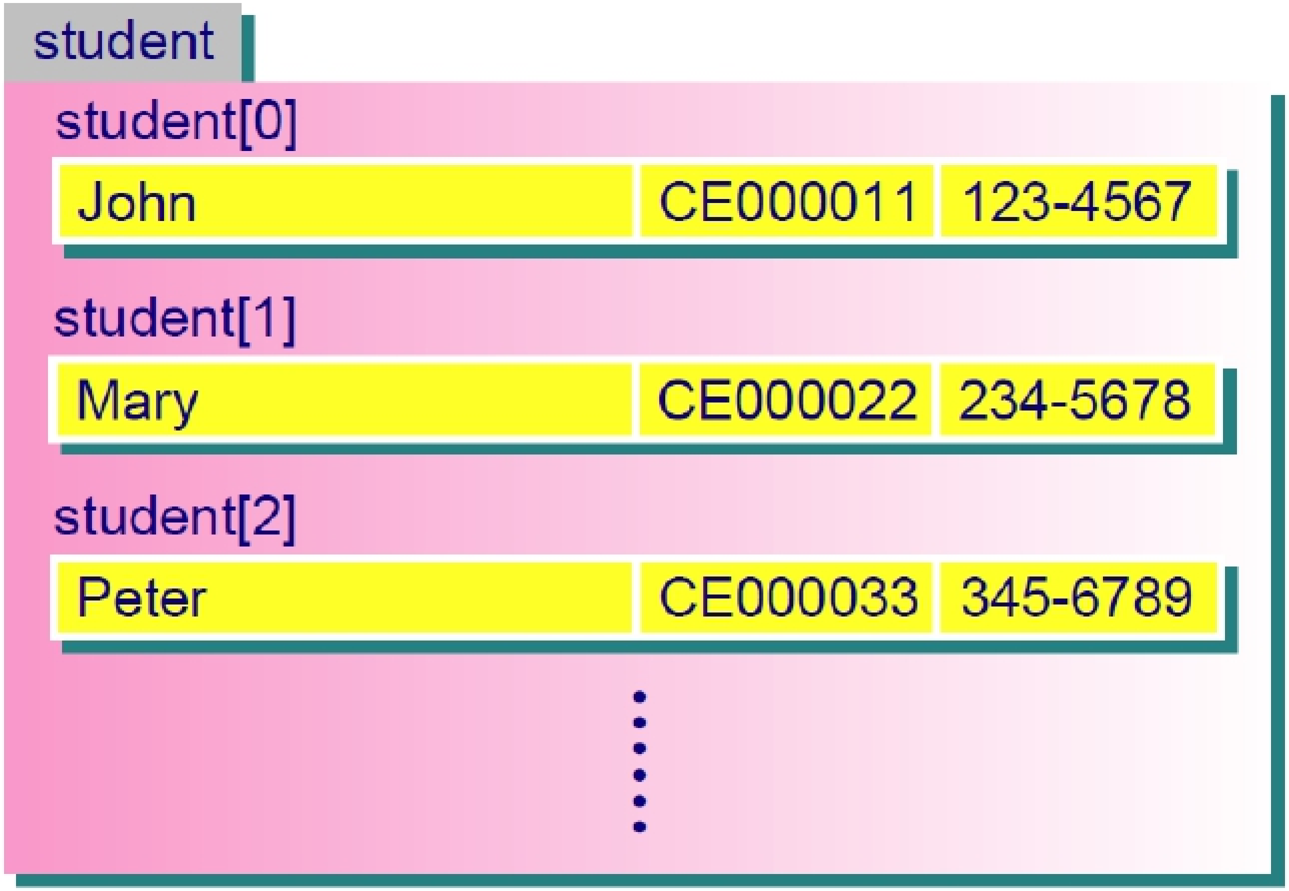
- A structure variable can be seen as a *record*, e.g., personal information record of the name, dob, home address, phone number …
- When structure variables of the same type are grouped together, we have a database of that structure type.
- One can create a database by defining an array of certain structure type.



```
/* 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"},
            ..... };

main(void)
{
    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]

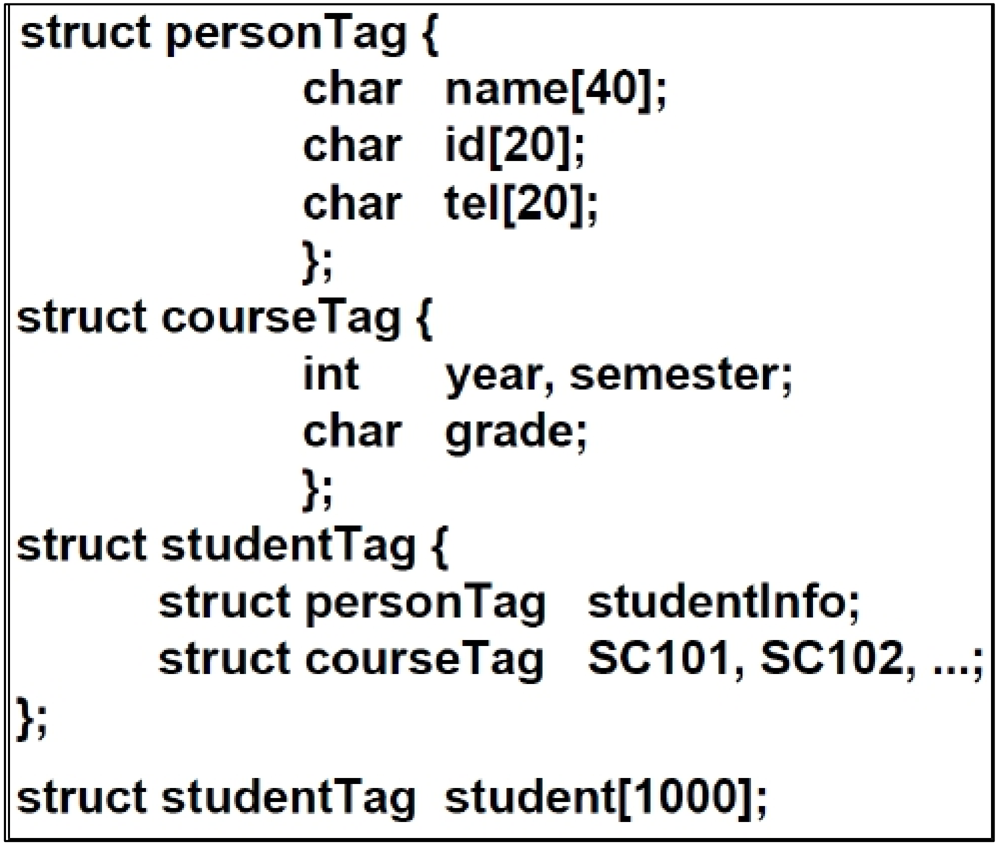
Mary	CE000022	234-5678

student[2]

Peter	CE000033	345-6789

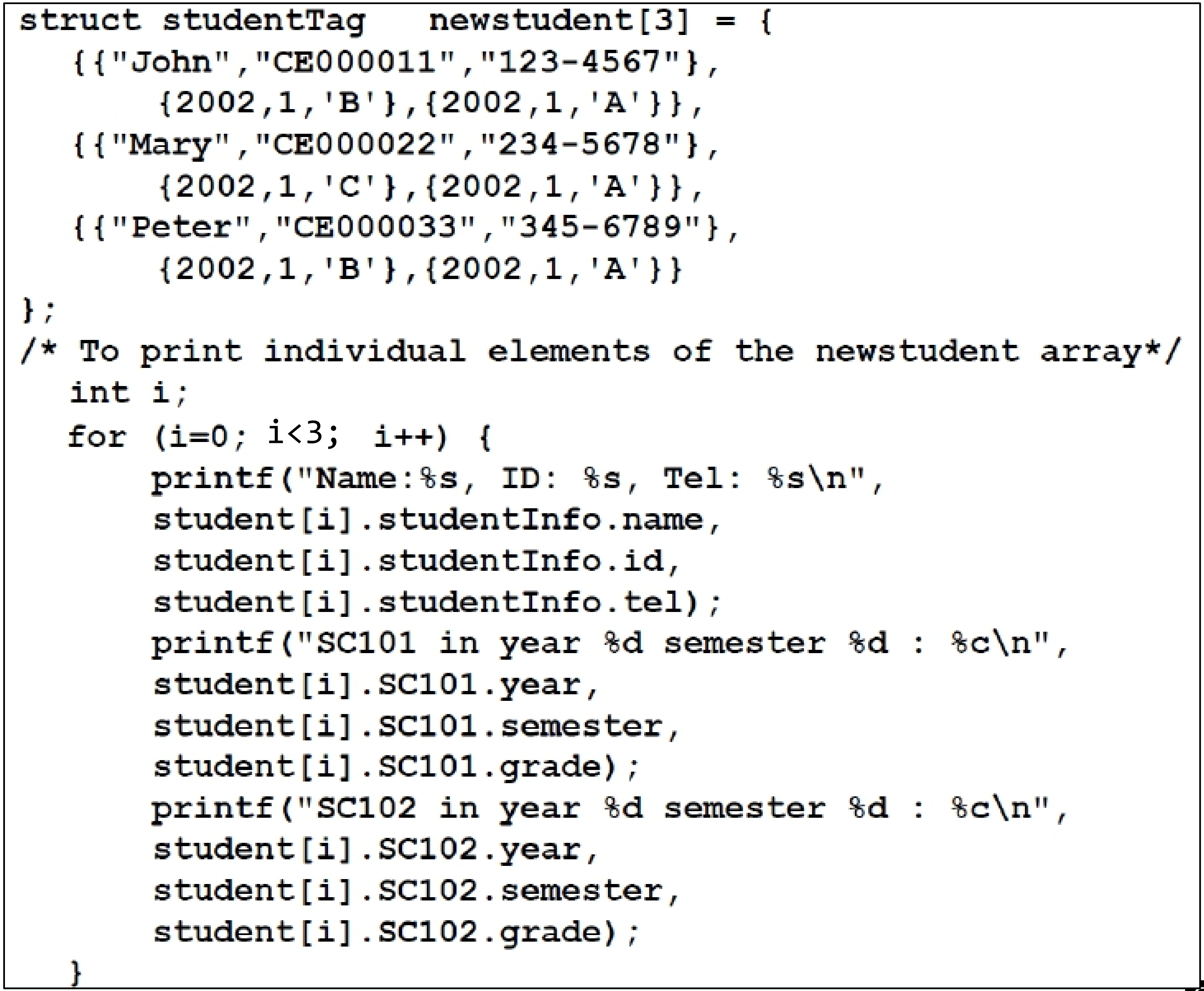
3.5 Nested Structures

- Astructure can also be included in other structure
- How to keep record of the subject history of a student?



```
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, ...;
};
struct studentTag  student[1000];
```

- `student` is the complete database
- `student[i]` denotes the *i+1th* record
- `student[i].studentInfo` denotes the personal information of the *i+1th* record
- `student[i].studentInfo.name` denotes the student name in that record
- `student[i].studentInfo.name[j]` denotes a single character value



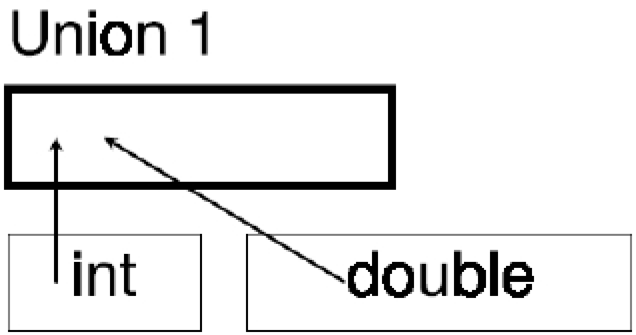
```
struct studentTag   newstudent[3] = {
   {{"John","CE000011","123-4567"},
       {2002,1,'B'},{2002,1,'A'}},
   {{"Mary","CE000022","234-5678"},
       {2002,1,'C'},{2002,1,'A'}},
   {{"Peter","CE000033","345-6789"},
       {2002,1,'B'},{2002,1,'A'}}
};
/* To print individual elements of the newstudent array*/
   int i;
   for (i=0; i<3;  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);
   }
```

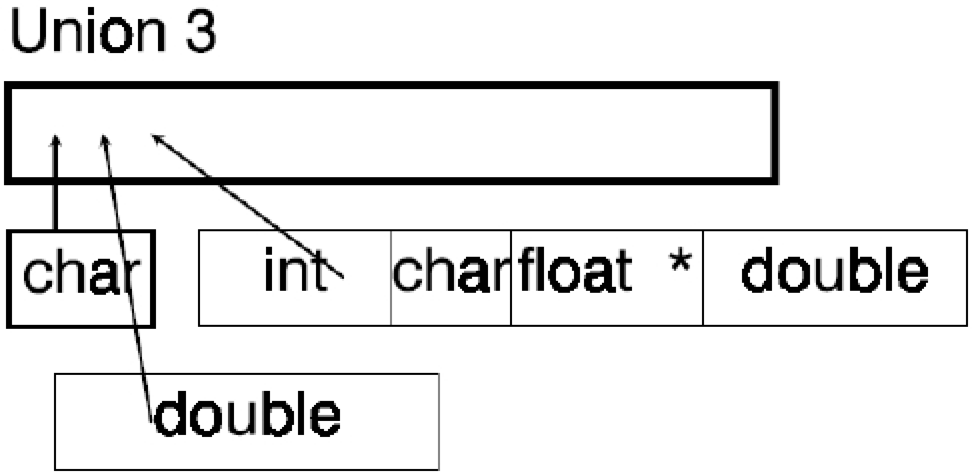
3.6 Unions

- Union is similar to structure in the sense that union also contains members of different data types and sizes.

- Union can only hold at most **one** of its members at a time.
- Members are overlaid in the storage allocated for the union.
- Compiler allocates sufficient storage to accommodate the largest of the union members.

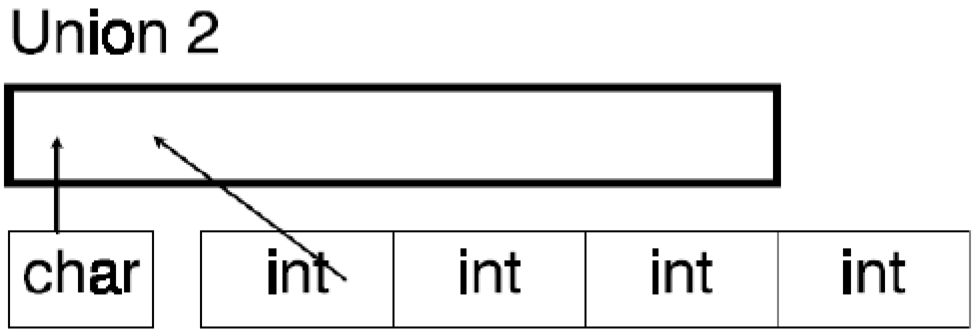
- Union 1 holds and represents either an integer or a double





- Union 3 holds and represents one char, one structure or one double

- Union 2 holds and represents either a char or an array of 4 integers



- It is the programmer’s responsibility to keep track of the data type currently being stored in a union

```
union  word  {
                    int  digit;
                    double  flnum;
                    char  letter;
                 }  guess;
guess.digit  =  3;                /* 3 stored in guess, 4 bytes used */
guess.flnum  =  2.3;          /* 3 cleared, 2.3 stored, 8 bytes used */
guess.letter  =  ‘h’;        /* 2.3 cleared, ‘h’ stored, 1 byte used */
```

- Notation for declaring and accessing members are identical to that of structures except the keyword struct is replaced by union in the declaration.

```
unionVariable.memberName
unionPointerVariable->memberName
```

3.7 User-defined data types (typedef)

- The typedef allows users to define new data types that are equivalent to existing data type. Once a user-defined data type has been established, new variables, arrays, structures can be declared in terms of this data type.

The use of typedef

```
typedef  int  age;
age  male,  female;  /*  int  male,  female;  */
typedef  float  height[100];
height  boy,  girl;  /*  float  boy[100],  girl[100];  */
```

- The typedef is particularly convenient for defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced.

A user-defined structure type declares structures

```
typedef struct {
      int month;
      int day;
      int year;
     } record;
     record oldcustomer, newcustomer;
/*  The above structure definitions are equivalent to
    the following ones */
struct record {
       int month;
       int day;
       int year;
      } oldcustomer, newcustomer;
```

Self-referential structures

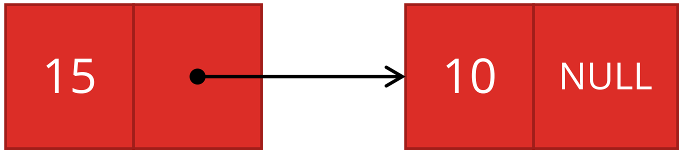
- a self-referential structure contains a pointer member that points to a structure of the same structure type.
- The following definition defines a type, struct node.

```
struct  node {
      int  data;
      struct node  *nextPtr;

};
```

Self-referential structures

- Self-referential structures can be *linked* together to form useful data structures, e.g. linked lists, queues, stack, etc.
- The following figure illustrates two self-referential structure objects to form a list.



NULL pointer

- A NULL pointer is placed in the linked member of the second self-referential structure, to indicate the end of a data.
- Not setting the link in the last node of a list to NULL will lead to runtime errors.