Chap 2. Arrays and Structures (1)

Contents

2.1 Arrays

2.2 Dynamically Allocated Array

2.3 Structures

2.4 Polynomials

2.5 Sparse Matrices

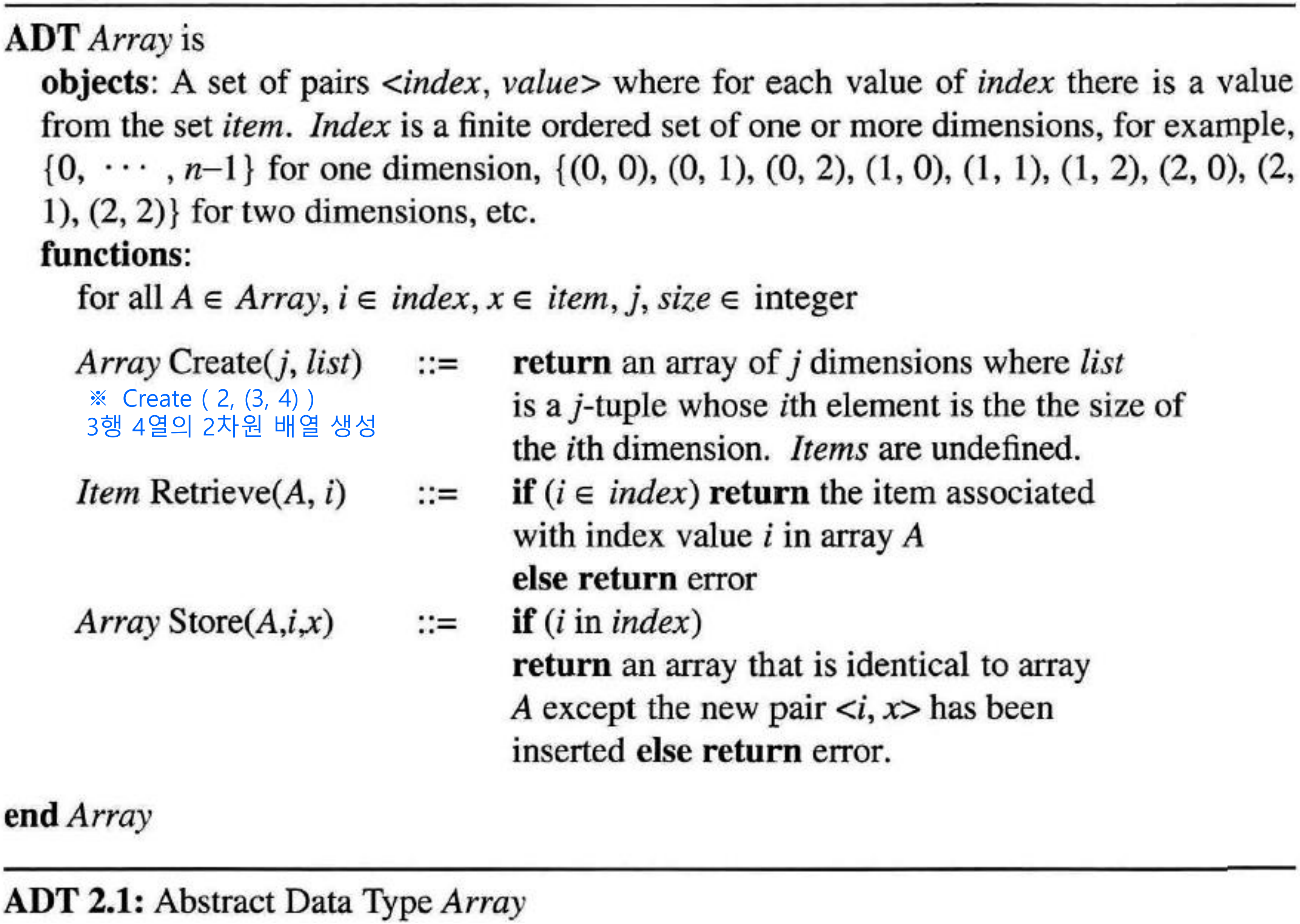
2.1 Arrays – three perspectives

- A consecutive set of memory locations
  - emphasis on implementation issues
  - not always true

- A set of pairs, <*index*, *value*>
  - set of *mappings* or *correspondence* between index and values
  - $array : i \rightarrow a_i$

- ADT
  - more concerned with the operations that can be performed on an array

2.1.1. The Abstract Data Type



ADT *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, ⋯ , *n*–1} for one dimension, {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.

functions:

for all *A* ∈ *Array*, *i* ∈ *index*, *x* ∈ *item*, *j*, *size* ∈ integer

Array Create(*j*, *list*) ::= **return** an array of *j* dimensions where *list* is a *j*-tuple whose *i*th element is the the size of the *i*th dimension. *Items* are undefined.

※ Create (2, (3, 4))
3행 4열의 2차원 배열 생성

Item Retrieve(*A*, *i*) ::= **if** (*i* ∈ *index*) **return** the item associated with index value *i* in array *A* **else return** error

Array Store(*A*,*i*,*x*) ::= **if** (*i* in *index*) **return** an array that is identical to array *A* except the new pair <*i*, *x*> has been inserted **else return** error.

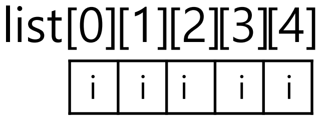
end *Array*

ADT 2.1: Abstract Data Type *Array*

2.1.2 Arrays in C

- one-dimensional array

int list[5];

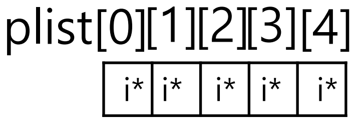


list[0][1][2][3][4]

i	i	i	i	i

※ i stands for int
i* stands for int pointer

int *plist[5];



plist[0][1][2][3][4]

i*	i*	i*	i*	i*

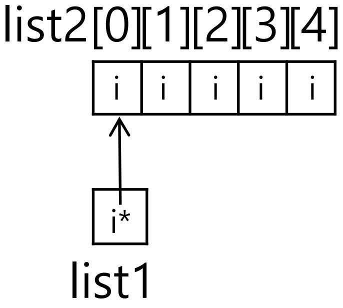
Variable	Memory address
list[0]	base address = α
list[1]	α + sizeof(int)
list[2]	α + 2·sizeof(int)
list[3]	α + 3·sizeof(int)
list[4]	α + 4·sizeof(int)

- interpretations of pointers: list1, list2

  int *list1, list2[5];

  list1 = list2;

  variable constant



```
list2 == &list2[0]
list2 + i == &list2[i]
*(list2+i) == list2[i]


list1 == &list2[0]
list1 + i == &list2[i]
*(list1+i) == list2[i]
```

```
#include <stdio.h>
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
void main(void)
{
   int i;
   for (i = 0; i < MAX_SIZE; i++)
      input[i] = i;
   answer = sum(input, MAX_SIZE);
   printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
   int i;
   float tempsum = 0;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

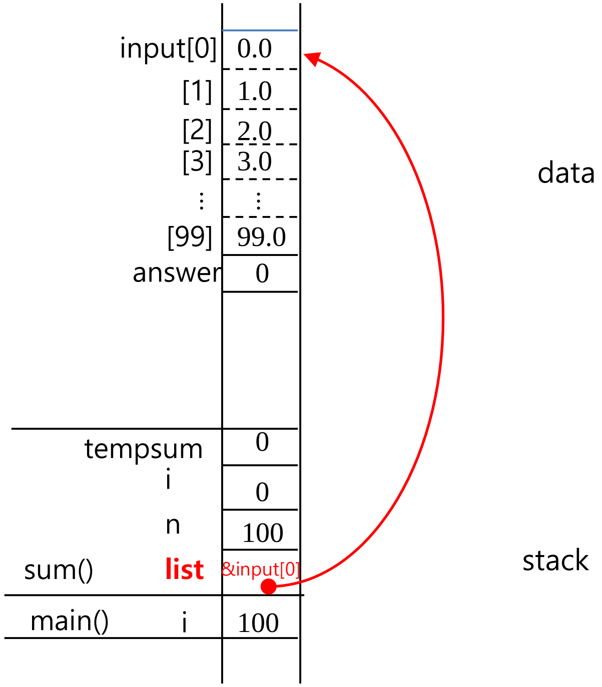
&input[0]

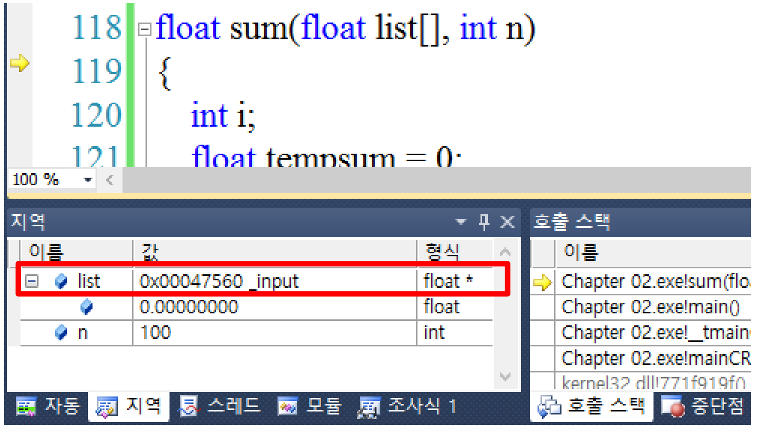
→ array parameter

float * list → pointer parameter

*(list+i)

Program 2.1: Example array program

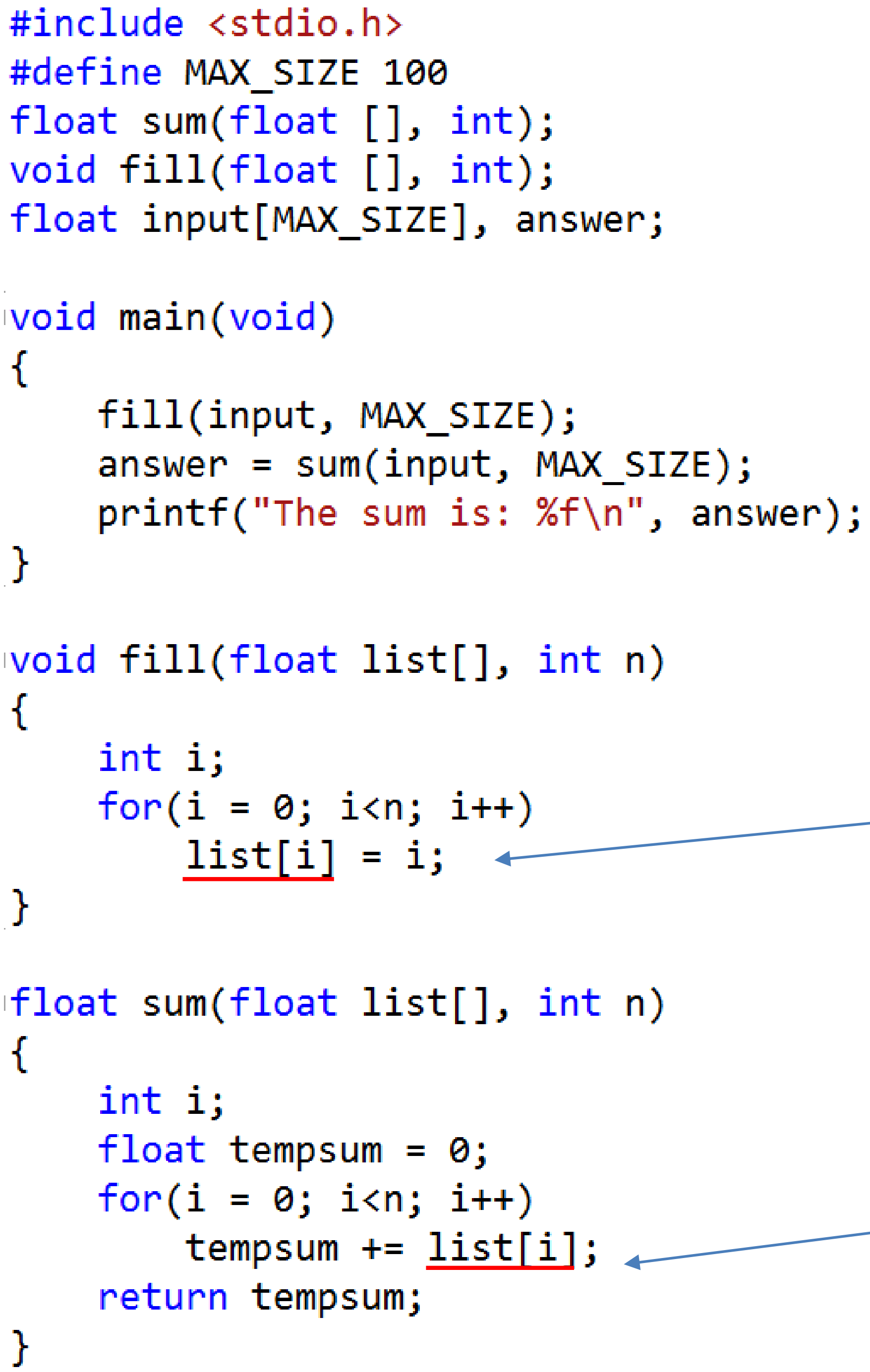




```
float sum(float list[], int n)
{
   int i;
   float tempsum = 0;
```

이름	값	형식
⊟ list	0x00047560 _input	float *
	0.00000000	float
n	100	int

이름
Chapter 02.exe!sum(flo
Chapter 02.exe!main()
Chapter 02.exe!_tmain
Chapter 02.exe!mainCR
kernel32.dll!771f919f0



```c
#include <stdio.h>
#define MAX_SIZE 100
float sum(float [], int);
void fill(float [], int);
float input[MAX_SIZE], answer;

void main(void)
{
    fill(input, MAX_SIZE);
    answer = sum(input, MAX_SIZE);
    printf("The sum is: %f\n", answer);
}

void fill(float list[], int n)
{
    int i;
    for(i = 0; i<n; i++)
        list[i] = i;
}

float sum(float list[], int n)
{
    int i;
    float tempsum = 0;
    for(i = 0; i<n; i++)
        tempsum += list[i];
    return tempsum;
}
```

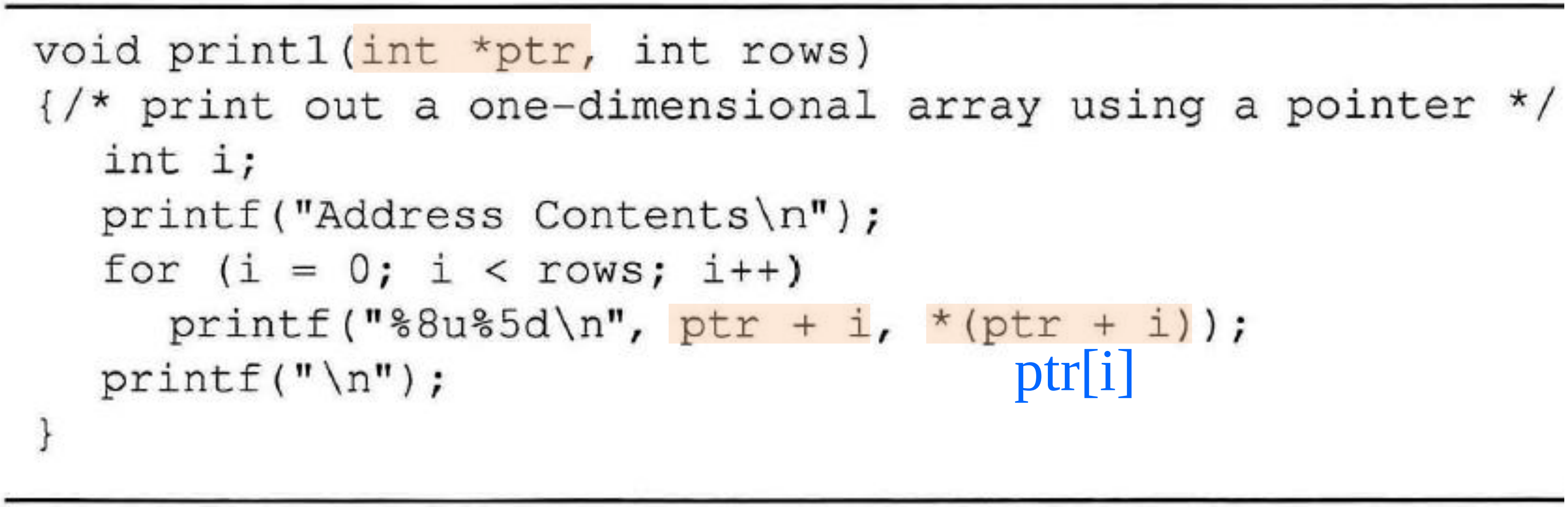
the value produced on the right-hand side is stored in the location (*list*+*i*)

a dereference takes place the value pointed at by *(list*+*i)* is returned

- In C, *array parameters have their values altered*, despite the fact that the parameter passing is done using *call-by-value*.

int one[] = {0, 1, 2, 3, 4};

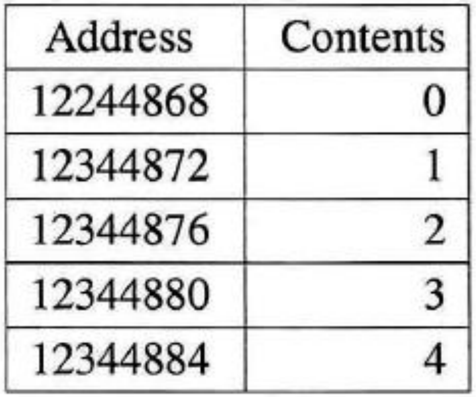
print1(&one[0], 5);



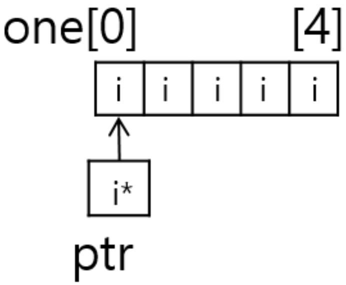
```
void print1(int *ptr, int rows)
{/* print out a one-dimensional array using a pointer */
   int i;
   printf("Address Contents\n");
   for (i = 0; i < rows; i++)
      printf("%8u%5d\n", ptr + i, *(ptr + i));
   printf("\n");
}
```

ptr[i]

Program 2.2: One-dimensional array accessed by address



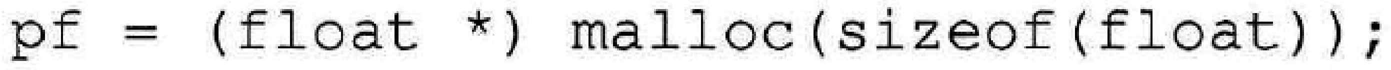
Address	Contents
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4



Assumption: sizeof(int) == 4

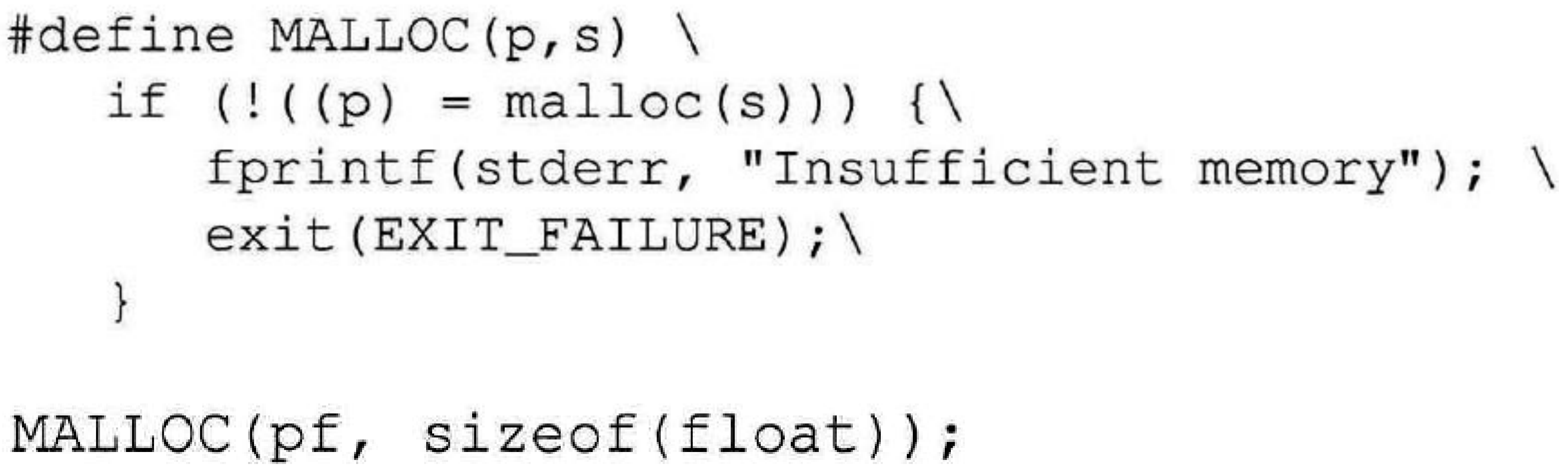
2.2 Dynamically Allocated Arrays

2.2.1 One-dimensional Arrays



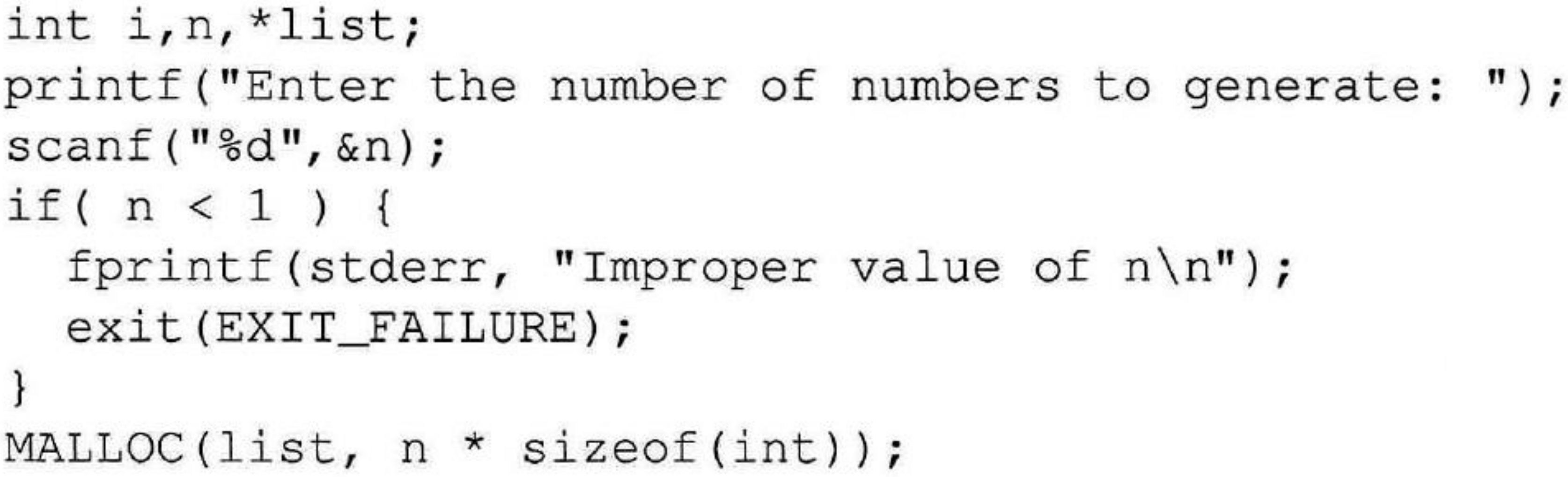
```
pf = (float *) malloc(sizeof(float));
```

can be replaced by

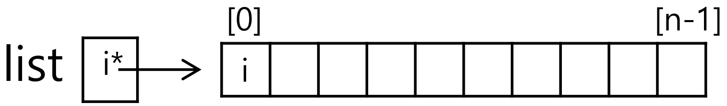


```
#define MALLOC(p,s) \
   if (!((p) = malloc(s))) {\
      fprintf(stderr, "Insufficient memory"); \
      exit(EXIT_FAILURE);\
   }

MALLOC(pf, sizeof(float));
```

- Change the first few lines of *main* of Program 1.4 to:



```
int i,n,*list;
printf("Enter the number of numbers to generate: ");
scanf("%d",&n);
if( n < 1 ) {
  fprintf(stderr, "Improper value of n\n");
  exit(EXIT_FAILURE);
}
MALLOC(list, n * sizeof(int));
```



2.2.2 Two-Dimensional Arrays

- A multidimensional array in C
  - *Array-of-arrays* representation

int x[3][5];

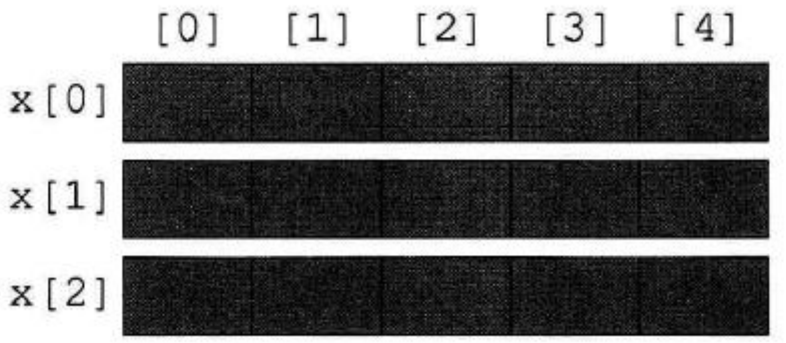
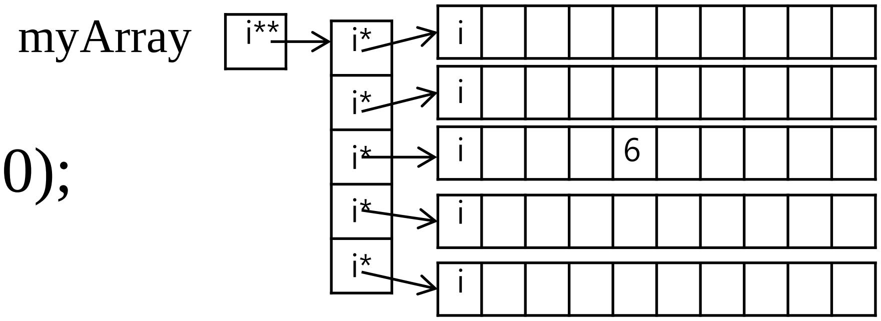


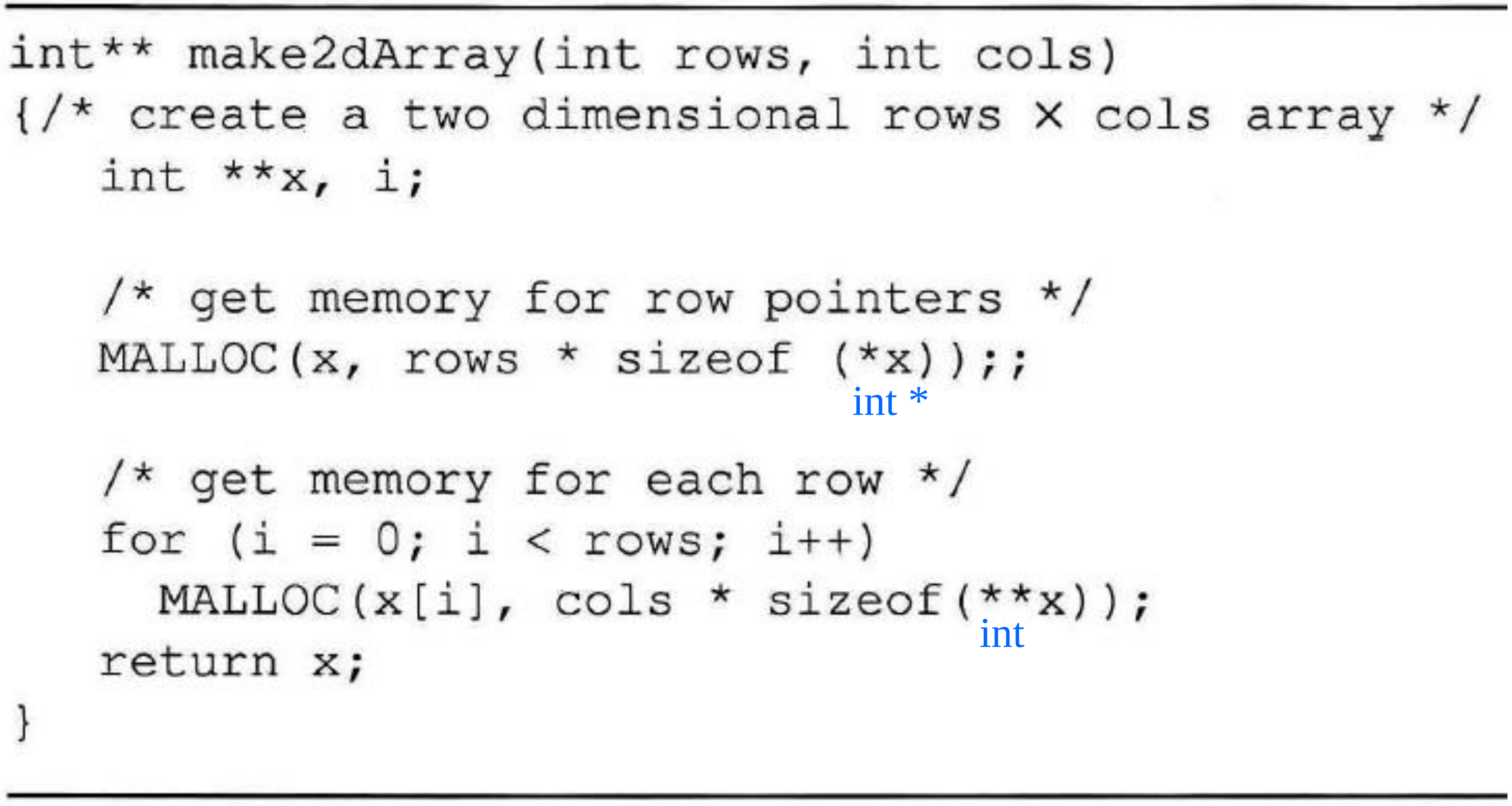
Figure 2.2: Array-of-arrays representation

x[i] : a pointer to zeroth element of row i of the array

x[i][j] : an element accessed by the address, x[i]+j*sizeof(int)

```
int **myArray;
myArray = make2dArray(5,10);
myArray[2][4] = 6;
```





```
int** make2dArray(int rows, int cols)
{/* create a two dimensional rows × cols array */
   int **x, i;

   /* get memory for row pointers */
   MALLOC(x, rows * sizeof (*x));;
                                int *

   /* get memory for each row */
   for (i = 0; i < rows; i++)
     MALLOC(x[i], cols * sizeof(**x));
                                  int
   return x;
}
```

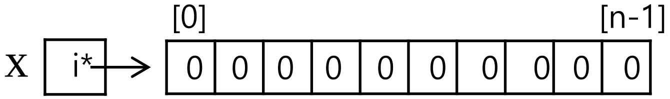
Program 2.3: Dynamically create a two-dimensional array

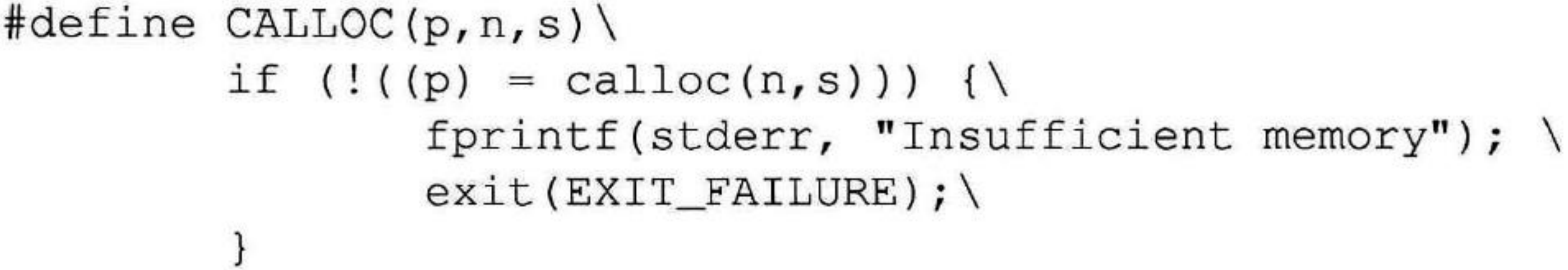
- calloc

  int *x, n;

  x = (int *) calloc(n, sizeof(int));

  /* allocated bits are set to 0*/





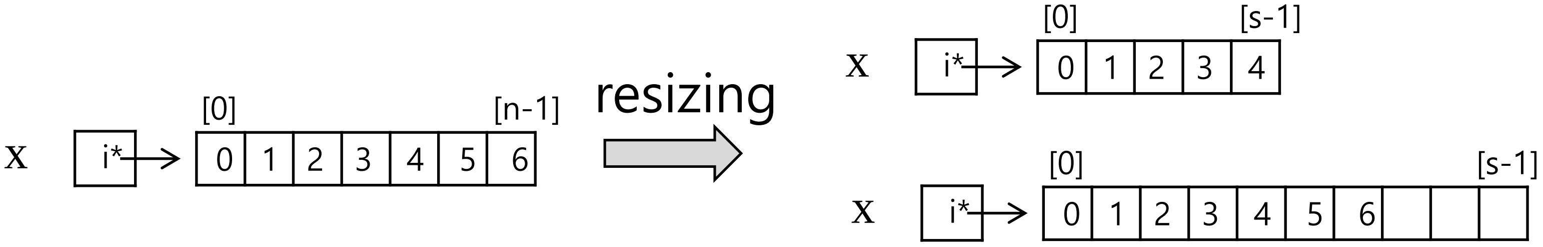
```
#define CALLOC(p,n,s)\
        if (!((p) = calloc(n,s))) {\
              fprintf(stderr, "Insufficient memory"); \
              exit(EXIT_FAILURE);\
        }
```

CALLOC(x, n, sizeof(int));

- realloc

```
int *old, *x, s;
...
old = x;
if ( (x = (int *)realloc(x, s*sizeof(int))) == NULL ){
        free(old);
        exit(EXIT_FAILURE);
}
…
free(x);
```

/* changes the size of memory block pointed by x to s*sizeof(int) */



- realloc(cont’)

```
#define REALLOC(o, p, s) \
       if (o = p && !((p) = realloc(o, s))) {\
               free(o);\
               exit(EXIT_FAILURE); \
        }
    ...
REALLOC(old, x, s*sizeof(int));//x = realloc(x, s*sizeof(int))
```

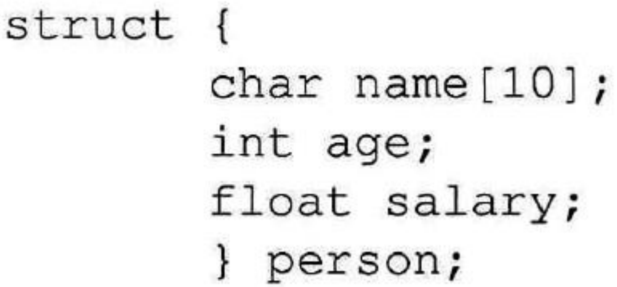
or

```
#define REALLOC(o, p, s) \
       if (o = p && !((p) = realloc(o, s))) {\
               p = o;\
               fprintf(stderr, "Insufficient memory"); \
        }
    ...
REALLOC(old, x, s*sizeof(int));//x = realloc(x, s*sizeof(int))
```

2.3 Structures

2.3.1 Structures

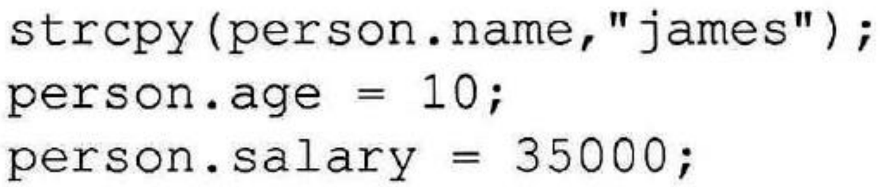
- Called a *record*
- Collection of data items
  - Each item is identified as to its type and name



```
struct {
        char name[10];
        int age;
        float salary;
        } person;
```

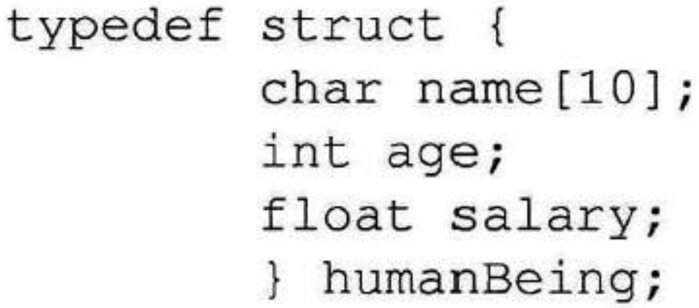
person is a variable.

  - Structure member operator : dot(.)



```
strcpy(person.name,"james");
person.age = 10;
person.salary = 35000;
```

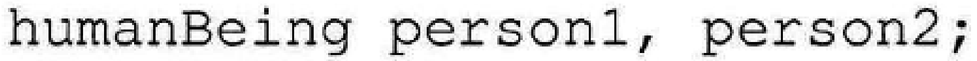
- Using the **typedef** statement



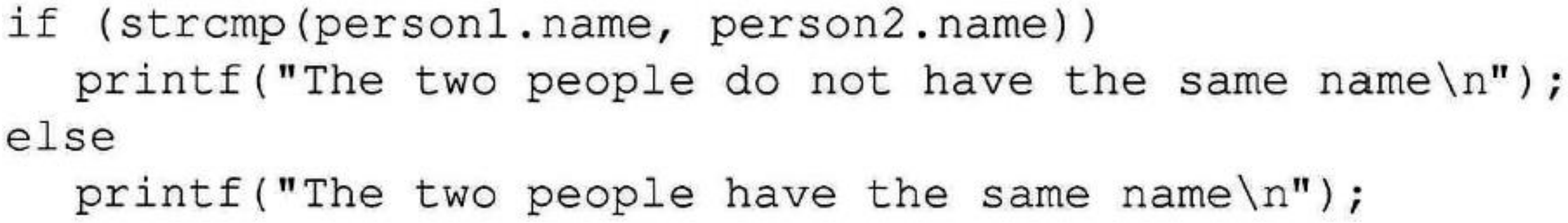
```
typedef struct {
        char name[10];
        int age;
        float salary;
        } humanBeing;
```

humanBeing is a data type.

- Declaration of variables

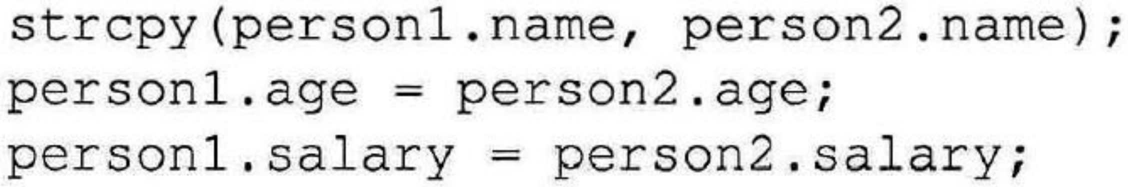


```
humanBeing person1, person2;
```



```
if (strcmp(person1.name, person2.name))
  printf("The two people do not have the same name\n");
else
  printf("The two people have the same name\n");
```

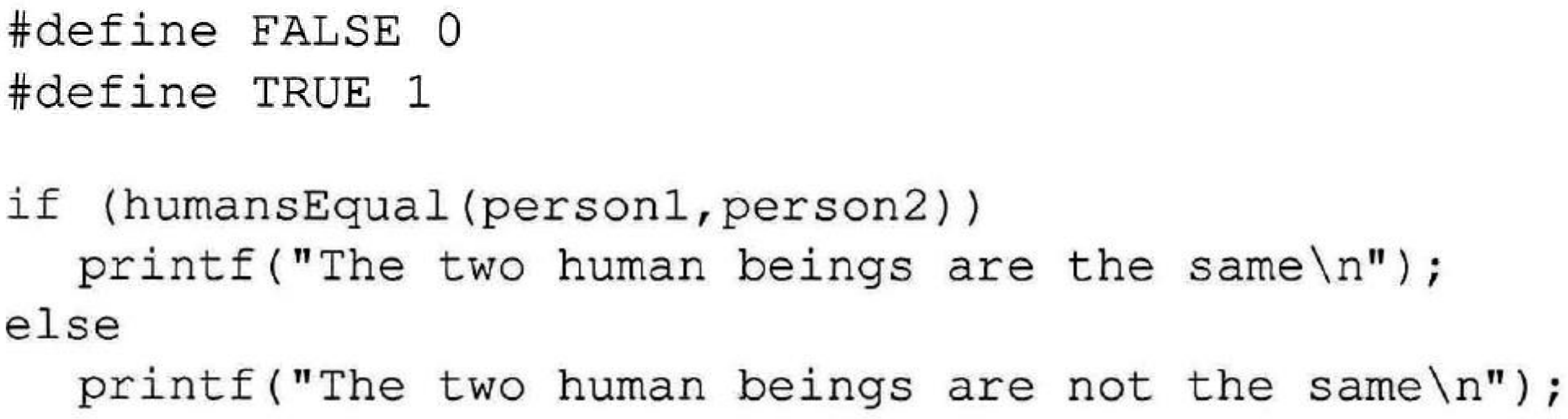
- Structure assignment : person1=person2;
  - in ANSI C, OK!
    - However, don’t use the assignment operation when the structure has a pointer to a memory space. Why?
  - in older versions of C, NOT OK!



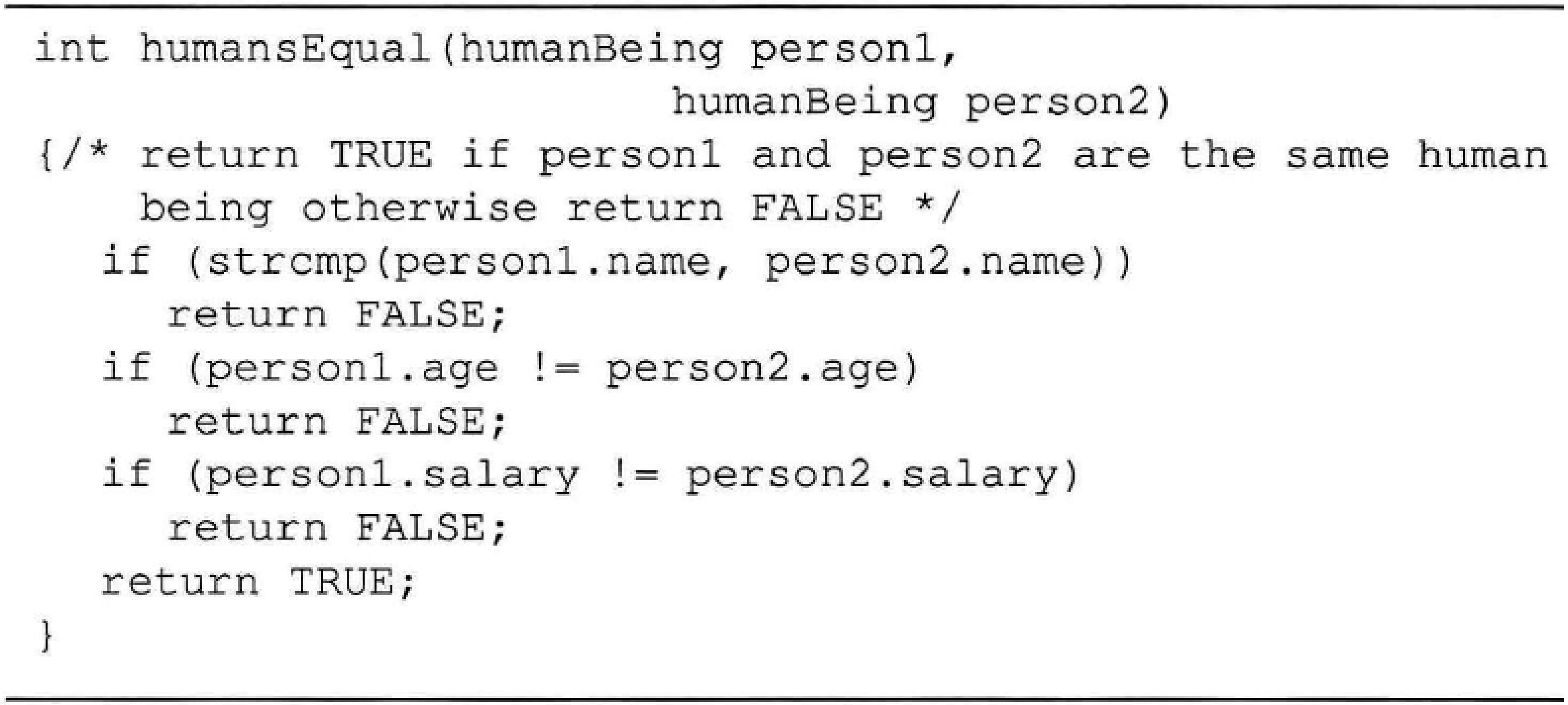
```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

- Check of equality or inequality : if(person1==person2)
  - cannot be checked directly

- Check of equality or inequality(cont')



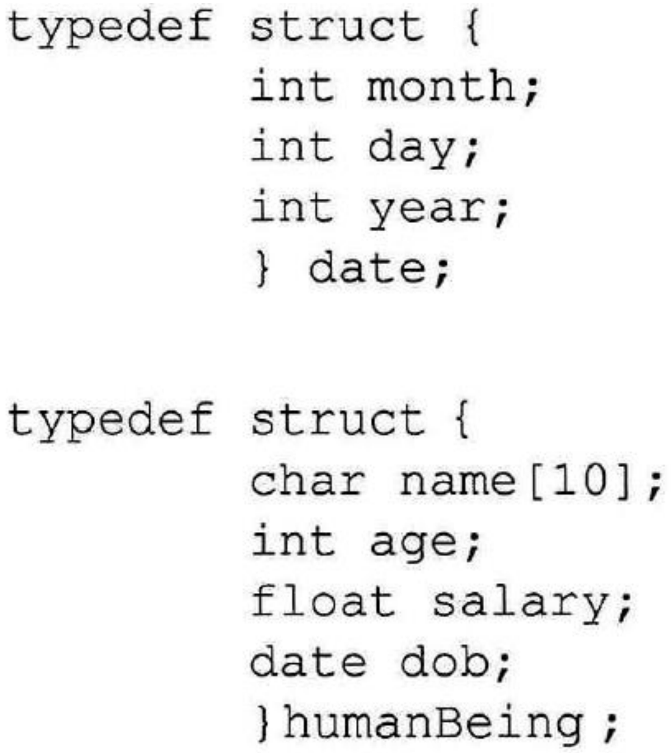
```
#define FALSE 0
#define TRUE 1

if (humansEqual(person1,person2))
  printf("The two human beings are the same\n");
else
  printf("The two human beings are not the same\n");
```



```
int humansEqual(humanBeing person1,
                        humanBeing person2)
{/* return TRUE if person1 and person2 are the same human
    being otherwise return FALSE */
   if (strcmp(person1.name, person2.name))
     return FALSE;
   if (person1.age != person2.age)
     return FALSE;
   if (person1.salary != person2.salary)
     return FALSE;
   return TRUE;
}
```

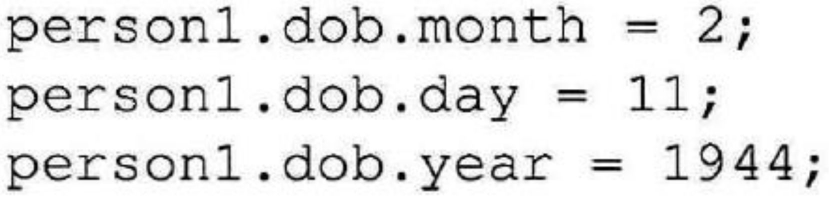
Program 2.4: Function to check equality of structures

- A structure within a structure



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


typedef struct {
        char name[10];
        int age;
        float salary;
        date dob;
        }humanBeing;
```

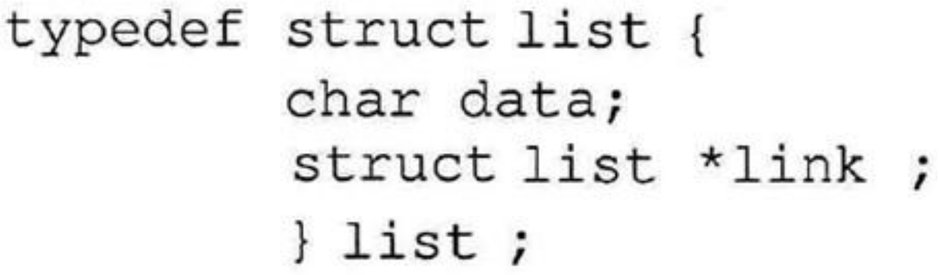
```
humanBeing person1;
```



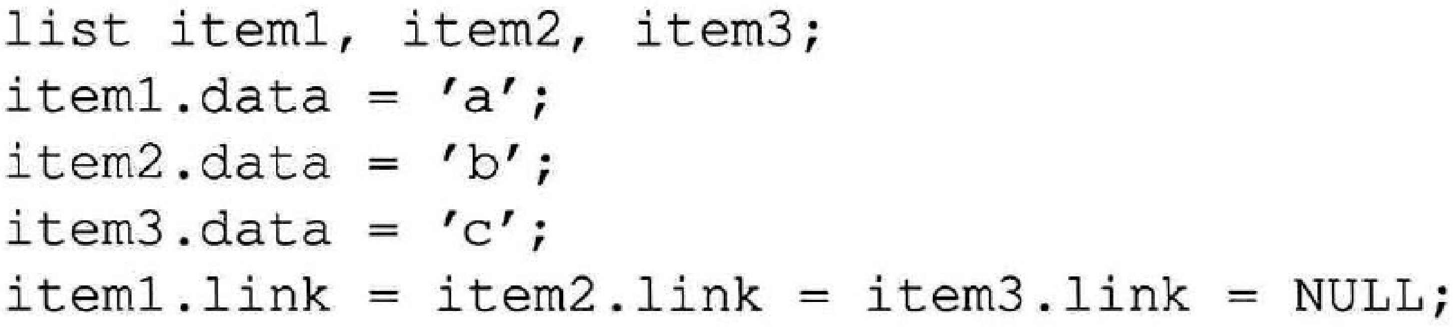
```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2.3.4 Self-Referential Structures

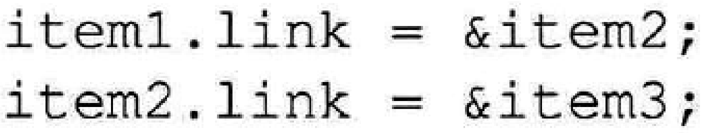
- A structure in which one or more of its components is a pointer to itself.



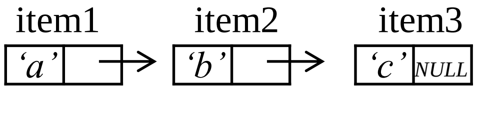
```
typedef struct list {
        char data;
        struct list *link ;
        } list ;
```



```
list item1, item2, item3;
item1.data = ’a’;
item2.data = ’b’;
item3.data = ’c’;
item1.link = item2.link = item3.link = NULL;
```



```
item1.link = &item2;
item2.link = &item3;
```



item1
item2
item3
‘a’
‘b’
‘c’
NULL