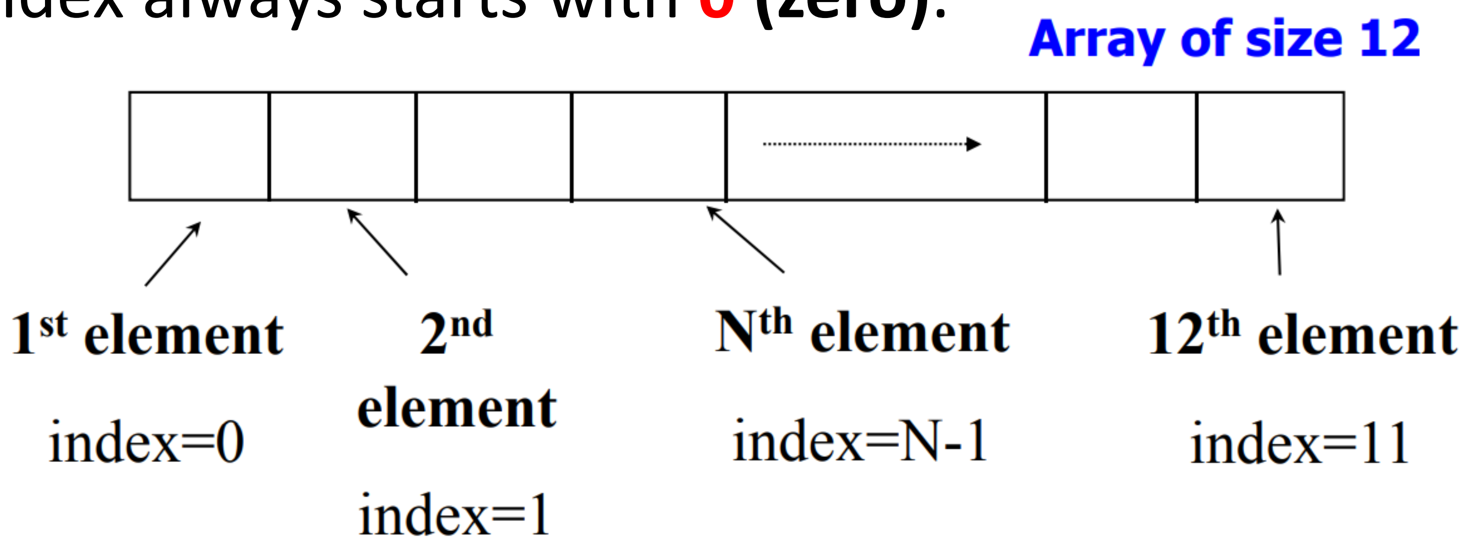
CS100
Introduction to Programming

Lecture 7. Arrays

What Is an Array

- An **array** is a list of values with the same data type. Each value is stored at a specific, numbered position in the array.
- An array uses an integer called **index** to reference an element in the array.
- The size of an array is **fixed once it is created**.
- Index always starts with **0 (zero)**.

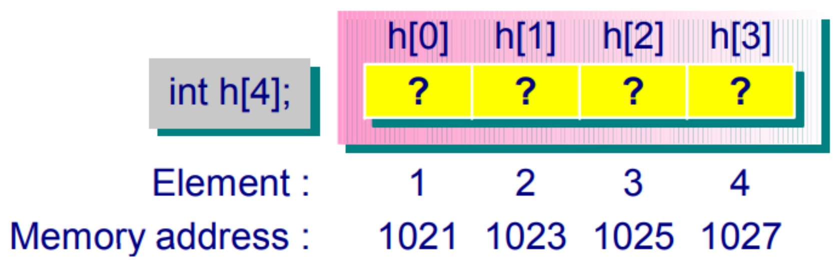


Array Declaration

- Declaration of arrays without initialization:

```
float sales[365];  // array of 365 floats
char name[12];      // array of 12 characters
int states[50];     // array of 50 integers
int *pointers[5];  // array of 5 pointers to integers
```

- When an array is declared, some consecutive memory locations are allocated by the compiler for the whole array (assume 2 bytes for an integer).



- The size of an array must be an integer constant or constant expression:

  e.g. char name[i];   // i is a variable ==> illegal

  int states[i*6];   // i is a variable ==> illegal

Initialization of Arrays

- Initialize array variables at declaration:

```
#define MTHS 12           /* define a constant */
int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	31	30	31	30	31

- Partial array initialization, e.g. to initialize the first 7 elements:

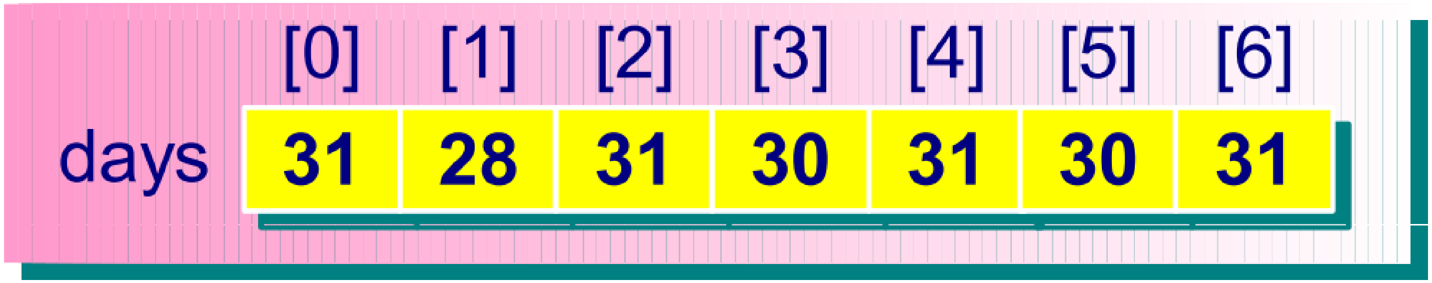
```
#define MTHS 12
int days[MTHS] = {31, 28, 31, 30, 31, 30, 31};
/* remaining elements are initialized to 0 */
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	0	0	0	0	0

Initialization of Arrays

- Omitting the size in array initialization:

```
int days[] = {31, 28, 31, 30, 31, 30, 31};
/* an array of 7 elements */
```



Operations on Arrays

- Accessing array elements:

```
sales[0] = 143.50;
if (sales[23] == 50.0) …
```

- **Subscripting**: The element indices range from **0** to **n – 1** where **n** is the declared size of the array:

```
char name[12];
name[12] = ‘c’;     // error: index out of range
```

- Working on array values:

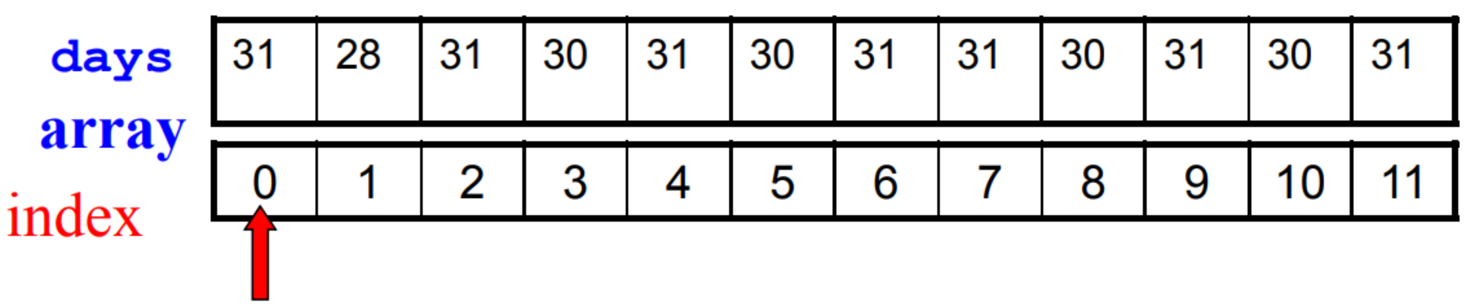
```
days[1] = 20;                     // valid
days[2] = days[2] + 4;            // valid
days[3] = days[2] + days[3];      // valid
days[MTHS] = {2, 3, 4, 5, 6};     // invalid
```

Traversing an Array

- One of the most common actions in dealing with arrays is to examine every array element in order to perform an operation.

- This action is also known as **traversing** an array.

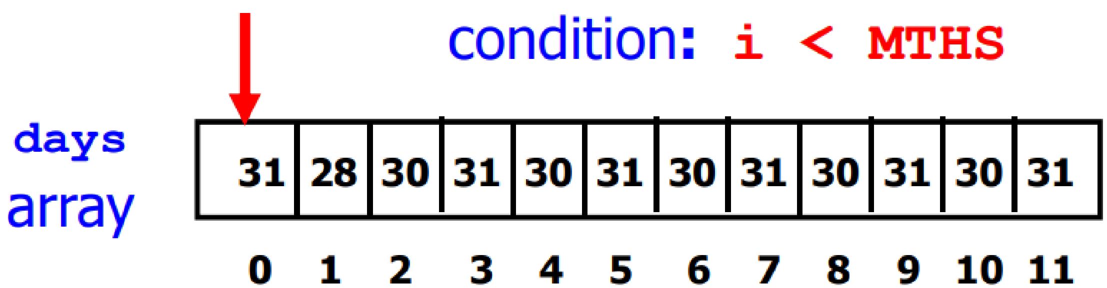
- Example:
  - Traverse the days[] array to display every element's content:



Traversing an Array – print values

```
#include <stdio.h>
#define MTHS 12   // define a constant
int main(void)
{
   int i;
   int days[MTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   // print the number of days in each month
   for (i = 0; i < MTHS; i++)
      printf(“Month %d has %d days.\n”, i+1, days[i]);
   return 0;
}
```

Output:
Month 1 has 31 days.
Month 2 has 28 days.
…
Month 12 has 31 days.



Traversing an Array – search for a value

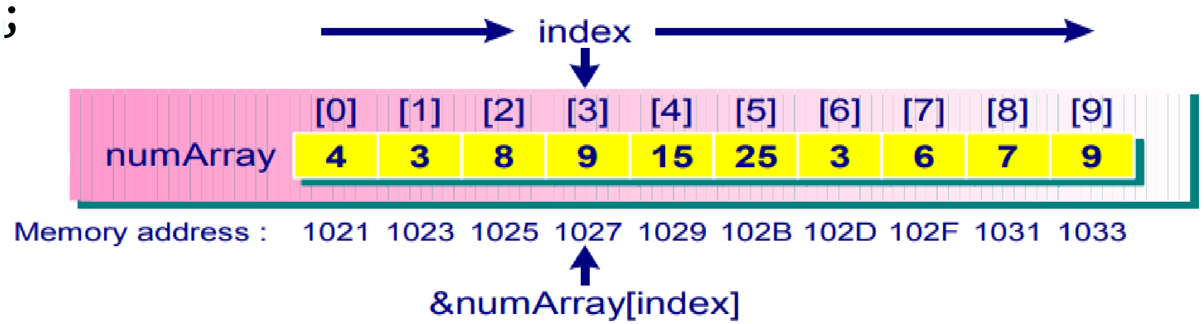
```
#include <stdio.h>
#define SIZE 5   // define a constant
int main(void)
{
   char myChar[SIZE] = {‘b’, ‘a’, ‘c’, ‘k’, ‘s’};
   // Reading in user’s input to search
   printf(“Enter a char to search: ”);
   char searchChar;
   scanf(“%c”, &searchChar);
   /* Traverse myChar array and output the index of
      searchChar if found */
   int i;
   for (i = 0; i < SIZE; i++) {
      if (myChar[i] == searchChar) {
         printf(“Found %c at index %d”, myChar[i], i);
         break;  // break out of the loop
      }
   }
   return 0;
}
```

Output:
Enter a char to search: ***<u>a</u>***
Found a at index 1

Traversing an Array – find maximum value

```
/* This example shows how to find the largest value in
an array of numbers. */
#include <stdio.h>
int main(void)
{
   int i, max, numArray[10];
   max = -1;
   printf(“Enter 10 numbers: \n”);
   for (i = 0; i < 10; i++)
      scanf(“%d”, &numArray[index]);
   for (i = 0; i < 10; i++) {
      if (numArray[i] > max)
         max = numArray[i];
   }
   printf(“The max value is %d.\n”, max);
   return 0;
}
```

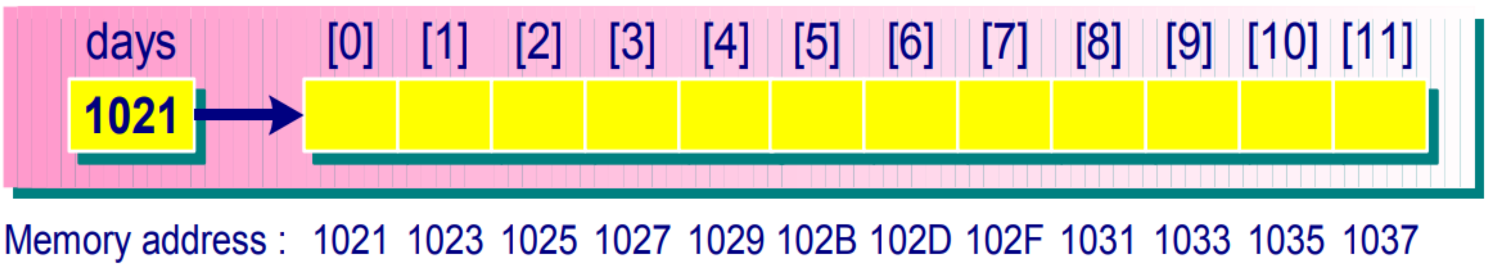
Output:
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
The max value is 25.



Pointers and Arrays

- The array name is really a **pointer constant**:

  e.g. int days[12];



- If an integer is represented by 2 bytes and the array **days** begins at memory location **1021**, the above figure shows the layout of the array.

- Address of an array element: e.g. int h[5];

  &h[0] is the address of the 1st element

  &h[i] is the address of the (i+1)-th element

Pointers and Arrays

- Thus, **days**, the array name by itself, is really the **address (or pointer)** of the 1st element of the array, e.g. when using the array of int days[12], the following expressions are all true:

```
days == &days[0]
*days == days[0]
days + 1 == &days[1]
*(days + 1) == days[1]
```

- You cannot change the value of the array name, because it is a **pointer constant**, **not a pointer variable**:

```
days += 5;  // invalid
days++;     // invalid
```

Pointers and Arrays

A ***pointer variable*** can take on different addresses, but an array cannot:

```
/* pointer arithmetic */
#define MTHS 12
int main(void)
{
   int days[MTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int *day_ptr;
   day_ptr = days;      /* points to the first element */
   day_ptr = &days[3];  /* points to the fourth element */
   day_ptr += 3;        /* points to the seventh element */
   day_ptr--;           /* points to the sixth element */
   return 0;
}
```

Pointers and Arrays

Statement	day_ptr	days	[0] 1021	[1] 1023	[2] 1025	[3] 1027	[4] 1029	[5] 102B	[6] 102D	[7] 102F	......	[11] 1037
int days[MTH] = {......};	?	1021	31	28	31	30	31	30	31	31	......	31
day_ptr = days;	1021	1021	31	28	31	30	31	30	31	31	......	31
day_ptr = &days[3];	1027	1021	31	28	31	30	31	30	31	31	......	31
day_ptr += 3;	102D	1021	31	28	31	30	31	30	31	31	......	31
day_ptr--;	102B	1021	31	28	31	30	31	30	31	31	......	31

Pointer – Finding Maximum Number

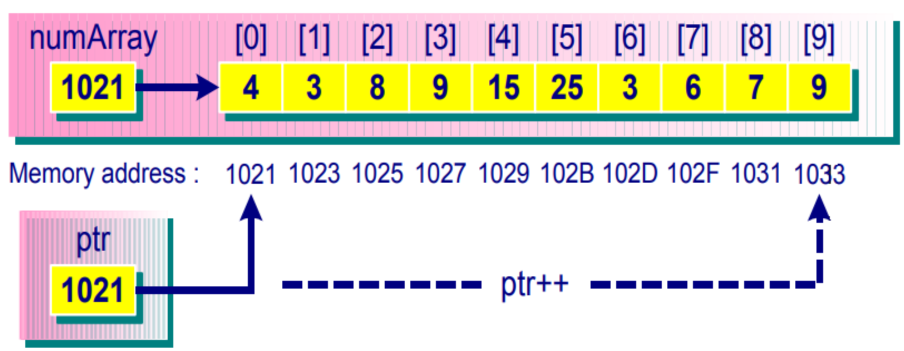
```
#include <stdio.h>
int main(void)
{
   int i, max, numArray[10];
   printf("Enter 10 numbers: \n");
   for (i = 0; i < 10; i++)
      scanf("%d", numArray + i);

   max = *numArray;
   for (i = 1; i < 10; i++) {
      if (*(numArray + i) > max)
         max = *(numArray + i);
   }
   printf("The max value is %d.\n", max);
   return 0;
}
```

Output:
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
The max value is 25.

Pointer – Finding Maximum Number

```
#include <stdio.h>
int main(void) {
   int i, max, numArray[10];
   int *ptr;
   ptr = numArray;
   printf(“Enter 10 numbers: \n”);
   for (i = 0; i < 10; i++)
      scanf(“%d”, ptr++);
   ptr = numArray;
   max = *ptr;
   for (i = 1; i < 10; i++) {  // find the max
      if (*ptr > max)
         max = *ptr;
      ptr++;
   }
   printf(“max is %d.\n”, max);
   return 0;
}
```

Output:
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
max is 25.



Arrays as Function Arguments

- Any dimensional array can be passed as a function argument, e.g.

```
fn(table);      /* call a function */
```

  where **fn()** is a function and **table** is a 1-D array.

- An array is passed **by reference** to a function. This means that the **address** of the first element of the array is passed to the function.

Arrays as Function Arguments

```
void fn(int table[], int n)
{
    ......
}
```

```
void fn(int table[TABLESIZE])
{
    ......
}
```

```
void fn(int *table, int n)
{
    ......
}
```

- The prototype of the function becomes

```
void fn(int table[], int n);          or
void fn(int table[TABLESIZE]);        or
void fn(int *table, int n);
```

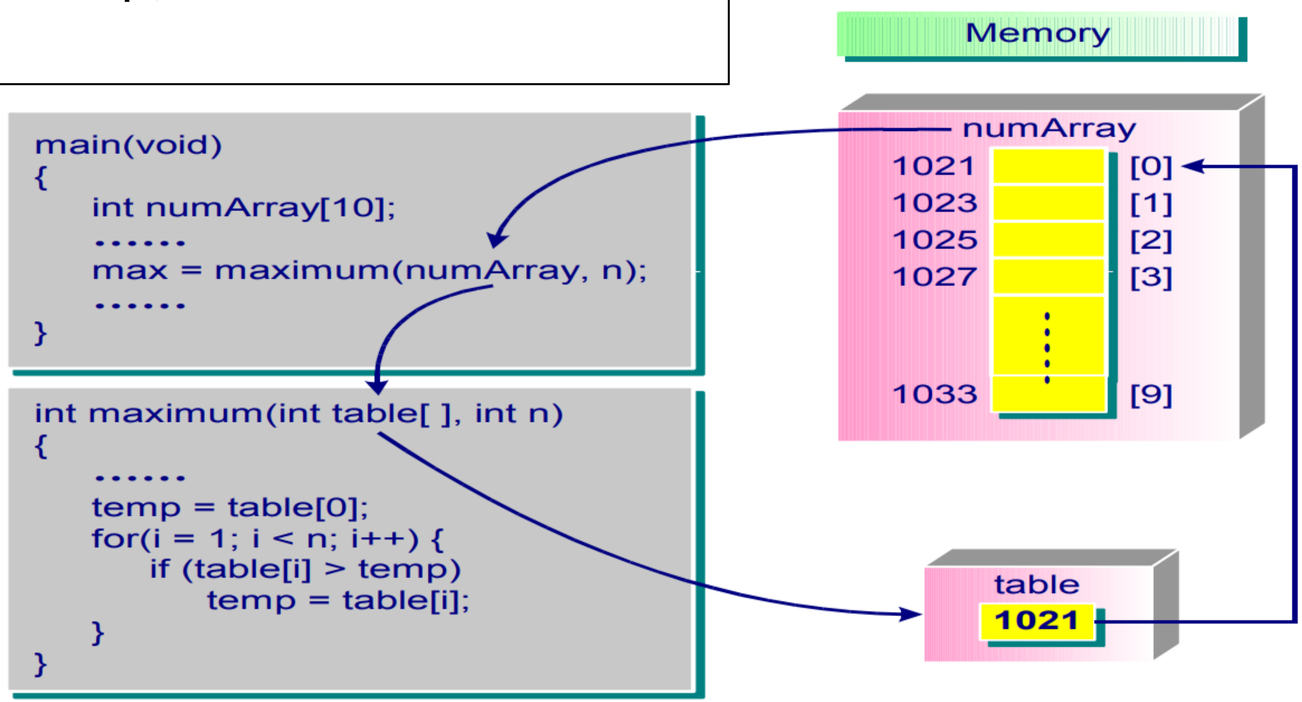
Passing an Array as a Function Argument

```
#include <stdio.h>

int maximum(int table[], int n);

int main(void)
{
   int max, i, n;
   int numArray[10];
   printf("Enter the number of values:");
   scanf("%d", &n);
   printf("Enter %d values: ", n);
   for (i = 0; i < n; i++)
      scanf("%d", &numArray[i]);
   max = maximum(numArray, n);
   printf("The max value is %d.\n", max);
   return 0;
}
```

Output:
Enter the number of values: ***5***
Enter 5 values: ***1 2 3 4 5***
The max value is 5.

```
int maximum(int table[], int n)
{
   int i, temp;
   temp = table[0];
   for (i = 1; i < n; i++)
      if (table[i] > temp)
         temp = table[i];
   return temp;
}
```

Passing an Array as a Function Argument



Multidimensional Arrays

- Declared as consecutive pairs of brackets.
- E.g. A 2-dimensional array, or a 3-element array of 5-element arrays:

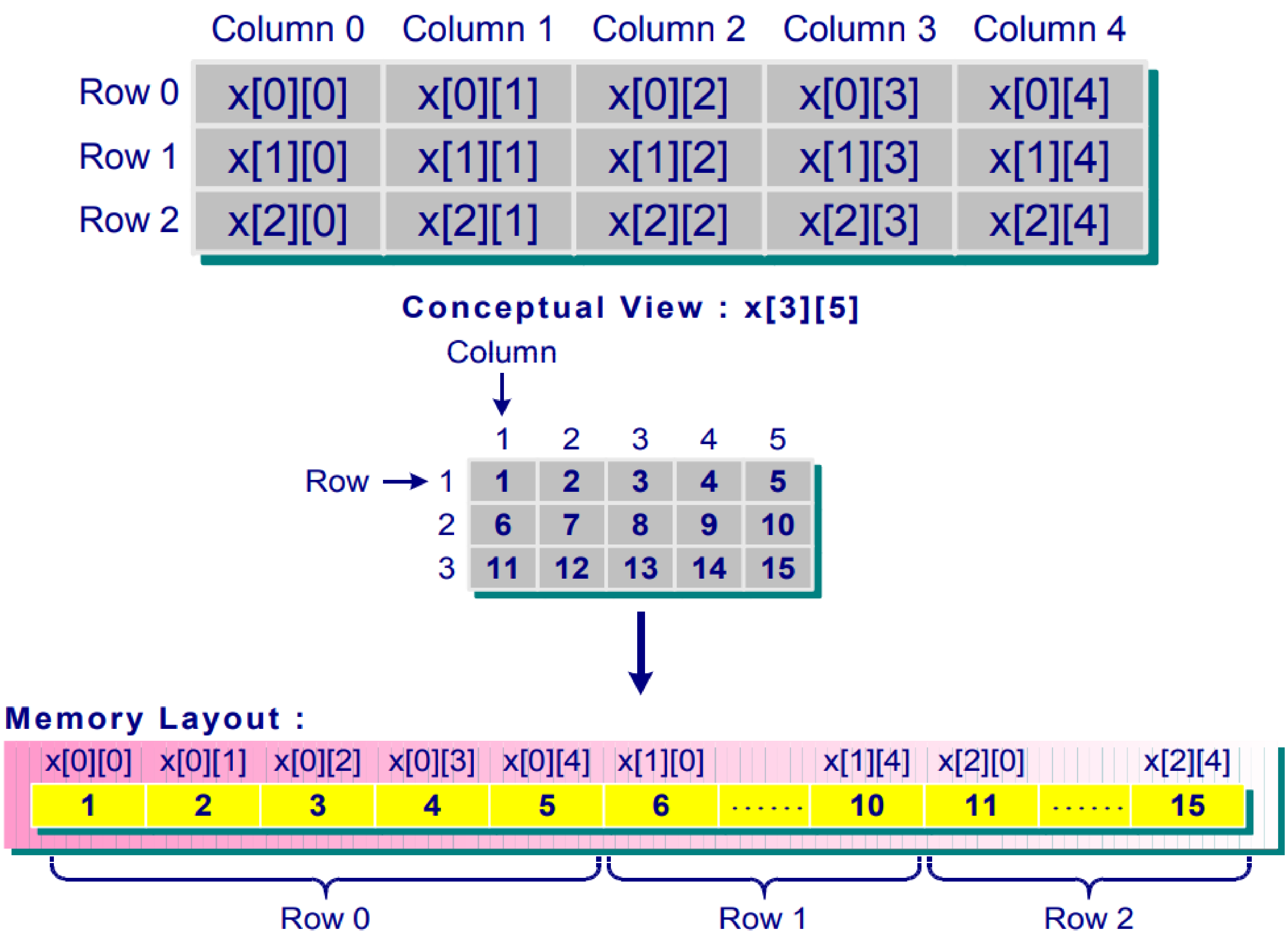
```
int x[3][5];
```

- E.g. A 3-dimensional array, or a 3-element array of 4-element arrays of 5-element arrays:

```
char x[3][4][5];
```

- ANSI standard requires a minimum of 6 dimensions to be supported.

Multidimensional Arrays



Initializing Multidimensional Arrays

- Initializing multidimensional arrays: enclose each row in curly braces.

```
int x[2][2] = {{1, 2},          /* 1st row */
               {6, 7}};         /* 2nd row */
```

or

```
int x[2][2] = {1, 2, 6, 7};
```

- Partial initialization (other cells are set to 0):

```
int exam[3][3] = {{1, 2}, {4}, {5, 7}};
```

```
int exam[3][3] = { 1, 2, 4, 5, 7 };
```

or

```
int exam[3][3] = { {1, 2, 4}, {5, 7} };
```

Initializing Multi-dimensional Arrays

- You can omit the outermost dimension because compiler can figure that out, e.g.

```
int arr[][3][2] = {
        { {1, 1}, {0, 0}, {1, 1} },
        { {0, 0}, {1, 2}, {0, 1} }
};
```

gives a [2][3][2] dimensioned array.

- The following is not correct. Why?

```
int wrong_arr[][] = {1, 2, 3, 4};
```

Operations on Multidimensional Arrays

```
#include <stdio.h>
int main(void)
{
   int array[3][3] = {
        {5, 10, 15},
        {10, 20, 30},
        {20, 40, 60}
     };
   int row, column, sum;
   /* sum of rows */
   for (row = 0; row < 3; row++) {
      sum = 0;
      for (column = 0; column < 3; column++)
         sum += array[row][column];
      printf("The sum of elements in row %d is %d\n",
              row + 1, sum);
   }
```

Code continues in next slide …

Operations on Multidimensional Arrays

```
   /* sum of columns */
   for (column = 0; column < 3; column++) {
      sum = 0;
      for (row = 0; row < 3; row++)
         sum += array[row][column];
      printf("The sum of elements in column %d is %d\n",
                     column + 1, sum);
   }
   return 0;
}
```

Output:
The sum of elements in row 1 is 30
The sum of elements in row 2 is 60
The sum of elements in row 3 is 120
The sum of elements in column 1 is 35
The sum of elements in column 2 is 70
The sum of elements in column 3 is 105

Multidimensional Arrays and Pointers

- Multidimensional arrays are also stored sequentially in memory, e.g.

```
int ar[4][2];  /* ar is an array of 4 elements;
                  each element is an array of 2 ints */
```

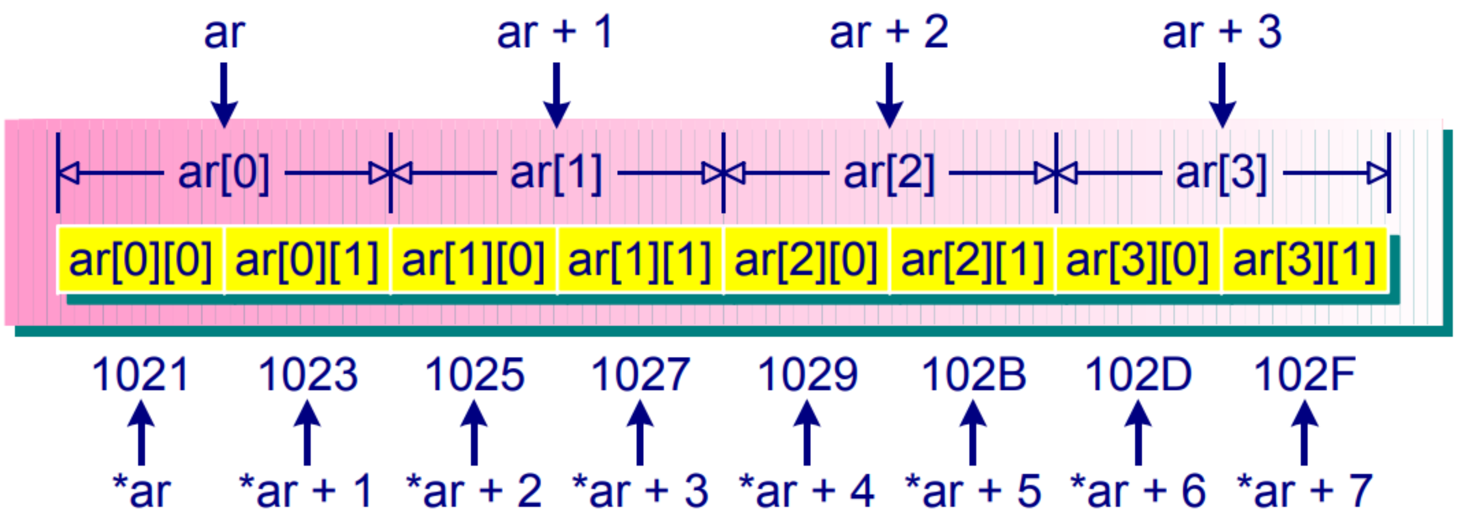
- **ar** is the address of the 1st element of the array. In this case, the 1st element is an array of 2 integers. Thus, ar is the address of a two-int-sized object.

```
ar == &ar[0]          *ar == ar[0]
ar + 1 == &ar[1]      *(ar + 1) == ar[1]
ar + 2 == &ar[2]      *(ar + 2) == ar[2]
ar + 3 == &ar[3]      *(ar + 3) == ar[3]
```

Multidimensional Arrays and Pointers

- **ar[0]** is an array of 2 integers, so ar[0] is the address of an int-sized object.

```
ar[0] == &ar[0][0]          *ar[0] == ar[0][0]
ar[1] == &ar[1][0]          *ar[1] == ar[1][0]
ar[2] == &ar[2][0]          *ar[2] == ar[2][0]
ar[3] == &ar[3][0]          *ar[3] == ar[3][0]
```



Multidimensional Arrays and Pointers

- Adding 1 to a pointer or address yields a value larger **by the size of the referred-to object**.

  E.g. **ar** has the same address value as **ar[0]**, but

  ar+1 (1025) is different from **ar[0]+1** (1023)

- Dereferencing a pointer or an address (apply *** operator**) yields the value represented by the referred-to object. For example:

  *(ar[0]) == the value stored in ar[0][0].

  *ar == the value of its first element, ar[0].

  **ar == the value of ar[0][0] (*double indirection*)

- In general,

  a[m][n] == *(*(a + m) + n)

Multidimensional Arrays as Function Arguments

- The definition of a function with a 2-D array as the argument is:

```
void fn(int ar2[2][4])        or        void fn(int ar2[][4])
{                                       {
      …                                       …
}                                       }
```

/* the first dimension can be excluded */

- In the above definition, the **first dimension can be excluded** because the C compiler needs the information of all but the first dimension.

Multidimensional Arrays as Function Arguments

- For example, the assignment operation

```
ar2[1][3] = 100;
```

requests the compiler to compute the address of **ar2[1][3]** and then place 100 to that address. In order to compute the address, the dimension information must be given to the compiler.

- Let us redefine **ar2** as

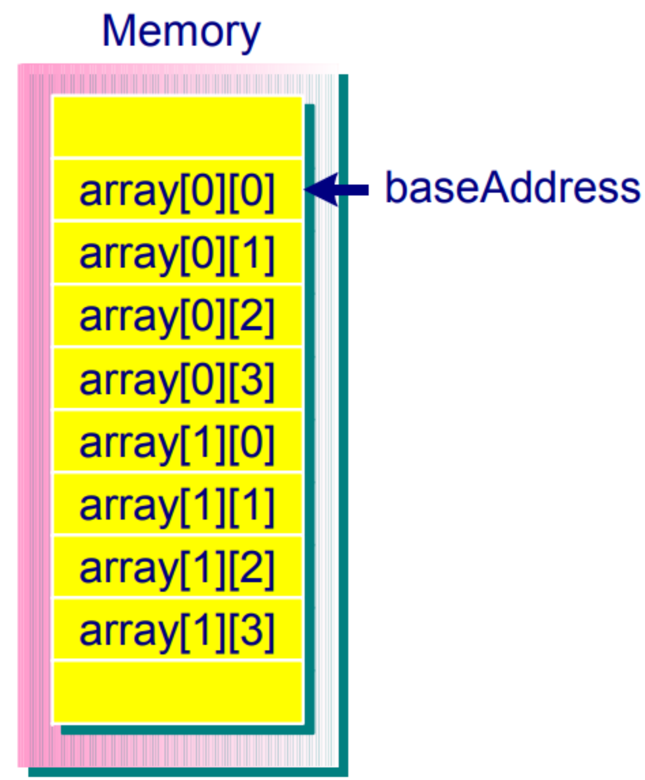
```
int ar2[D1][D2];
```

The address of **ar2[1][3]** is computed as

baseAddress + row * D2 + column
==> baseAddress + 1 * 4 + 3
==> baseAddress + 7

Multidimensional Arrays as Function Arguments

- The **baseAddress** is the address pointing to the beginning of **ar2**. Because **D1** is not needed in computing the address, one can omit the first dimension value in defining a function which takes arrays as its formal arguments.



- The prototype of the function becomes

```
void fn(int ar2[2][4]); or
void fn(int ar2[][4]);
```

Passing 2-D Array as Function Arguments

```
#include <stdio.h>
int sum_rows(int ar[][3]);
int sum_columns(int ar[][3]);
int main(void)
{
   int array[3][3] = {
         {5, 10, 15},
         {10, 20, 30},
         {20, 40, 60}
      };
   int total_row, total_column;
   total_row = sum_rows(array);
   total_column = sum_columns(array);
   printf("The sum of all elements in rows is %d\n",
      total_row);
   printf("The sum of all elements in columns is %d\n",
         total_column);
   return 0;
}
```

Output:

The sum of all elements in rows is 210

The sum of all elements in columns is 210

Passing 2-D Array as Function Arguments

```
int sum_rows(int ar[][3]) {
   int row, column;
   int sum = 0;
   for (row = 0; row < 3; row++) {
      for (column = 0; column < 3; column++)
         sum += ar[row][column];
   }
   return sum;
}

int sum_columns(int ar[][3]) {
   int row, column;
   int sum = 0;
   for (column = 0; column < 3; column++) {
      for (row = 0; row < 3; row++)
         sum += ar[row][column];
   }
   return sum;
}
```

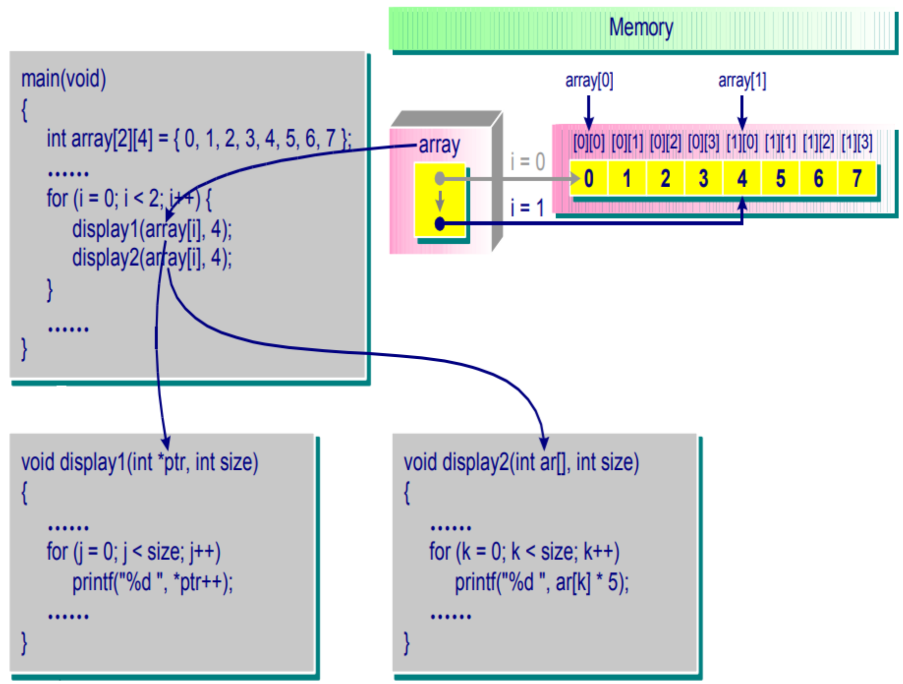
Processing 2-D Arrays as 1-D Arrays

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[], int size);
void display3(int ar[][4], int size);

int main(void)
{
   int array[2][4] = {0,1,2,3,4,5,6,7};
   int i;

   for (i = 0; i < 2; i++) {
      display1(array[i], 4);
      display2(array[i], 4);
   }
   display3(array, 2);
   display1(array, 8);
   display2(array, 8);
   return 0;
}
```

Output:
Display1 result: 0 1 2 3
Display2 result: 0 5 10 15
Display1 result: 4 5 6 7
Display2 result: 20 25 30 35
Display3 result: 0 10 20 30 40 50 60 70
Display1 result: 0 1 2 3 4 5 6 7
Display2 result: 0 5 10 15 20 25 30 35

Processing 2-D Arrays as 1-D Arrays

```
void display1(int *ptr, int size)
{
    int j;
    printf(“Display1 result: ”);
    for (j=0; j < size; j++)
        printf(“%d ”, *ptr++);
    putchar(‘\n’);
}
void display2(int ar[], int size)
{
    int k;
    printf(“Display2 result: ”);
    for (k=0; k < size; k++)
        printf(“%d ”, ar[k]*5);
    putchar(‘\n’);
}
void display3(int ar[][4], int size)
{
    int i,j;
    printf(“Display3 result: ”);
    for (i=0; i < size; i++)
        for (j=0; j < 4; j++)
            printf(“%d ”, ar[i][j]*10);
    putchar(‘\n’);
}
```



Passing Array as Pointer Argument

- Passing 1D array as pointer

```
void display(int *ptr, int size)
{
    int i;
    printf("Display result: ");
    for (i=0; i < size; i++)
        printf("%d ", *ptr++);
        //printf("%d ", ptr[i]);
    putchar('\n');
}
```

```
#include <stdio.h>
void display(int *ptr, int size);

int main(void)
{
    int array[8] = {0,1,2,3,4,5,6,7};

    display(array, 8);
   //display(&array[0], 8);
    return 0;
}
```

Passing Array as Pointer Argument

- Passing 2D static array as pointer

```
void display(int *ptr,
           int size1, int size2)
{
    int i, j;
    printf("Display result: ");
    for (i=0; i < size1; i++)
         for (j=0; j < size2; j++)
            printf("%d ", ptr[size2*i+j]);
    putchar('\n');
}
```

```
#include <stdio.h>
void display(int *ptr, int size1, int size2);
void display(int **ptr, int size1, int size2);

int main(void)
{
    int array[2][4] = {{0,1,2,3},{4,5,6,7}};
    display((int *)array, 2, 4);
    // display((int **)array, 2, 4);
    return 0;
}
```

```
void display(int **ptr,
           int size1, int size2)
{
    int i, j;
    printf("Display result: ");
    for (i=0; i < size1; i++)
         for (j=0; j < size2; j++)
            printf("%d ", *((int *)ptr+size2*i+j));
    putchar('\n');
}
```

The sizeof Operator and Array

```
#include <stdio.h>

int main(void)
{
    int ar2[2][4];
    printf("Array size is %d",
           sizeof(ar2) / sizeof(ar2[0][0]));
    return 0;
}
```

Output:
Array size is 8

The sizeof Operator and Array

- sizeof(operand) is an operator which gives the size (how many bytes) of its operand. Its syntax is

  sizeof (**operand**)   or

  sizeof operand

- The **operand** can be:

  int, float, …., complexDataTypeName,

  variableName, arrayName

```
#include <stdio.h>
#define SIZE 5
int item[SIZE] = {1, 2, 3, 4, 5};
void main(void)
{
    total = sum(item, SIZE);
    printf("Size of item = %d\n", sizeof(item));
}
```

The sizeof Operator and Array

```
int sum(int a[], int n)
{
    int i;
    int s = 0;
    printf("Size of a = %d\n", sizeof(a));
    for (i=0; i < n; i++)
        s += a[i];
    return s;
}
```

Output:

Size of a = 4

Size of item = 20

Applying ***sizeof*** to an **array name** yields the array size

BUT

Applying ***sizeof*** to a **pointer variable** yields the size of the pointer

Size of a Pointer

- **What is the size of a pointer?**
  - Always fixed size for pointer of any type
  - OS System dependent
    - Old system: 32bit (4 bytes); can only access 4G memory
    - System nowadays: 64bit (8 bytes); can access very large

```
#include <stdio.h>

void main(void)
{
    float a=10.0; float *p_a=&a;
    int b=5; int *p_b=&b;
    printf("Size of a = %d\n", sizeof(p_a));
    printf("Size of b = %d\n", sizeof(p_b));
}
```

Dynamically Allocated Array

- **Static v.s. dynamic array**
  - Whether the size of the array can be dynamically changed during execution

```
float A[10];


#define SIZE 10
float A[SIZE]


const int SIZE=10;
float A[SIZE];
```

Examples of declaring static array

```
int SIZE=10;
float A[SIZE];
```

How about the dynamic array size?

Dynamically Allocated 1D Array

- **How to achieve dynamic array?**
  - Using pointers
  - With dynamic memory allocation function

```
int main()
{
    int size=10;
    float *pArray=NULL;
    pArray=(float *)malloc(sizeof(float)*size);
    memset(pArray,0,sizeof(float)*size);
    /*access the array*/
    free(pArray);

    return 0;
}
```

Dynamically Allocated 1D Array

- **Access array elements**
  - The allocated array pointer can be taken as the normal array

```
int main()
{
    …
    /*access the array*/
    for(int i=0;i<size;i++)
        pArray[i]=(float)(i);
         //*(pArray+i)=(float)(i);
    …

    return 0;
}
```

Dynamically Allocated 2D Array

- **Dynamically allocating 2D array**
  - Using double pointer indexing

```
int main()
{
        int size1=10, size2=8;
        float **pArray=NULL;
        pArray=(float **)malloc(sizeof(float *)*size1);
        for(int i=0;i<size1;i++)
             pArray[i]=(float *)malloc(sizeof(float)*size2);

        /*access the array*/

        for(int i=0;i<size1;i++)
                free(pArray[i]);
        free(pArray);

        return 0;
}
```

Out-of-Bound Array Access

- **What is out-of-bound array access?**
  - The array index is out of the allowable range
  - The range is specified when array is created

```
int main()
{
    int size=10;
    float *pArray=NULL;
    pArray=(float *)malloc(sizeof(float)*size);

    /*access the array*/
    for(int i=0;i<20;i++)
        pArray[i]=float(i);

    free(pArray);
    return 0;
}
```

Memory Leak

- **What is the potential problem for dynamic memory allocation?**
  - The memory may not be released (freed)
  - Completely depend on programmer’s design

- **Memory leak**
  - Memory which is no longer needed is not released
  - They can exhaust available system memory as an application runs

Memory Leak

- Example

```
int main()
{
    int size=10;
    float *pArray=NULL;
    pArray=(float *)malloc(sizeof(float)*size);

    /*access the array*/
    for(int i=0;i<size;i++)
        pArray[i]=float(i);

     return 0;
}
```

Memory Leak

- Example

```
float* CreateArray(int size)
{
    return (float *)malloc(sizeof(float)*size);
}

int main()
{
    float* pArray=CreateArray(10);

    /*access the array*/
    for(int i=0;i<10;i++)
        pArray[i]=float(i);

     return 0;
}
```

Dangling Pointer

- **Pointers that do not point to a valid object of the appropriate type**
  - Usually a pointer which points to a dynamically allocated array which has been freed

```
float* InitArray(int size)
{
    float* pArray=(float *)
           malloc(sizeof(float)*size);
    for(int i=0;i<size;i++)
         pArray[i]=0

    free(pArray);

    return pArray;
}
```

```
int main()
{
    float* pArray=InitArray(10);

    /*access the array*/
    for(int i=0;i<10;i++)
         pArray[i]=float(i);

     return 0;
}
```