# Programming in C II

# Content

- Program Looping
- Decision Making
- Arrays

# Program looping

- Introduction to program looping
  - There are two classes of program loops which are *Unconditional* and *Conditional*
  - **Conditional loop:** The iterations are halted when a certain condition is true. ***Rational operators*** are helpful to set up the condition required to control a conditional loop
  - **Unconditional loop:** Is repeated a set number of times.

# Program looping

- Rational operators
  - Allow the comparison of two expressions
  - For example:

```
x > y
z >= x
```

# Program looping

- Boolean Expression
  - Boolean data type
    - It is used in creating a Boolean variable.
    - For example:

      ```
      bool lightsOn = true;
      ```
    - The value of a Boolean variable is always true or false.
    - Include the header *stdbool.h* if a program has a Boolean variable.

# Program looping

- Boolean Expression
  - Comparison operator (Rational operator)
    - A binary operator that takes two operands whose values are being compared
    - The result of the comparison is either true or false

# Program looping

- The Rational operators

| Name | Symbol | Usage | Answer |
|---|---|---|---|
| **Greater than** | **>** | 2 > 4 | false |
| **Greater than or equal to** | **>=** | 9 >= 7 | true |
| **Less than** | **<** | 8 < 10 | true |
| **Less than or equal to** | **<=** | 4 <= 6 | true |
| **Equal to** | **==** | 3 == 3 | true |
| **Not equal to** | **!=** | 3 != 3 | false |

# Program looping

- Logical Operators
  - Used to create more sophisticated conditional expressions which can then be used in any of the C looping or decision making statements
  - When expressions are combined with a logical operator, either TRUE (i.e., 1) or FALSE (i.e., 0) is returned.

# Program looping

- The Logical operators

| Name | Symbol | Usage | Operation |
|---|---|---|---|
| **Logical AND** | **&&** | exp1 && exp2 | Requires both **exp1** and **exp2** to be TRUE to return TRUE. Otherwise, the logical expression is FALSE. |
| **Logical OR** | \|\| | exp1 \|\|exp2 | Will be TRUE if either (or both) **exp1** or **exp2** is TRUE. Otherwise, it is FALSE. |
| **Logical NOT** | **!** | !exp1 | Negates (changes from TRUE to FALSE and visa versa) the expression. |

# Program looping

- Loop structures in C
  - C has three looping structures:
    - for loop
    - while loop
    - do…while loop

# Program looping

- **`for`** loop
  - C’s form of an unconditional loop
  - The basic syntax of the `for` statement is,

```
for ( initialization expression; test expr;
  increment expr )
   program statement ;
```

Example

```
sum=10;
for (i=0; i<6; ++i)
 sum = sum+i;
```

# Program looping

- **`for`** loop
  - Control expressions are separated by ; not ,
  - The primary purpose of a for loop is repeat statements with known number of iterations
  - If there are multiple C statements that make up the loop body, enclose them in brackets { }

# Program looping

- **for** loop
  - More examples

```
for (int i = 0; i < 10; i++){
 printf("%d\n",i);
}


for (int i = 1; i <= 5; i++){
printf("Hello!\n“);
printf(" *\n“);
}
```

# Program looping

- **`while`** loop
  - A mechanism for repeating C statements while a condition is true.
  - The basic syntax of the `while` statement is,

```
while(control expression)
    program statement ;
```

# Program looping

- `while` loop
  - The `control expression` provides an entry condition
  - If the expression evaluates to true, the statement executes the loop and the expression is then reevaluated. If it again evaluates to true, the statement executes again.
  - The `program statement` (body of the loop) continues to execute until the expression is false.
  - A loop that continues to execute endlessly is called an **infinite loop**.
  - To avoid an infinite loop, make sure that the control expression will be false at some point during the execution.

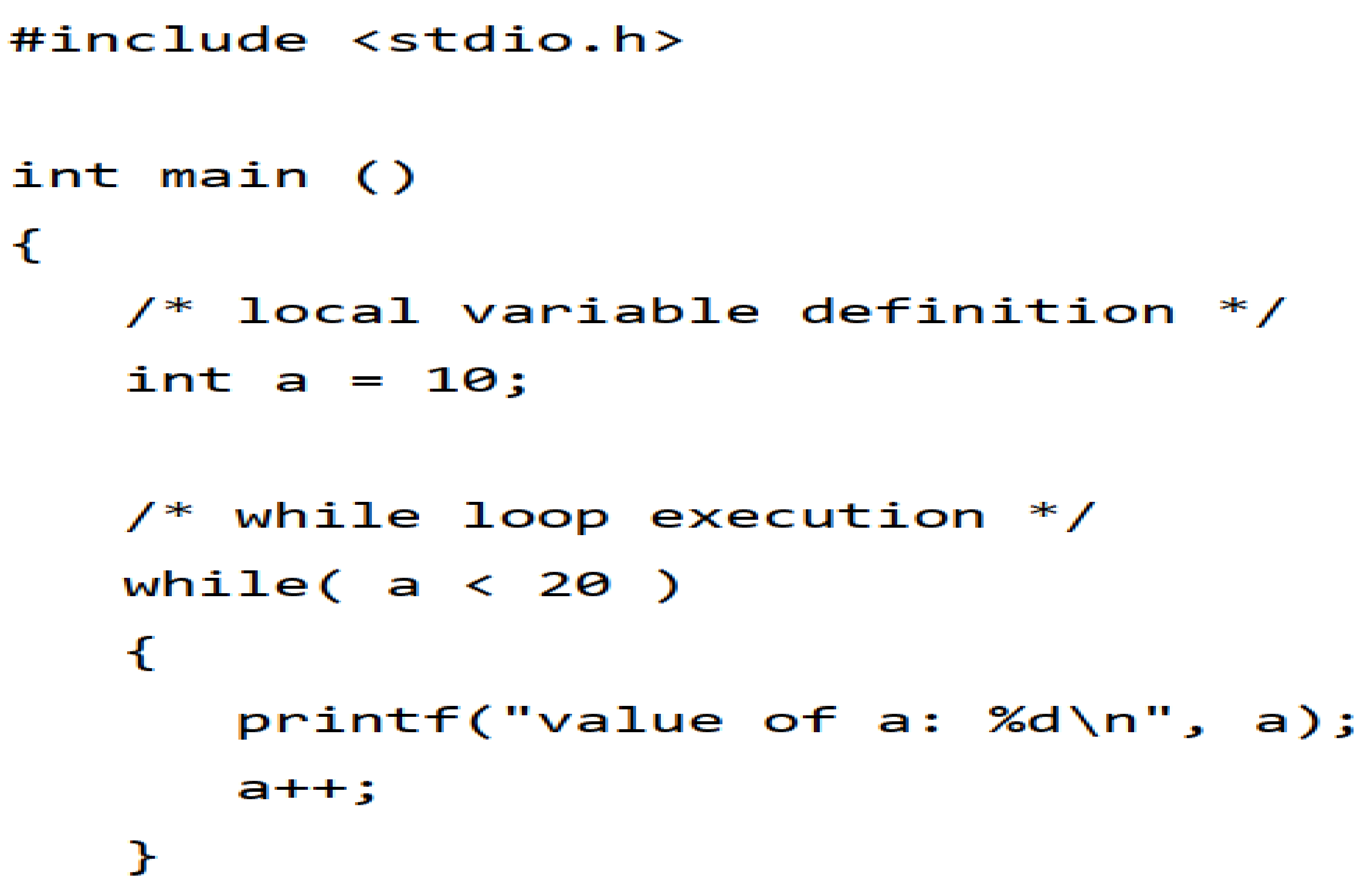
# Program looping

- **`while`** loop
  - Example program

```
i=1; factorial=1;
while (i<=n) {
factorial *= i;
i=i+1;
}
```

# Program looping

- **`while`** loop
  - Example program



```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
    }
```

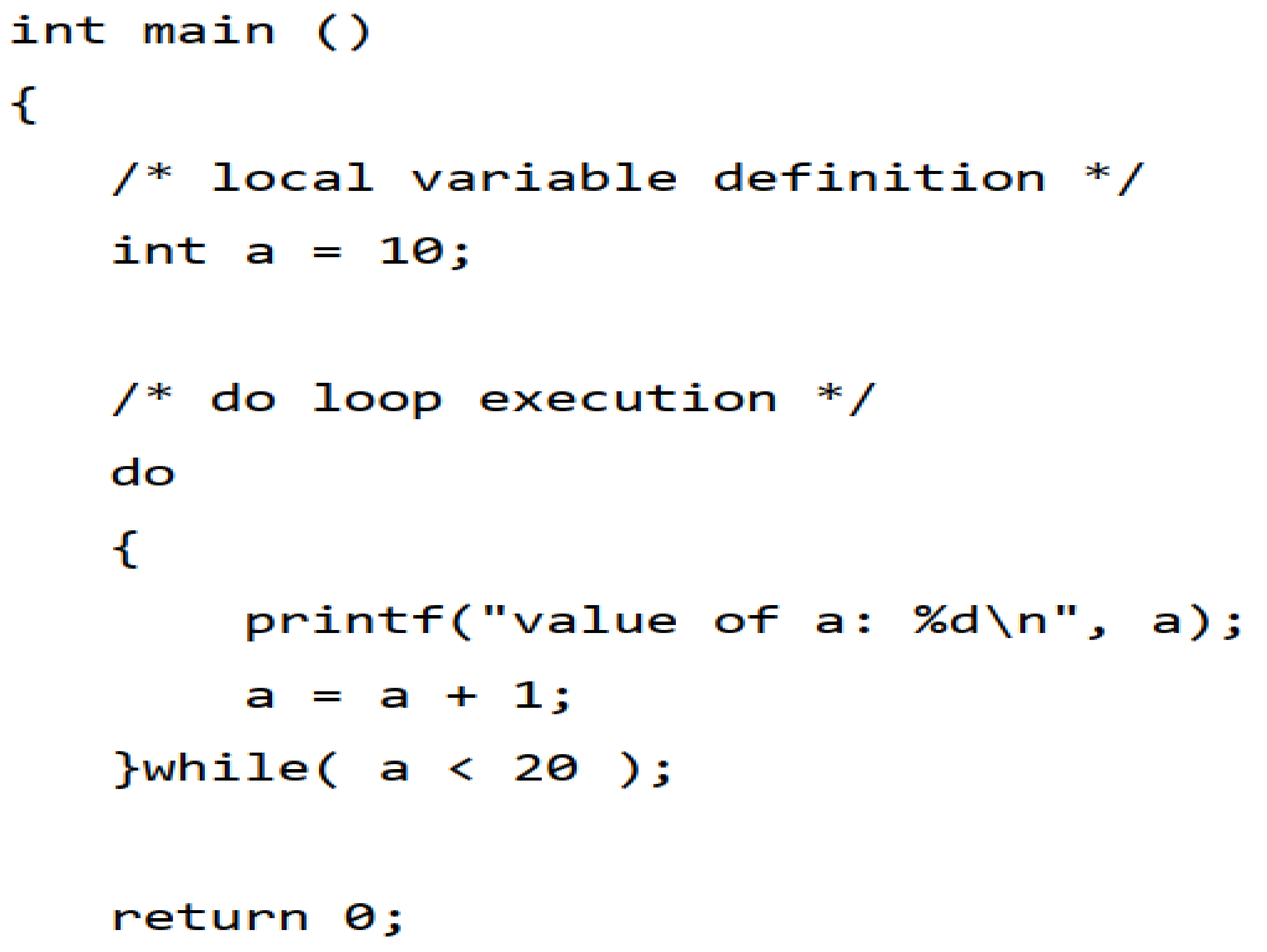
# Program looping

- **`do...while`** loop
  - A variant of the `while statement` in which the condition test is performed at the "bottom" of the loop

```
do
        program statement ;
while ( control expression );
```

# Program looping

- **`do...while`** loop
  - Example



```
int main ()
{
   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do
   {
       printf("value of a: %d\n", a);
       a = a + 1;
   }while( a < 20 );

   return 0;
```

# Program looping

- **do...while** loop
  - Example

```
main()  {
        int value, r_digit;
        printf("Enter the number to be reversed.\
           n");
        scanf("%d", &value);
   do {
        r_digit = value % 10;
        printf("%d", r_digit);
        value = value / 10;
   } while (value != 0);
        printf("\n");
   }
```

# Program looping

## Pre-test loop

- Is a loop which the loop condition is evaluated before executing the body of the loop.
- while and for loops are called pretest loops.

## Post-test loop

- Is a loop which the loop condition is evaluated after executing the body of the loop.
- do. . .while loop is a called posttest loop.

# Decision Making

- Used to have a program execute different statements depending on certain conditions
  - **C has three decision making statements:**
    - **`if`**
    - **`if-else`**
    - **`switch`**

# Decision Making

- **`if`** statement
  - It executes an action if and only if the condition is true.
  - The **`if`** statement allows branching (decision making) depending upon a condition. Program code is executed or skipped . The basic syntax is:

```
        if (condition)
        program statement ;
```

  - If the control expression is TRUE, the body of the if is executed. If it is FALSE, the body of the if is skipped.

# Decision Making

- **`if`** statement
  - Example

```
double radius = 3;
    if(radius > 0){
      printf(“Valid\n”);
      }
      grade=95;
  if (grade>=90)
  printf("\nCongratulations!");
  printf("\nYour grade is "%d",grade);
```

# Decision Making

- **`if – else`** statement
  - It executes an action if condition is either true or false. The syntax is:

```
if(condition){
 Statement 1;
} else{
Statement 2;
}
```

  - If the expression is TRUE, statement1 is executed; statement2 is skipped.
  - If the expression is FALSE, statement2 is executed; statement1 is skipped.

# Decision Making

- **`if – else`** statement
  - Example:

```
double radius = 3;
if(radius > 0){
   printf(“Valid\n”);
} else{
    printf(“Invalid\n”);
}
```

# Decision Making

- **if – else Ladder**
  - It is used when there are more than two possible action based on different conditions. The syntax:

```
if(condition1){
        Statement 1;
        } else if (condition2){
        Statement 2;
        }
    ……
    else if(conditionN){
        Statement N;
    }else{
    Default_Statement;
    }
```

# Decision Making

- **if – else Ladder**
  - Example:

```
int main(){
            int day;
            printf("Enter day number: ");
            scanf("%d", &day);
            if(day==1){
              printf("SUNDAY.");
            }else if(day==2){
              printf("MONDAY.");
            }else{
            printf("TEST AGAIN");
            }
            return(0);
            }
```

# Decision Making

- **`switch`** statement
  - The `switch` statement presents a better way of writing a program which employs an `if-else` ladder. The syntax:

```
switch(control expression){
    case constant1:
      Statements;
      break;
     case  constant2:
     Statements;
     break;
  default:
    Statements;
    break;
    }
```

# Decision Making

- **`switch`** statement
  - The keyword `break` should be included at the end of each case statement
  - If the `break` statement is omitted then the statement for subsequent cases will also be executed. Even through a match has already take place
  - The `default` clause is optional

# Decision Making

- **`switch`** statement
  - Example:

```
switch(n) {
case 12:
    printf("Value is 12\n");
    break;
case 25:
    printf("Value is 25\n");
    break;
case 99:
    printf("Value is 99\n");
    break;
default:
    printf("Number is not known\n");
}
```

# Arrays

- Introduction to Array Variables
  - An array is defined to be a group of logically related data items of similar type stored in contiguous memory location is called array.
  - Arrays are a data structure which hold multiple values of the same data type.
  - An array is a group of elements (data items) that have common characteristics (Ex:Numeric data, Character data etc.,) and share a common name.
  - The elements of an array are differentiated from one another by their positions within an array.

# Arrays

- One-dimension array
  - A one-dimensional array is an array in which the components are arranged in a list form.
  - The general form for declaring a one-dimensional array is:

```
    dataType    arrayName[arrayLength];
```

  - `arrayLength` is a positive integer that specifies the number of components in the array.
  - During declaration consecutive memory locations are reserved for the array and all its elements.

# Arrays

- One-dimension array
  - Example of array declaration:

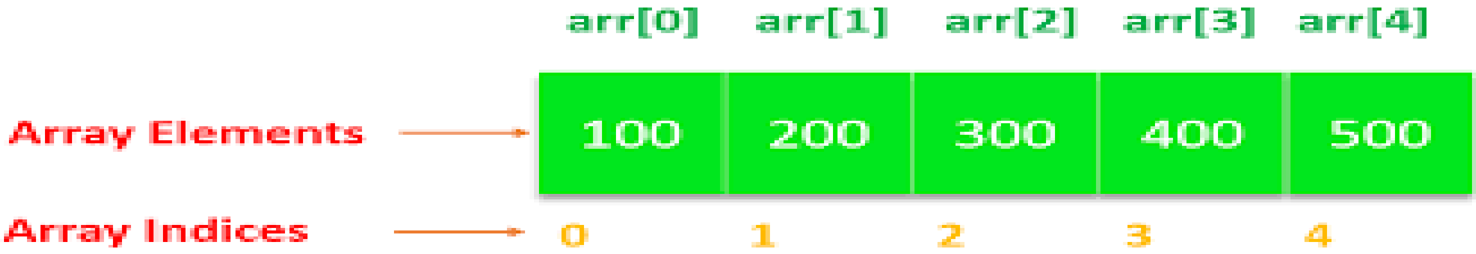
    ```
    int    numArray[3];
    ```

  - Declares an array `numArray` of three components; each component is of type `int`.
  - Each array component is called a value or an element.
  - Each value is stored at a specific position called an index.
  - To access a value stored in an array, you need to know its index.

# Arrays

- One-dimension array
  - Example of an array:

```
int      arr[5];
```



ARRAY IN C

# Arrays

- One-dimension array
  - Accessing array elements
    - The syntax used for accessing an array component is:

```
arrayName[indexExp]
```

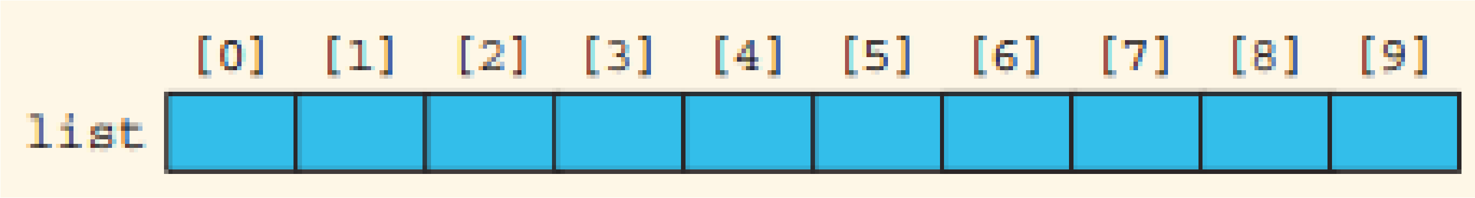
    - `indexExp` is the index of the value accessed.
    - Consider the following statement:

```
int list[10];
```

    - This statement declares an array list of 10 components.

# Arrays

- One-dimension array
  - **Accessing array elements**
    - The declared array named list



    - The components of the array are list[0], list[1], list[2] ,…., list[9].

# Arrays

- One-dimension array
  - **Array initialization**
    - Array can be initialized at declaration. The initial values are enclosed in braces.
    - Example:

      ```
      int values[8] = {1,2,3,4,5,6,7,8};
      ```

    - A common programming error is ***out-of-bounds array indexing.***

# Arrays

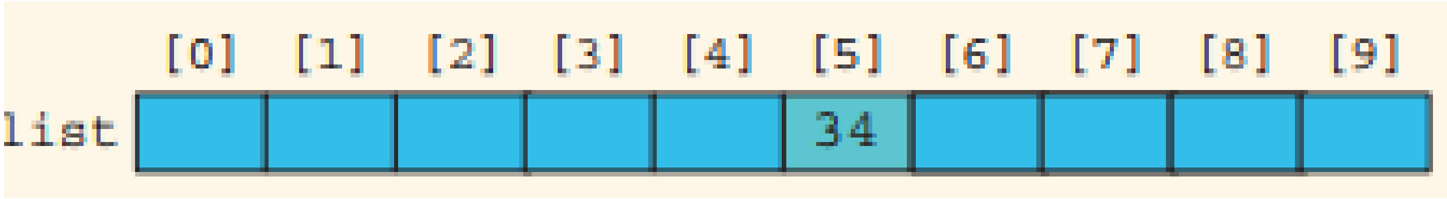
- One-dimension array
  - **Out-of-bounds indexing**
    - The index is in bounds if `index >= 0` and `index <= ARRAY_SIZE - 1`
    - If either `index < 0` or `index > ARRAY_SIZE - 1,` then the index is ***out of bounds***

# Arrays

- One-dimension array
  - **Assigning elements to an array**
    - Example 1:

```
list[5] = 34;
```

    - Stores 34 at the sixth position of array list which has index 5.

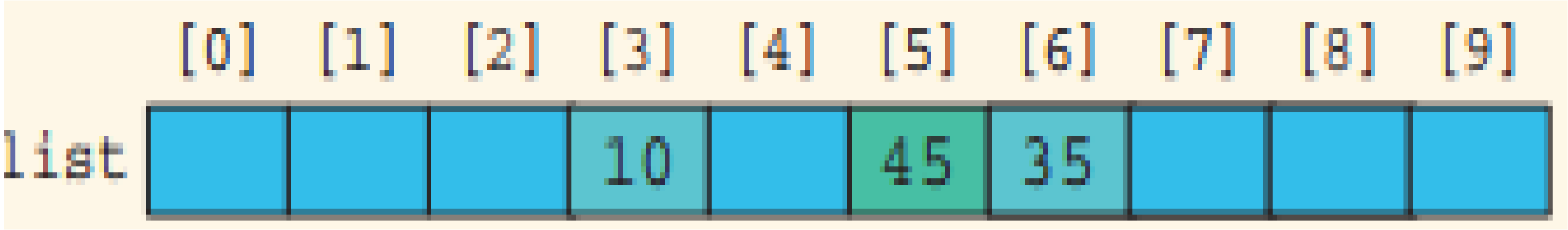


|  | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|---|
| list |  |  |  |  |  | 34 |  |  |  |  |

# Arrays

- One-dimension array
  - **Assigning elements to an array**
    - Example 2:

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

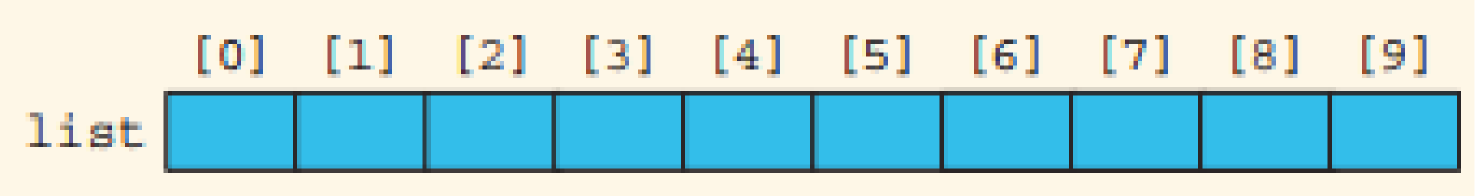


|  | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|---|
| list |  |  |  | 10 |  | 45 | 35 |  |  |  |

# Arrays

- One-dimension array
  - **Assigning elements to an array**
    - Example 3:

```
list[10] = 14;
```



|  | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|---|
| list |  |  |  |  |  |  |  |  |  |  |

- ***Out-of-bounds assignment error.***
- Array list has no memory location with index 10.

# Arrays

- One-dimension array
  - **Processing one-dimension array**
    - Can be accomplished using a loop. `For` loop works better.
    - Data input example:

```
int list[100];
int i;
for (i = 0; i < 100; i++){
 printf(“Enter value %d\n”, i+1);
 scanf(“%d”,&list[i]);
}
```

# Arrays

- One-dimension array
  - **Processing one-dimension array**
    - Data output example:

```
for (i = 0; i < 100; i++){
   printf(“%d\n”, list[i]);
}
```

# Arrays

- Multi-dimensional arrays
  - Multi-dimensional arrays have two or more index values which are used to specify a particular element in the array.
  - For example, a two-dimension (2D) array

    ```
    num[i][j]
    ```

  - The first index value `i` specifies a row index, while `j` specifies a column index.

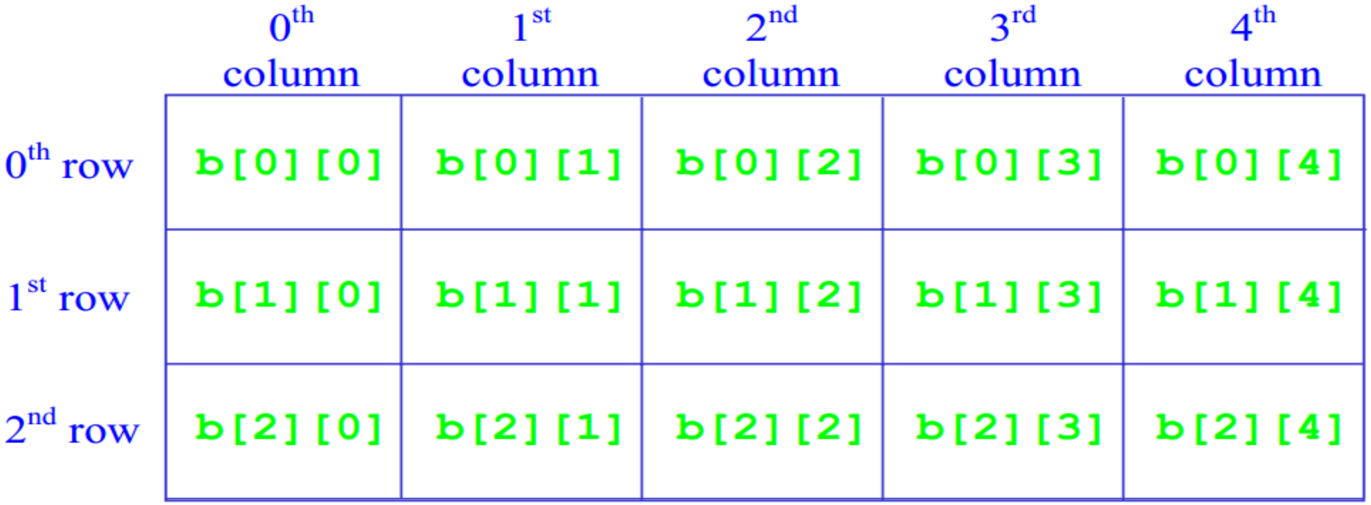
# Arrays

- Multi-dimensional arrays
  - Declaring multi-dimensional arrays is similar to the one-dimension(1D) array case:

```
int a[13]; /* declare 1D array */
float b[3][8]; /*declare 2D array */
double c[6][5][6]; /* declare 3D
 array */
```

# Arrays

- Multi-dimensional arrays
  - **Two-dimension array**
    - A useful way to picture a 2D array is as a grid or matrix.
    - Example see: `float b[3][8];`



|  | 0th column | 1st column | 2nd column | 3rd column | 4th column |
|---|---|---|---|---|---|
| 0th row | b[0][0] | b[0][1] | b[0][2] | b[0][3] | b[0][4] |
| 1st row | b[1][0] | b[1][1] | b[1][2] | b[1][3] | b[1][4] |
| 2nd row | b[2][0] | b[2][1] | b[2][2] | b[2][3] | b[2][4] |

# Arrays

- Multi-dimensional arrays
  - **Initializing 2D array**
    - The procedure is similar to that used to initialize 1D arrays at their declaration. For example:

```
int num[2][3]={4,8,12,19,6,-1};
```

    - The array is initialized row by row. Thus, the above statement is equivalent to:

```
num[0][0]=4;  num[0][1]=8;
num[0][2]=12;   num[1][0]=19;
num[1][1]=6;  num[1][2]=-1;
```

# Arrays

- Multi-dimensional arrays
  - **Initializing 2D array**
    - To make your program more readable, you can put the values to be assigned to the same row in inner curly brackets. For example:

```
int num[2][3]={{4,8,12},{19,6,-1}};
```

    - If there are fewer initialization values than array elements, the remainder are initialized to zero.

# Arrays

- Multi-dimensional arrays
  - **Working with 2D array**
    - Example:

```
double temp[2][3],sum=0;
int i,j;
 for (i=0; i<2; ++i)
        for (j=0; j<3; ++j)
               sum += temp[i][j];
```