One-Dimensional Arrays

- The solutions of some problems are possible only if the data to be processed are stored in the main memory, and are processed as a list.

  For example, in order to read the test scores of 20 students, compute their average, and find out how many students have a test score above the average, we must be able to represent all 20 test scores inside the main memory, so that we can compare each one to the average test score after it has been computed.

- A List of variables with the same data type is created in the C++ programming language as a *one-dimensional array*.

Defining and Referencing One-Dimensional Arrays

- A **one-dimensional array** is a list of variables with the same data type.
- You define a one-dimensional array as follows:

<data-type>   <array-name> [size];

<array-name>	is the name of the array or the list,
<data-type>	is the data type of the variables in the list, and
size	is the number of variables in the list.

Example A1

```
char  letter[5];        /*list of 5 variables to hold character values */
int  idlist[5];         /* list of 5 variables to hold integer values */
double  scores[10];     /*list of 10 variables to hold double precision floating point values */
```

- An individual variable of a one-dimensional array is referred to as an **element of the array**.
- It has a name that consists of the name of the array and an *index* enclosed between square brackets, and is called an **indexed variable**.
- The indexed variables of an array named *A* and of size *n* are: A[0], A[1], . . . , and A[*n*-1].
- The indexed variables for the arrays defined above are given as follows:

  - letter[0], letter[1], letter[2], letter[3], and letter[4].
  - idlist[0], idlist[1], idlist[2], idlist[3], and idlist[4].
  - scores[0], scores[1], scores[2], scores[3], . . . , and scores[9].

Exercise A1*

Write a declaration statement to define the array *scores* of 20 double precision floating-point elements.

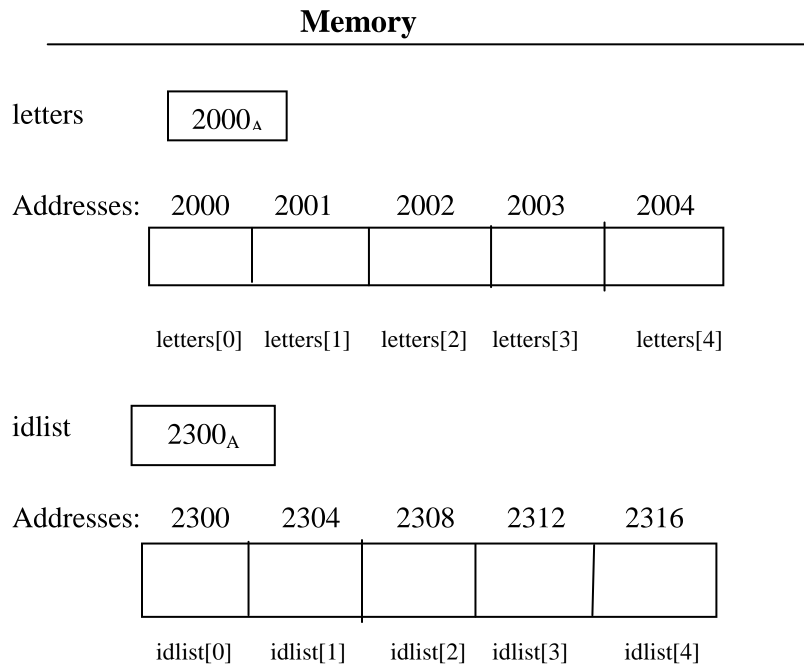
Indexed Variables and Memory Locations

- Contiguous memory locations are allocated in the memory for the indexed variables of an array.
- **The name of an array is a pointer constant** that holds the address of the first element of the array.

Example A2

The following two arrays are represented in memory as specified below:

```
char   letters[5];
int   idlist[5];
```



- Note that a character variable occupies a byte of memory location whereas an integer value occupies a four-byte memory location in an Intel Pentium computer.

Exercise A2*

Show the memory representation of each of the following arrays (you should make up memory addresses if they are needed):

a) char letters[5];
b) int numbers[5];

Using Indexed Variables

- In a C++ program, an indexed variable is used in the same way that a simple variable with the same data type is used.
- The index of an indexed variable does not have to be a constant value: It can be any arithmetic expression (called **index expression**) that evaluates to an index of the array.

Example A3

Given the following definitions of variable *tscore* and array *scores*:

```
double   tscore,
         scores[10];
```

The indexed variables of the array *scores* could be used as follows:

```
scores[2]  =  90.5;
scores[0]  =  scores[2] - 6;
tscore  =  scores[2] + 3;
scores[1]  =  tscores - 10;
cin   >>   scores[5];
cout   <<   scores[2] + 1;
scores[5]++;
```

With the following definitions of the variables *i* and *j*,

```
int   i = 2,   j = 4;
```

Index expressions may be used to specify the elements of the array *scores* as follows:

Using Index Expressions	is processed as
scores[i] = 85.5;	scores[2] = 85.5;
scores[i + 4] = 96.4;	scores[6] = 96.4;
scores[2 * j - 1] = scores[i +3];	scores[7] = scores[5];
scores[i ++] = 75.0;	scores[2] = 75.0;
scores[-- j] = 87.5;	scores[3] = 87.5;

- The index expression to access an element of an array must evaluate to a value in the range 0 to (*size*-1), where *size* is the number of elements in that array.

Initializing the Elements of a One-Dimensional Array

- You can initialize the elements of a one-dimensional array only when it is defined.
- This is done by following its definition with the assignment operator, which is followed by the list of the initial values enclosed between braces:
  - The first value in the list is assigned to the first element of the array,
  - The second value to the second element, ... , etc.
  - If there are not enough values for all the elements of the array, the rest of the elements are initialized to 0 (for numeric values) and the **null character** (for characters).

Example A4

Declaration Statement	**Initialization of the Elements of the Array**
int values[5] = { 10, 20, 30, 40, 50 };	values[0] = 10 values[1] = 20 values[2] = 3 values[3] = 40 values[4] = 50
Int values[5] = {10, 20, 30 };	values[0] = 10 values[1] = 20 values[2] = 30 values[3] = 0 values[4] = 0
char letters[5] = { 'A', 'B'};	letters[0] = 'A' letters[1] = 'B' letters[2] = '\0' letters[3] = '\0' letters[4] = '\0'
int values[ ] = {10, 20, 30, 40};	values[0] = 10 values[1] = 20 values[2] = 30 values[3] = 40

- You do not have to specify the size of an array in a declaration statement with initial values:
  The number of elements will be exactly the same as the number of initial values provided in the list of values.
- **An array of characters may also be initialized with a string constant when it is defined** in one of the following ways:

  char <array-name> [size] = <string-constant>;

  or

  char <array-name> [size] = { <string-constant> };

- The characters of the string constant are stored into the elements of the array, with a Null character at the end.

Example A5

Declaration Statement	Initialization of the Elements of the Array
char name[6] = { "John" };	name[0] = 'J' name[1] = 'o' name[2] = 'h' name[3] = 'n' name[4] = '\0' name[5] = '\0'
char car[ ] = "ford";	car[0] = 'f' car[1] = 'o' car[2] = 'r' car[3] = 'd' car[4] = '\0'

Exercise A3*

1. Write the declaration statements to define the following arrays:
   a. Array *numbers* of 15 integer elements initialized with the values: 12, 4, 8, 40, -5, and 10.
   b. Array *name* initialized with the string constant: "John Doe".
2. Show the memory representation of each of the following arrays (you should make up memory addresses if they are needed):
   a. char alist[8] = {'A', 'B', 'C', 'D', 'E'};
   b. int numlist[8] = { 15, -10, 25, 30};

Exercise A4

1. Show the memory representation of each of the following arrays (you should make up memory addresses if they are needed):
   a. *int newlist[ ] = {10, 20, 30, 40, 50, 60};*
   b. *char name1[10] = { "BE HAPPY "};*
2. Given the following definitions of array *list* and variables *i* and *j* :

   int list[5] = { 5, 10, 15, 20, 25}, i = 5, j = 2;

   a. what is wrong with the statement: *cin >> list[i];*
   b. show the memory representation of array *list* after the execution of each of the following statements:

      1. *list[j - 2] = 7;*
      2. *list[j]++;*
      3. *list[- - j] = list[4] + 5;*
      4. *i = j = 3;*
      5. *list[i ++] = list[j + 1] + 10;*
      6. *list[i] = 50;*

Processing One-Dimensional Arrays

- The indexed variables of a one-dimensional array are in general processed as a list of variables: Identical operations are performed on these variables by using a counter-controlled loop in which these operations are specified in the body-of-the-loop and the index variable is used as the loop counter.
- In general, this processing can be specified in C++ using a **for** structure as follows:

```
for(index = 0 ; index < size ; index ++ )
    <process element( index)>
```

- In the following examples, we will use the array of 10 integer elements named *list* and the named constant *SIZE* defined as follows:

```
#define   SIZE    10          /* number of elements in the array */
int   list [10];
```

1. Reading values into an Array

```
for (int i = 0;  i < SIZE;  i++)
{
    cout   <<   "\n value for element # \t"   <<   i + 1   <<   '\t';
    cin   >>   list[i];
}
```

2. Printing the Values of the Elements of an Array

```
for (int i = 0;  i < SIZE;  i++)
    cout   <<   endl   <<   list[i];
```

3. Computing the Total Value of the Elements of an Array

```
int totalvalue = 0;
for (int i = 0;  i < SIZE;  i++)
    totalvalue   += list[i];
cout     << endl     << "the total value is:\t"  << totalvalue;
```

©2011 G. Ndjatou

Exercise A5*

1. Given the following definition of array *list*: *int list[5] = {10, 20, 30, 40, 50};*
   Show the output of the following program segment:

```
for(int j = 0 ; j < 5 ; j ++ )
  cout   <<  endl  <<   (list[j] + j );
```

2. Array *list* is an array of 10 double precision values defined as follows*: double list[10];*
   a. Write a code segment to read values into array *list*.
   b. Write a code segment to compute and print the average value of the elements of array *list*.

Exercise A6

1. Given the following definition of array *list*: *int list[5] = {10, 20, 30, 40, 50};*
   Show the output of the following program segment:

```
for (int j = 4 ; j >= 0 ; j - - )
   cout  <<  endl  <<   list[j];
```

2. Given the following definition of array *letters*: *char letters[10];*
   a. Write a code segment to read 10 letters of the alphabet into array *letters*.
   b. Write a code segment to print the letters in array *letters* in reverse order. That means, if the input is: A B C D E F G H I J, the output should be: J I H G F E D C B A.

4. Looking for the Maximum Value in an Array

- To look for the maximum value in an array *list*, we use a variable named *maxvalue* to hold the current maximum. The algorithm is described in pseudo-code as follows:

  1. Set variable *maxvalue* to *list[0]*, and set the array index (i) to 1.
  2. As long as the array index (i) is less than *SIZE*, do the following:
     2.1 if current value of variable *maxvalue* is less than *list[i]*, assign *list[i]* to variable *maxvalue*
     2.2 increment the array index (i) by 1
  3. Output the maximum value (*maxvalue*)

The C/C++ code segment is provided as follows:

```
int maxvalue;             // to hold the maximum value
for (maxvalue = list[0],  int i = 1;  i < SIZE;  i ++)
      if (maxvalue < list[i])
            maxvalue = list[i];
cout     << "\n the maximum value in the array is:\t"     << maxvalue;
```

Exercise A7

Array *list* is an array of 10 double precision values defined as follows: *double list[10];*

a. Write a code segment to read values into array *list*.
b. Write a code segment to look for the minimum value in array *list* and print it.

5. Parallel Arrays

Given the following definitions of array *list1* and *list2* : *int list1[10], list2[10];*

a. Write a code segment to read values into array *list1*.
b. Write a code segment to add 5 to each element of array *list1* and to store the result into the corresponding element of array *list2*.

```
for ( int i = 0 ; i <  10;  i + + )
     cin  >>  list1[ i ];
for ( int i = 0; i < 10;  i+ + )
     list2[i] = list1[i] + 5;
```

Exercise A8

Arrays *list1*, *list2*, and *list3* are defined as follows:

int list1[5] = { 23, -12, 45, 9, 11},   list2[5] = {21, -4, 63, 21, -8},   list3[5];

1. Write a code segment to subtract each entry of array *list2* from the corresponding entry of array *list1*, and to store the result in the corresponding entry of array *list3*.
2. Write a code segment to multiply each entry of array *list1* by 5, and to store the result in the corresponding entry of array *list3*.

6. Partially Filled Arrays

- When a program to process information represented using arrays is being designed, the exact number of items to be processed is in general not known.
- Also, most programs are designed in such a way that the size of the data to be processed may vary from one execution of the program to another.
- Programmers usually deal with this problem by defining the array(s) with the largest size that the program could possibly need, and then keep track in the program of the last index used in the array(s).

Code segment to read one or more values into array *list*
(-99 is entered as the last dummy value)

```
int lastNdx;            // to hold the last index used in the array
int   avalue,           // to hold the value read from the keyboard
      j;
cin >> avalue;          // read the first value
for ( j = 0 ;  j < SIZE  &&  avalue  !=  -99 ;  j++)
{
      list[j] = avalue;
      cin >> avalue;    // read the nest value
}
lastNdx = j - 1;
```

- The loop condition here is “j < SIZE && avalue != -99“ :
  - We have to make sure that we do not input more values than there are elements in the array, and
  - We must also make sure that the dummy value -99 is not entered into the array.
- When the loop iterations stop, the current value of the index is one position past the last element used in the array.

Code segment to print the element used in the array

```
for (int j = 0; j <= lastNdx; j++)
      cout     << endl     << list[j];
```

- The loop condition here is “j <= lastNdx” instead of “j < SIZE”:
  The elements used in the array are list[0], list[1], .   .   ., list[lastNdx].

Exercise A9

A computer science class can hold a maximum of 20 students and you are asked to write a code segment to read students’ ID numbers into the array *idNum[20]* and their test scores into the array *scoreList[20]*. The dummy ID number -99 is used to end the entry of students’ ID numbers. You must also do the following:

a. Compute and print the average test score of the class.
b. Print each student’s ID number with its test score and how far away his test score is from the class average.

7. Arrays and Static Local Variables

Because memory locations for static local variables are allocated in the initialized data segment and not in the stack, large program constants such as arrays of constant values are often defined as static in a function for efficiency reasons.

Example: static int daysPerMont [13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Given the following class *DayOfYear* defined in example O9:

```
class  DayOfYear
{
    public:
        DayOfYear(int newMonth = 1, int  newDay = 1);    // constructor with default arguments
        void  input( );                                  // to input the month and the day
        void  output( );                                 // to output the month and the day
        int getMonth( );                                 // to return the month
        int getDay( );                                   // to return the day
    private:
        void  checkDate( );                      // to validate the month and the day
        int month;                               // to hold the month (1 – 12)
        int day;                                 // to hold the day (1 – 31)
    };
```

The member function *void checkDate( )* is redefined using a static local array to hold the number of days in every month as follows:

```
/*--------------------member function checkDate( )------------------------*/
/*   validate the date                                                    */
void  DayOfYear :: checkDate(  )
{
     static const int daysPerMonth[ 13 ] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( (month < 1) ||  (month > 12) || (day < 1) || (day > daysPerMonth [ month ]) )
    {
        cout   <<   endl   <<   "Invalid date";
        exit ( 1 );
    }
}
```

Exercise A10

Redefine the member function *void checkDate( )* of the class *Date* of exercise O7 so that it validates a date by using a static local array to hold the number of days in every month.

Passing an Array to a Function

- **The parameter that corresponds to an array** is specified in a function header as follows:

  <data-type>   <Parameter-name>[ ]

Example A6

The following are examples of function headers of functions that receive arrays as arguments:

1. *void readvalues (int list[ ], int size)*
2. *double computeAverage(double scores[ ], int size)*
3. *void addlist(int list1[ ], int list2[ ], int num, int size)*

- **You pass an array to a function** by passing the address of its first element: that means the name of the array.
- In addition to passing the address of the first element of an array to a function, you must also pass to it the size of the array so that it knows how many elements are in the array.

Example A7

Given the following definitions of arrays *idlist*, *numlist1*, *numlist2*, *testscore*, and variable *average*:

int idlist[20], numlist1[5] = {10, 20, 30, 40, 50,}, numlist2[5];

double testscore[5] = {92.0, 78.5, 86.0, 98.5, 76.0}, average;

The following are valid calls to the functions with the function headers provided in example A6.

1. *readvalues(idlist, 20);*
2. *average = computeAverage(testscore, 5);*
3. *addlist(numlist1, numlist2, 50, 5);*

- An array passed to a function must have a data type compatible with the data type of the corresponding parameter.
- Inside the body of a function, the name of the parameter, *<parameter-name>* is used to access the elements of the array passed to the function in the same way that the name of that array is used to access its elements.

Example A8

The definitions of functions *readvalues()*, *computeAverage()*, and *addlist()* with the function headers provided in example A6 follow:

```
/*---------------------------- function readvalues() --------------------------------------*/
/*   read values into an array                                                              */
void readvalue(int list[ ], int size)
{
     for(int j = 0; j < size; j++)
     {
          cout     << "\nEnter a value:\t";
          cin      >> list[ j ];
     }
     return;
}


/*---------------------------- function computeAverage()------------------------------------*/
/* compute the average of the elements of an array and return it                            */
double computeAverage(double scores[ ], int size)
{
     double total = 0;
     /*------------------------compute the total of all elements ----------------------*/
     for(int j = 0; j < size; j++)
     total += scores[j];

     /*--------------------- compute and return the average -------------------------*/
     return ( total / size );
}


/*---------------------------- function addlist -------------------------------------------*/
/*   add a value to each element of an array and build a new one                            */
void addlist(int list1[ ], int list2[ ], int num, int size)
{
     for(int j = 0; j < size; j++)
     list2[j] = list1[j] + num;
     return;
}
```

Example A9

Function template to find the maximum value in a list.

```
template < class T >
T findMax (  T  list[   ], int size )
{
     T maxvalue;                // to hold the maximum value
     for (maxvalue = list[0],  int i = 1;  i < size;  i ++)
               if (maxvalue < list[i])
                    maxvalue = list[i];
     return ( maxvalue );
}
```

Exercise A11*

a. Write the function with function header *int computeSumm(int list[ ], int size)* that receives as argument an array of integer values and its size, and then computes and returns the sum of the elements of the array.
b. Write the statement(s) to compute the sum of the elements of the array
   numlist = {3, 5, 8, 2, 4, 12, 32, 45, 2, 35}; (by calling the function in a.) and to print it.

Exercise A12

a. Write the function with function header *void addconst2(int list1[ ], int list2[ ], int num, int size)* that adds *num* to each element of array *list1* and stores the result in the corresponding entry of array *list2*.
b. Write the statement(s) to add 50 to each element of array *numlist* and to store the result in the corresponding entry of array *resultlist* by calling function *addconst2( )*. Array *numlist* and *resultlist* are defined as follows: *int resultlist[10], numlist[10] = { 2, 4, 5, 10, 35, 4, 21, 50, 3, 45};*

Exercise A13

a. Write the definition of function template *findMin* that finds the minimum value in an array.
b. Write the statement(s) to find and print the minimum value in each in the following arrays by calling function template *findMin*.
   int numlist[ ] = {3, 5, 8, 2, 4, 12, 32, 45, 2, 35};
   double dnumlist [ ] = { 24.5, 90.35, 15.3, 4.27, 11.0, 36.1};

Pointer Arithmetic

➢ If **pt** is a pointer variable that contains the address of a memory location with size *n*, and **a** is a positive integer constant, then the expression:

pt + a holds the address: pt + a * *n* and

pt – a holds the address: pt – a * *n* (with pt – a * *n* positive)

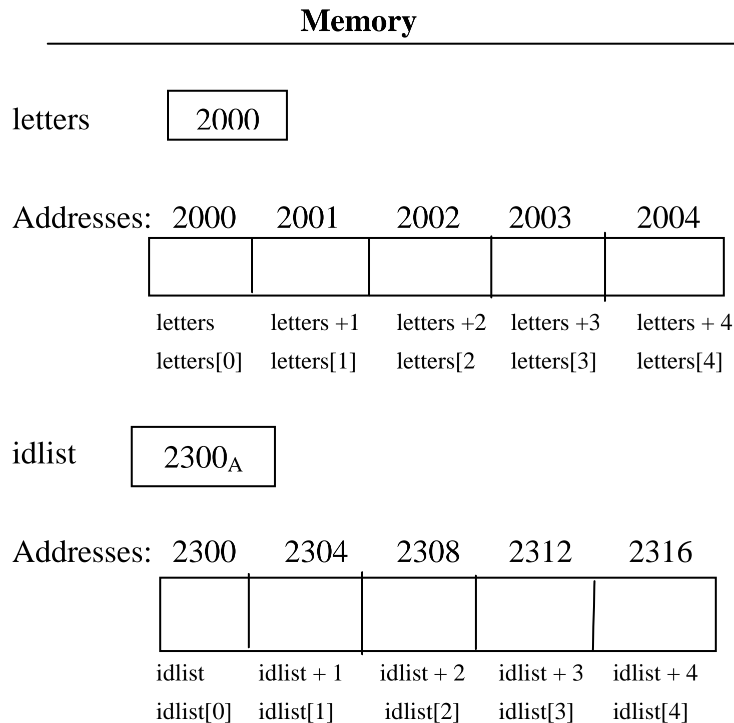
➢ A pointer may be initialized to 0 or **NULL** (symbolic constant defined in the header file *iostream*).

➢ A pointer with the value 0 or NULL is known as the **null pointer** and cannot be used to reference a memory location.

Example A10

Given the following definitions of arrays *letters* and *idlist* with their memory representations:

```
char letters[5];
int idlist[5];
```



We have the following:

Letters + 1 = = 2001	letters + 2 == 2002	letters + 3 = = 2003	letters + 4 = = 2004
idlist + 1 = = 2304	idlist + 2 = = 2308	idlist + 3 = = 2312	idlist + 4 = = 2316

➢ The elements of an array can be accessed either by using indexed variables or using pointer arithmetic.

Example A11

```
/*------- code segment to read the values into the array letters[ ] by using pointer arithmetic -------*/
for( int i = 0 ;  i < 5 ; i ++ )
     cin   >>   *(letters + i) ;

/*------- code segment to read the values into the array idlist[ ] by using pointer arithmetic --------*/
for( int i = 0 ;  i < 5 ; i ++ )
     cin   >>   *(idlist + i) ;

/* code segment to write the elements of array letters in reverse order by using pointer arithmetic */
for( char *pt = letters + 4 ;  pt > = letters ; pt - - )
     cout   <<   *pt ;
```

Dynamic Arrays

- A **dynamic array** is a sequence of consecutive memory locations allocated in the heap by using the **new** operator.
- You allocate a dynamic array as follows: *<type-pointer>* = **new** *<type> [size]*;

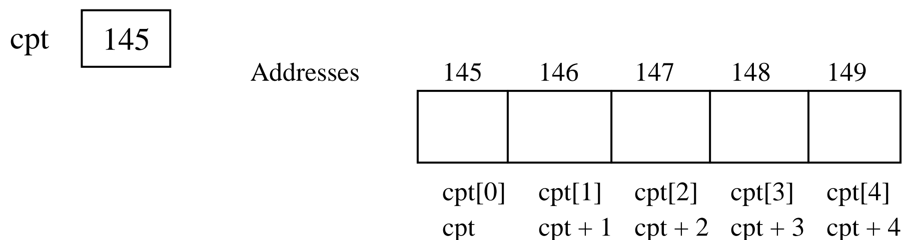
  Where *<type>* is any valid data type and *size* is the number of memory locations to be allocated.
- The **new** operator returns the address of the first of the memory locations allocated.

Example A12

A. The elements of a dynamic array to hold 5 character values are created as follows:

```
char  * cpt;
cpt = new char [ 5 ];
```

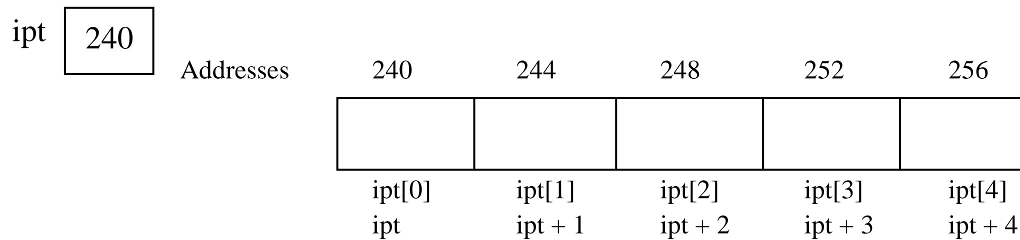
These memory locations are created in the heap as follows (we made up the addresses):



B. The elements of a dynamic array to hold 5 integer values are created as follows:

```
int  * ipt;
ipt = new int [ 5 ];
```

These memory locations are created in the heap as follows (we made up the addresses):



- The elements of a dynamic array may be accessed in two different ways:
  1. By subscripting the pointer variable that contains the address of the first element of the array.
  2. By using pointer arithmetic.

Example A13

Giving the following definition of pointer variable *ipt* and the allocation of the dynamic array:

```
int  * ipt;
ipt = new int [ 5 ];
```

The following three code segments read values into the dynamic array:

Code1

```
for (int i = 0 ; i < 5 ;  i ++ )
     cin  >>  ipt[ i ];
```

Code2

```
for (int i = 0 ; i < 5 ;  i ++ )
     cin  >>  *(ipt + i ) ;
```

Code3

```
int *p;
for (p = ipt ; p <  ipt  + 5 ;  p ++ )
     cin  >>  *p ;
```

- When you are done using a dynamic array, its memory locations must be released to the heap by using the **delete[ ]** operator.

Example A14

```
delete [ ] cpt;
delete [ ] ipt;
```

Exercise A14

1. What is the output of the following code segment?

```
int *aptr;
aptr = new int[ 10 ];
int i;
for ( i = 0 ; i < 10 ; i ++ )
    *(aptr + i ) = i;
for ( i = 0 ; i < 10 ; i ++ )
    cout << aptr [ i ];
cout << endl;
```

2. What is the output of the following code segment?

```
int *aptr;
aptr = new int[ 10 ];
int i;
for ( i = 0 ; i < 10 ; i ++ )
    aptr [ i ] = i;

int *pt = aptr + 9;
while ( pt >= 0 )
{
    cout << *pt << '\t';
    pt--;
}
cout << endl;
```

3. Write a code segment to do the following:
   a. Define a dynamic array to hold 10 integer values.
   b. Read values into the array (use a pointer to access the elements of the array).
   c. Print the elements of the array in reverse order (use a pointer to access the elements of the array).

One-Dimensional Arrays and Structures

- **A member of a structure can be a one-dimensional array**.

Example A15

The information about a student consists of its name, its ID number and his scores in 10 quizzes.

a) This information is represented using a structure as follows:

```
struct   StudentInfo
{
     string  name;
     int  idNum;
     double  testScores[10];
};
```

b) Write a code segment to read the information about a student and to compute his average test score.

```
StudentInfo  student;

/*-----------------------------read the student’s personal information ----------------------*/
cin   >>   student.name    >>    student.idNum;

/*-----------------------------read the student’s  test scores ----------------------------------*/
for (int tcount = 0 ;  tcount < 10 ; tcount ++)
     cin   >>   student.testScores[ tcount ];

/*--------------------compute and print his average test scores ----------------------------*/
double totalScore = 0;
for (int tcount = 0 ;  tcount < 10 ; tcount ++)
     totalScore  +=   student.testScores[ tcount ];
cout   <<  endl   <<   “Average test score is:\t”   <<   totalScore /10;
```

Exercise A15

The information about a salesperson consists of his name, ID number, and his total sales in each of the twelve months of a year held in an array named *monthlySales*.

1. Write the definition of a structure type named *SalespersonInfo* to represent this information.
2. Define a structure variable named *salesperson* of the structure type *SalespersonInfo*.
3. Write a code segment to read the values of the member variables of structure variable *salesperson*.
4. Write a code segment to compute and print its average monthly sale.

- **The members of an array can be a structure type.**

Example A16

a. Using the structure type *StudentInfo* defined in example A15, define an array of structures named *studentList* of 20 students.
b. Write a code segment to read the information about each student in the array and to compute and print his average test score.

```
StudentInfo  studentList [20];

/*------------------------ --------------------------------------------- ---------------------------------*/
for (int scount = 0 ; scount < 20 ; scount ++)
{
    /*-------------------------read student # scount's personal information ------------------------*/
    cin   >>   studentList[scount].name    >>    studentList[scount].idNum;

    /*------------------------read student # scount's  test scores --------------------------------------*/
    for (int tcount = 0 ;  tcount < 10 ; tcount ++)
        cin   >>   studentList[scount].testScores[ tcount ];

        /*--------------compute and print student # scount's average test scores -----------------*/
        double totalScore = 0;
        for (int tcount = 0 ;  tcount < 10 ; tcount ++)
                totalScore  +=   students[scount].testScores[ tcount ];
            cout   <<  endl   <<  "Average test score of student #"   <<   scount +1  <<  " is:\t"
                    <<  totalScore /10;
}
```

Exercise A16

1. Using the following structure type *ProductInfo*, define an array of structures named *productList* of 15 products.

```
struct    ProductInfo
{
    int   prodNum;              //  to hold a product number
    double  unitPrice;          //  to hold a product unit price
    int  quantity;              //  to hold a product quantity
};
```

2. Write a code segment to read the information about each product in the array.
3. Write a code segment to compute and print the price of each product in the array.

Exercise A17

1. Using the structure type *SalespersonInfo* defined in exercise A15, define an array of structures named *salespersonList* of 30 salespersons.
2. Write a code segment to read the information about each salesperson in the list.
3. Write a code segment to compute and print the total (yearly) sale of each salesperson in the list.

Arrays of Objects

- **The members of an array can have a class data type**: that means that they can be objects.
- You can define an array of objects of a class only if that class has a default constructor.
- The data members of the elements of an array of objects are all initialized with the default constructor.

Example A17

Assume given the following class **DemoA**:

```
class  DemoA
{
    public:
        DemoA( int num1= 0, int num2 = 0);     // constructor with default arguments
        void readValues(void);                 // to read the values of the member variables
        void  setValue1(double num);           // to set the value of the first member variable
        void  setValue2(double num);           // to set the value of the second member variable
        double  getValue1(void);               // to return the value of the first member variable
        double  getValue2(void);               // to return the value of the second member variable
        double  getAverage( );                 // to compute the average of both values
    private:
        double computeSum( );                  // to compute the sum of both values
        double val1;                           // the first member variable
        double val2;                           // the second member variable
};
```

We define the array *objList* of ten objects as follows:

```
DemoA  objList [10];        //  objList[i].val1 = 0  and  objList[i].val2 = 0
```

The member variables *val1* and *val2* of each object element of the array are set to 0.

a. The following code segment reads values into the member variables *val1* and *val2* of each object element of the array:

```
for (int i = 0 ; i <  10 ;  i ++)
{
     objectList[ i ].readValues(  );
}
```

b. The following code segment computes and prints the average value of the member variables *val1* and *val2* of each object elements of the array:

```
for (int i = 0 ; i < 10 ; i ++)

     cout  <<  endl  <<  objectList [ i ].getAverage( );
```

Exercise A18

Assume given the following class *Rectangle*:

```
class  Rectangle
{
      public:
           Rectangle(double len = 1, double wth = 1);  //  constructor
           void  input(void);                          // read the values for the length and the width
           void  print(void);                          // output the length and the width
           double  computeArea( );                     // compute the area of the rectangle
      private:
           double length;                              // the length of the rectangle
           double  width;                              // the width of the rectangle
};
```

1. Write the statement to define the array *rectangleList* of ten objects of the class *Rectangle*.
2. Write a code segment to read values into the member variables of each object element of this array.
3. Write a code segment to compute and print the area of each object element of this array.
4. Write the statement to define the dynamic array *dynamicRList* of ten dynamic objects of the class *Rectangle*.
5. Write a code segment to read values into the member variables of each object element of this array.
6. Write a code segment to compute and print the area of each object element of this array.

Exercise A19

1. Write the statements to define the dynamic array *employeeDList* of 10 objects of the class *Employee* defined in exercise O9.
2. Write a code segment to read values into the member variables of each object element of this array.
3. Write a code segment to output the personal and pay information of each employee.

Exercise A20

1. Write the statements to define the dynamic array *BonusEmployeeDList* of 10 objects of the class *BonusEmployee* defined in exercise O12.
2. Write a code segment to read values into the member variables of each object element of this array.
3. Write a code segment to output the personal and pay information of each employee.

Exercise A21 Extra Credit

1. Write the statements to define the dynamic array *HourlyEmployeeDList* of 10 objects of the class *HourlyEmployee* defined in exercise O13.
2. Write a code segment to read values into the member variables of each object element of this array.
3. Write a code segment to output the personal and pay information of each employee.

Arrays as Class Data Members

- The following example illustrates the use of an array as a class data member: the class has a one-dimensional array as a data member and just one default constructor to initialize the members of the array.

```
/*--------------------------------------Salesperson.h-----------------------------------------*/
/*   Salesperson class definition.
     Member functions are defined in Salesperson.cpp.                                       */

#ifndef  SALESPERSON_H
#define SALESPERSON_H

class Salesperson
{
    public:
      Salesperson(  );                    // constructor
      void getSalesFromUser(  );          // input sales from keyboard
      void setSales( int, double );       // set sales for a specific month
      double getSales( int );             // return sales for a specific month
      double totalAnnualSales(  );        // add all monthly sales
      void printAnnualSales(  );          // summarize and print yearly total sale
   private:
     double sales[ 12 ];                  // 12 monthly sales figures
};

#endif
```

```
/*----------------------SalesPerson.cpp -----------------------------------*/
/* Member functions of the class Salesperson.
*/
#include <iostream>
#include <iomanip>
#include  <cstdlib>
using namespace std;

#include "Salesperson.h"       //include SalesPerson class definition

/*-------------------Default constructor --------------------------------*/
/* initialize elements of array sales to 0.0
*/
Salesperson :: Salesperson()
{
  for ( int i = 0; i < 12; i++ )
     sales[ i ] = 0.0;
}

/*-----------------Member function: getSalesFromUser( )------------------*/
/* get 12 sales figures from the user at the keyboard
*/
void Salesperson :: getSalesFromUser()
{
     for ( int i = 1; i <= 12; i++ )
     {
        cout << "Enter sales amount for month " << i << ": ";
        cin >> sales[ i - 1 ];
    }
}

/*---------------------Member function: setSales(   )------------------------*/
/* Set one of the 12 monthly sales figures
*/
void Salesperson :: setSales( int month, double amount )
{
   /*--------- test for valid month and amount values -------------------*/
   if ( month >= 1 && month <= 12 && amount > = 0 )
      sales[ month - 1 ] = amount;                    // adjust for subscripts 0-11
   else                                              // invalid month or amount value
  {
        cout << endl  <<   "Invalid month or sales figure";
        exit(1);
  }
}
```

```
/*----------------------Member function: getSales(    )-------------------------*/
/* Return one of the 12 monthly sales figures
*/
double Salesperson :: getSales( int month )
{
/*------------------------ test for valid month -----------------------------------*/
   if ( month >= 1 && month <= 12 )
      return( sales[ month - 1 ] );                   // adjust for subscripts 0-11
   else                                               // invalid month
   {
        cout << endl  <<   "Invalid month or sales figure";
        exit(1);
   }
}

/*------------------Member function: printAnnualSale(   )------------------*/
/* print the total annual sale
*/
void Salesperson::printAnnualSales(   )
{
      cout.setf(ios :: fixed);
      cout.setf(ios :: showpoint);
       cout   <<    setprecision( 2 )    <<   fixed
              <<    "\nThe total annual sales is: $"
              <<    totalAnnualSales(    )    <<    endl;
}

/*---------------Member function: total AnnualSales(   )------------------*/
/* total annual sales
*/
double Salesperson :: totalAnnualSales(   )
{
     double total = 0.0;

   for ( int i = 0; i < 12; i++ )
     total += sales[ i ];                    // add month i sales to total

   return total;
}
```

```
/*----------------------------using the Salesperson class ------------------------------*/
/* read a salesperson's 12 months sales and compute and print his yearly total sale */

#include "Salesperson.h"

int main()
{
  Salesperson first;          //create a SalesPerson object first

  /*------------------read his 12 month sales ---------------------------*/
  first.getSalesFromUser( );

  /*--------------compute his yearly total sales and print it -----------*/
  first.printAnnualSales( );

  return 0;
}
```

Dynamic Arrays as Class Data Members

- We illustrate the use of a dynamic array as a class data member with the class, *DynamicList*. Its data members are the pointer variable *listPtr* and *length* that holds the number of elements in the array.

```
class   DynamicList
{
    public:
        DynamicList(    );                              // the default length of a list is 10
        DynamicList( int  num );                        // create and initialize a list of num elements
        DynamicList( const double array[  ], int size);
          //  create a list and initialize it with an array with size number of elements
        DynamicList( const DynamicList & listObject );     // copy constructor
        ~DynamicList(   );          // destructor
        const DynamicList  & operator  = ( const  DynamicList & rightListObject );
          // overloaded assignment operator:  use of const return to avoid  left associativity: (A = B) = C
        void  setValue( double newValue, int index );
           // set the new value of the element of the list at position index
        double  getValue( int index );      //  return the value of the element of the list at position index
        double  getLength(   );             //  return the number of elements in the list
        void readValues(   );               // read values into the list
        void  printValues(   );             // output the values of the elements of the list
        double  getAverage(   );            // compute and return the average of the values in the list
    private:
        double   *listPtr;
        int  length;
};
```

```
/*----------------------------------------------DynamicList.cpp---------------------------------------------*/
#include <iostream>
#include <iomanip>
#include  <cstdlib>
using namespace std;

#include "DynamicList.h"

/*-------------------------------------------default constructor-----------------------------------------------*/
/* Default length of the list is 10.  Initialize the elements of the list to 0.0                                    */
DynamicList :: DynamicList(    )  :  length ( 10 )
{
     listPtr  =  new  double [ 10 ];
     for ( int i = 0;  i  <  10  ;  i ++ )
          listPtr [ i ] = 0.0;
}

/*-------------------constructor DynamicList( int  num )-----------------------------------------------*/
/* Initialize the elements of the list to 0.0                                         */
DynamicList :: DynamicList(  int  num  )  :  length ( num )
{
     listPtr  =  new  double [ num ];
     for ( int i = 0;  i  <  length  ;  i ++ )
          listPtr [ i ] = 0.0;
}

/*-------------------constructor DynamicList( const double array[  ], int size)-------------------------*/
/* Initialize the elements of the new list with the elements of the array argument                          */
DynamicList :: DynamicList( const double newlist[  ], int size)  :  length ( size )
{
     listPtr  =  new  double [ size ];
     for ( int i = 0;  i  <  length  ;  i ++ )
          listPtr [ i ] = newlist [ i ];
}

/*---------------copy constructor DynamicList( const DynamicList & listObject ) -----------------*/
/* Initialize the elements of the new list with the elements of listObject                                      */
DynamicList :: DynamicList( const DynamicList & listObject )  :   length ( listObject.length )
{
     listPtr  =  new  double [ length ];
     for ( int i = 0;  i  <  length  ;  i ++ )
          listPtr [ i ] = listObject.listPtr [ i ];
}
```

```
/*-------------------------------destructor ~DynamicList(   )---------------------------------------------*/
DynamicList :: ~DynamicList(   )
{
    delete  [ ] listPtr;
}

/*----------------------overloaded assignment operator---------------------------------------------------*/
/* use of const return to avoid left associativity: (A = B) = C                                          */
/* do not copy a list into itself and if the two list do not have the same number of elements, deallocate
   the original list and create another one with the same number of elements as the list being copied
*/
const DynamicList  & DynamicList :: operator  = ( const  DynamicList & rightListObject )
{
   if ( this  != &rightListObject )      // copy only if the two lists are not the same
  {
        if ( length  != rightListObject.length )
         {
           delete   [  ]  listPtr;       //  deallocate memory locations for the left side list
           length =  rightListObject.length;
           listPtr = new double [ length ];
        }
        for ( int  i = 0; i < length ;  i ++ )
             listPtr [ i ] = rightListObject.listPtr [ i ];
  }
}
```

Exercise A22

1. Write the definitions of the member functions *setValue( )*, *getValue( )*, *getLength(   )*, *readValues(   )*, *printValues(   )*, and *getAverage(   )*.
2. Write a declaration statement to define the following objects of the class DynamicList:
   a. Default list dListObj1.
   b. Object dListObj2 of 50 elements with its values set to 0.0
   c. Object dListObj3 initialized with the array with elements initialized to {1, 2, 3, 4, 5, 6, 7}.
   d. Object dListObj4 initialized with object dListOnj3.
3. Write a statement to read values into object dListObj1.
4. Write a statement to print the values of the elements of object dListObj1.
5. Write a statement to compute the average value of the elements of object dListObj1.
6. Write a statement to output the 5th element of object dListObj3.
7. Write a statement to change the value of the 4th element of object dListObj3 to 50.

Two-Dimensional Arrays

- A **two-dimensional array** allows you to organize and access information as a table.
- **You define a two-dimensional array** as follows:

<data-type>   <array-name> [row][column];

<array-name>	is the name of the array or the table,
<data-type>	is the data type of the elements (variables) in the table,
row	is the number of rows in the table, and
column	the number of columns.

Example A18

In a computer science course class, 10 students have completed 6 lab. assignments and have received a score in the range of 0 to 10 in each of these lab. assignments. Students' scores on these lab. assignments can be represented using a table as follows:

			Lab. Assignment #					
			1	2	3	4	5	6
		Column	0	1	2	3	4	5
		Row						
	1	0	8	9	9	7	10	8
	2	1	9	9	9	10	8	10
	3	2	10	10	10	9	10	10
	4	3	6	7	7	6	9	8
Student #	5	4	8	10	8	8	6	9
	6	5	9	10	8	9	10	10
	7	6	6	5	5	7	6	7
	8	7	9	8	8	7	9	6
	9	8	8	8	9	9	9	10
	10	9	8	9	9	8	8	8

- This table is represented in C++ by the two-dimensional array defined as follows:

int classScores[10][6];

- The score of student number 2 on lab. assignment number 5 is 8, and is represented by the array element *classScores[1][4]*
- The score of student number 6 on lab. assignment number 4 is 9, and is represented by the array element *classScores[5][3]*.

Example A19

A page of a computer document that is formatted to hold a maximum of 50 lines of text with a maximum of 80 characters per line can be represented in C++ by the two-dimensional array, *page[50][80]* defined as follows:

```
char page[50][80];
```

- The first character of the first line is represented by the array element page[0][0].
- The 25th character of the 43rd line is represented by the array element page[42][24].
- Indexed variables of a two-dimensional array are used in the same way that you use indexed variables of a one-dimensional array with the same data type:

1. Statement to store character 'H' in the 8th position of the 16th line: page[15][7] = 'H';
2. Statement to adds 2 to the 4th lab. assignment score of the 8th student: classScores[7][3] += 2;
3. Statement to prints the character in the 47th position of the 20th line, and the 6th lab. assignment score of the 9th student:

```
cout << endl << "the character is:\t" << page[19][46]
     << endl << "and the score is:\t" << classScores[8][5];
```

4. Two index expressions (one for the rows and another one for the columns) may also be used to access the elements of a two-dimensional array:

```
int  i = 5, j = 2;
classScores[i][j]           refers to the same array element as     classScores[5][2]
classScores[i - 1][j + 3]   refers to the same array element as     classScores[4][5].
```

Exercise A23*

Write the declaration statements to define the following two-dimensional arrays:

a. Array to hold 6 test scores (integer values) in a class of 20 students.
b. Array to hold the total sales (double precision values) made by 25 salesmen in 10 cities.
c. Array to represent a page of a computer document that is formatted to hold a maximum of 45 lines of text with a maximum of 65 characters per line.

Initializing the Elements of a Two-Dimensional Array

- A two-dimensional array may be initialized only when it is declared by following its declaration with the equal sign, which is also followed by the list of values.
- Initial values are listed row by row and braces may be used to separate the initial values for one row from those of another.

Example A20

The statement:

```
int M[4][5]   =   { { 10, 5, -3 17, 83 },
                    { 9, 0, 0, 8, -7 },
                    { 32, 20, 1, 0, 14},
                    { 0, 0, 8, 7, 6}};
```

initializes the elements of array *M* as follows:

M[0][0] = 10	M[0][1] = 5	M[0][2] = -3	M[0][3] = 17	M[0][4] = 83
M[1][0] = 9	M[1][1] = 0	M[1][2] = 0	M[1][3] = 8	M[1][4] = -7
M[2][0] = 32	M[2][1] = 20	M[2][2] = 1	M[2][3] = 0	M[2][4] = 14
M[3][0] = 0	M[3][1] = 0	M[3][2] = 8	M[3][3] = 7	M[3][4] = 6.

The above initializations could also be achieved with the following statement in which the braces are omitted:

```
int M[4][5]   =   { 10, 5, -3, 17, 83 ,  9, 0, 0, 8, -7, 32, 20, 1, 0, 14,
                     0, 0, 8, 7, 6};
```

However, the statement:

```
int M[4][5]   =   { { 10, 5, -3 },
                    { 9, 2, 7 },
                    { 32, 20, 1},
                     {0, 0, 8}};
```

only initializes the first three columns of each row to the values specified. The last two columns of each row are set to 0.

Memory Allocation for a Two-Dimensional Array

- A two-dimensional array is a one-dimensional array of one-dimensional arrays that consist of the rows of the array.
- In addition to the one-dimensional arrays that represent the rows of the array, there is another array of pointers such that each element of this array holds the address of the first element of the corresponding one-dimensional array.
- The name of a two-dimensional array is a pointer constant that holds the address of the first element of the one-dimensional array of pointers.

Example

Array **int table[5][3];**
is represented as follows:

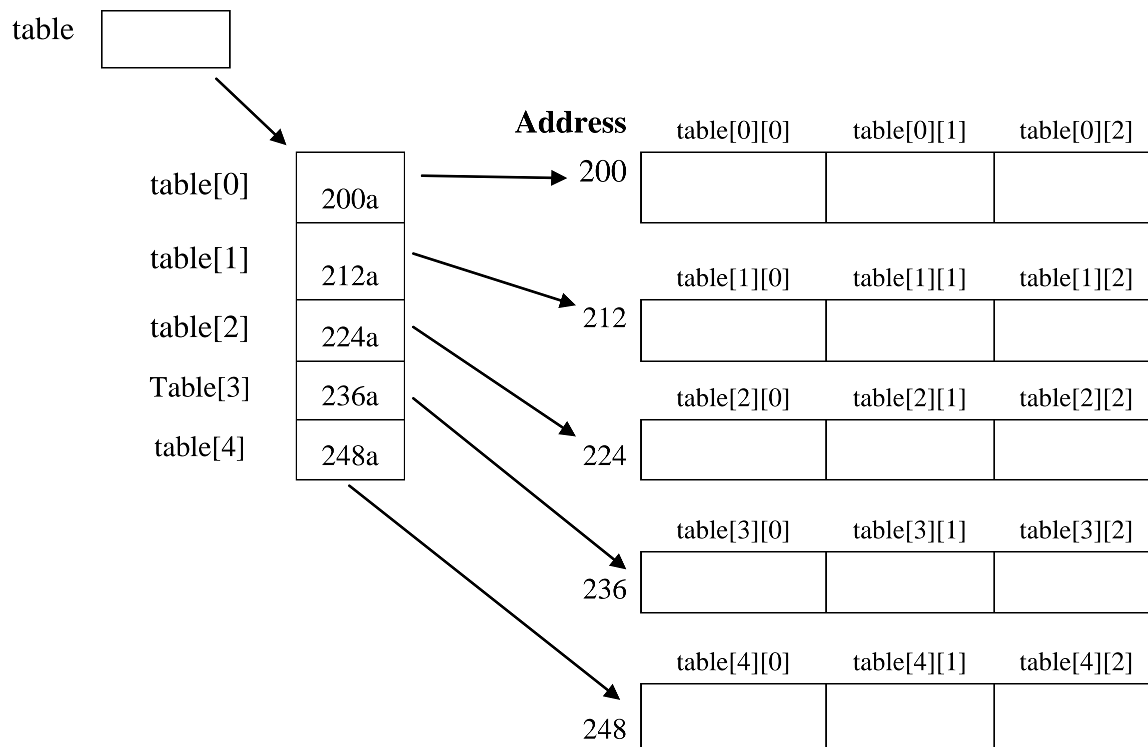


table holds the address of table[0].

table[0] holds the address of table[0][0].

table[1] holds the address of table[1][0].

table[2] holds the address of table[2][0].

table[3] holds the address of table[3][0].

table[4] holds the address of table[4][0].

Processing Two-Dimensional Arrays

- Two-dimensional arrays are either processed row by row or column by column using nested iterations.
- In the following examples, we will use the array of integer values consisting of 15 rows and 10 columns named *table* and the named constants MAXCOLUMN and MAXROW defined as follows:

```
#define   MAXROW    15          /* number of rows in the array */
#define MAXCOLUMN   10          /* number of columns in the array */
int   table [15][10];
```

1. **Reading values into an Array (It is assumed that the values are entered row by row)**.

```
for (int row  = 0 ; row < MAXROW ; row ++)
{
    cout   <<   "\n enter all the values for row # \t"   << (row + 1)   <<   '\t';
    for (int column = 0 ; column < MAXCOLUMN ; column ++)
    cin   >>   table[row][column];
}
```

2. **Printing the Elements of an Array**

   a. **The values are printed one row per line**:

```
for (int row  = 0; row < MAXROW; row ++)
{
    cout   <<   "\n values for row # \t"   <<   (row + 1);
    for (int column = 0 ; column < MAXCOLUMN ; column ++)
    cout    <<    '\t'    <<   table[row][column]';
}
```

   b. **the values are printed one column per line**:

```
for (int column  = 0; column < MAXCOLUMN; column ++)
{
    cout   <<   "\n values for column # \t"   <<   (column + 1);
    for (int row = 0 ; row < MAXROW ; row ++)
        cout    <<    '\t'    <<   table[row][column]';
}
```

Exercise A24*

Given the following definitions of arrays *table1*, *table2*, and *table3*:

```
int table1[3][5]   = {{1, 3, 5, 7, 9}, {0, 2, 4, 6, 8}, {1, 2, 3, 4, 5}},
    table2[3][5], table3[10][10];
```

a. what is the output of the following program segment?

```
for (int r = 0; r < 3 ; r ++)
{
    cout   <<   endl;
    for (int c = 0 ; c < 5 ; c ++)
        cout   <<   '\t'   <<   table1[r][c] + 2;
}
```

b. Write a code segment to print the elements of array *table1* row by row.
c. Write a code segment to print the elements of array table1 column by column.
d. Write a code segment to add 5 to each element of array table1 and to store the result in the corresponding element of array *table2*.
e. Write a code segment to built array *table3* such that the value for element table[i][j] is i + j.

Exercise A25

a. Write a declaration statement to define an array to hold 6 test scores in a class of 20 students.
b. Write a code segment to read students' test scores into this array.
c. Write a code segment to compute the average test score for each test and print it.
d. Write a code segment to compute the average test score for each student and print it.

Passing a Two-Dimensional Array to a Function

- In a function definition, the parameter that corresponds to a two-dimensional array argument is specified as follows:

  <data-type>   <Parameter-name>[ ] [number-of-columns]

Example A21

Examples of function headers of functions that receive a two-dimensional array as arguments:

1. *void readvalues (int table[ ][6], int numRow)*
2. *void computeAverage(int table[ ][6], int numRow)*
3. *void addvalue(int matrix[ ][5], int num, int numRow)*

Notes

- Note that you must specify the number of columns with the parameter.
- The number of rows may also be specified; but it is not required, and is not used by the compiler.
- The number of rows must be passed as an argument.

➢ You pass a two-dimensional array to a function by just passing the address of the first element of that array to it: that means the name of the array.

Example A22

Given the following definitions of arrays *classScores* and *mathTable*:

```
int classScores[10][6];
int mathTable[4][5] = { { 12, 7, 10, 4, 24}, {5, 21, 5, 9, 4},
                        {23, -5, 7, 30, 34}, {14, 12, 11, 4, 8}};
```

The following are valid calls to the functions with the above function headers:

```
1. addvalue ( mathTable, 50, 4);
2. readvalues( classScores, 10);
3. computeAverage (classScores, 10);
```

➢ Any array passed to a function must have a data type compatible with the data type of the corresponding parameter.

➢ Inside the body of a function, <*parameter-name*> is used to access the elements of the array passed to the function in the same way that the name of that array is used to access its elements.

Example A23

Definitions of the functions *readvalues(int table[ ][6], int numRow)*, *computeAverage(int table[ ][6], int numRow)*, and *addvalue(int matrix[ ][5], int num, int numRow)*:

```
/*-------------------------------------function addvalue( )-----------------------------------*/
/*  add a constant value to each element of a two-dimensional array with 5 column*/
void addvalue(int matrix[ ][5], int num, int numRow)
{
    for (int row = 0 ; row < numRow ; row ++)
        for (int column = 0 ; column < 5 ; column ++)
            matrix[row][column] += num;
}
```

```
/*---------------------------------function readvalues( ) ---------------------------------*/
/* read values into a two-dimensional array with 6 columns.  The values are entered row by row. */
void readvalues (int table[ ][6], int numRow)
{
    for (int row  = 0 ; row < numRow ; row ++)
    {
        cout   <<   "\n enter all the values for row # \t"   << (row + 1)   <<   '\t';
        for (int column = 0 ; column < 6 ; column ++)
            cin   >>   table[row][column];
    }
}

/*---------------------------------function computeAverage( ) -----------------------------------------*/
/* compute the average of each row of a two-dimensional array with 6 columns, and print it. */
void computeAverage(int table[ ][6], int numRow)
{
    int total;
    for (int row  = 0 ; row < numRow ; row ++)
    {
        /*-----------compute the sum of the values in this row------------------------*/
        for ( total = 0, int column = 0 ; column < 6 ; column ++)
            total  +=  table[row][column];

        /*----------------compute and print the average --------------------------------*/
        cout   <<   "\n the average for row #  "   << (row + 1)   <<   '\tis:   ';
        cout   <<   ( total / 6);
    }
}
```

Exercise A26*

a. Write the function with function header *void addten(int table[][5], int row)* that receives a two-dimensional array with 5 columns and its number of rows, and adds 10 to each element of that array.
b. Write a statement to add 10 to each element of array *table1* defined in exercise A18 by calling function *addten*.

Exercise A27

a. Write the function with function header *int computeTotal(int table[][5], int row)* that receives a two-dimensional array with 5 columns and its number of rows, and computes the sum of all the elements of that array and returns it.
b. Write the statements to compute the sum of all the elements of array table1 defined in exercise A18 and print it by calling function *computeTotal*.

Exercise A28

a. Write the function with function header *void computeRowSum(int table[][5], int list[], int row)* that receives a two-dimensional array with 5 columns, a one- dimensional array, and the number of rows in the two-dimensional array which is also the number of elements in the one-dimensional array. This function computes the sum of the elements of the array in each row, and stores the result in the corresponding element of the one-dimensional array.

b. Given the definition of array *result*: *int result[3];*
Write a statement to compute the sum of each row of array *table1* defined in exercise A18 and to store the result in the corresponding element of array *result*.

A member of a structure/class can be a multidimensional array.

Example A24

a) The information about the total sales made by 15 salesmen in 10 cities for a company is represented using a structure as follows:

```
struct   SalesInfo
{
     string  name;
     int  idNum;
     double  citySales[15][10];
};
```

b) Write a code segment to read the city sales for each salesman and to compute his total sale. Also compute the total sale for the company.

```
SalesInfo  company;
double companyTotal;          // to hold the company total of all sales
double  salesmanTotal;        // to hold a salesman total of sales in all cities
          citySale;           // to hold a salesman city sale

/*------------------read the company's information --------------------------------------------*/
cin   >>   company.name    >>    company.idNum;
```

```
/*-----------------------------read  all sales for the company ----------------------------------*/
          for (int scount = 0 ;  scount < 15 ; scount ++)

               /*---------------- read the sales for a salesman in all cities----------------------------*/
               for (int ccount = 0 ; ccount <  10 ; ccount ++)
                    cin   >>   company.citySales[ scount ][ccount];
     /*---------------compute and print the company total of sales----------------------------------*/
     for (companyTotal = 0, int scount = 0 ;  scount < 15 ; scount ++)
     {
          /*--------------- compute the total sale for a salesman in all cities------------------------*/
          for (salesmanTotal = 0, int ccount = 0 ; ccount <  10 ; ccount ++)
                    salesmanTotal  +=   company.citySales[ scount ][ccount];
          cout   <<   endl   <<   "salesman #\t"   <<   scount +1
                    <<   "total sale is:\t"   <<   salesmanTotal;
          companyTotal  += salesmanTotal;
     }
     cout   <<  endl   <<   "The company total sale is:\t"   <<   companyTotal;
```