CHAPTER 4

Arrays

Arrays are data structures consisting of related items or collection of items of the same data type. An array is a consecutive group or continuous group of memory locations that contain elements of the same data type and are referenced with the same name. To retrieve or access a particular memory location or to access a specific array element, the array name and the position, commonly referred to as index is specified. In C++, array indexing or element numbering starts from 0. Thus, `a[2]` would be the syntax to refer or retrieve the third element of the array named assuming indexing is from 0.

In general `a[i]`, the argument `i` within the square brackets is referred to as index or subscript of the array. Index should always be an integer or integer expression. The array name is an `lvalue` and can be used on the LHS of an assignment statement. As we have already seen arrays when declared occupy memory space, to be more precise they occupy contiguous memory blocks or locations. For an array declaration statement in C/C++, the compiler requires an array name, data type of elements that shall be stored in the array and the number of elements [cells or locations] required for the array. For example, the following declaration statement `int a[20]` reserves 20 elements for integer array `(a[0]-a[19])`.

4.1 Array Declaration

The programmer can define an array of characters or floats, or doubles by making use of the appropriate keyword [`char` or `float` or `double`] in the declaration statement. Values for arrays can either be accepted from the user via an input statement or initialized using an assignment statement. The two syntaxes for input of array elements for an array `a` of size 5 is as follows:

```
int a[5];
User Input:
int i;
for(i=0;i<5;i++)
scanf("%d",&a[i]);
Assignment Statement:  int a[5]= {0,5,6,7,8};
```

4.1.1 The First Syntax

Since it is more than one value [in fact, 5 values] that is to be read in from the user, we use a `for loop` with the `scanf()` function to accept 5 values from the user. The `%d` indicates that

the types of elements to be read in are all of integer data types and remember that with array indexing from 0, the `for loop` that runs from 0–4 actually iterates five times, i.e. the values of integer types are read in from the user.

4.1.2 The Second Syntax

Array element if known at compilation itself can be initialized with assignment statement and flower brace syntax as shown above. If there are fewer initializers than the elements of the array, then the compiler automatically initializes the remaining or pending to be assigned elements to zero. For example, in the statement `int n[10] = {0,5,6}`, elements 3–9 are automatically initialized to zero.

As another example `int b[5]= 0` initializes the first element to 09 and the remaining four elements are implicitly assigned 0. However, the programmer must explicitly initialize the first element for the remaining elements to be zeroed. However, additional initializers such as in `int a[5]={10,20,30,40,50,60,70}` where there 7 initializers for 5 array elements would cause a syntax error. Another syntactic provision is that the array size need not be specified if the initializer is available at the compile stage itself. That is to say the following statement `int a[]={5,4,3,2,1}`, creates an array of 5 elements and initializes to the value in the list.

Arrays getting initialized at compile time or with assignments contribute to performance improvement, in terms of execution time, than arrays that need to be accepted from the user. A simple program for better understanding of the above concepts is given below. The following program initializes an array of 5 elements with values of 1–5 and computes the 4th power of each array element and stores in the resultant array result, code for which is shown in Figure 4.1.

```
1. #include<stdio.h>
2. #include<math.h>
3. int main ()
4. {
5. int size=5,i;
6. int input[size]={1,2,3,4,5}, result[size];
7. for(i=0;i<size;i++)
8. {
9. result[i]=pow(input[i],4);
10. printf("%d %d \n",input[i],result[i]);
11. }
12. return 0;
13. }
```

Figure 4.1: C code using arrays.

Observe that the size of the array is stored in a variable, so that for further scaling (array sizes varying but same problem on hand), the modification needs to be done only one line while hard coding the size directly in the loop or other statements would require that many statements or lines of code to be modified. This is purely a scalability issue and creates programs that are easier to maintain.

4.1.3 Syntactic Variations

Another syntactic exploit with arrays is to make the size variable a constant (keyword `const` preceding the `int` keyword) with the assumption that array sizes remain non-modifiable once program execution begins. However, programmers need to take care while manipulating arrays that they do not walk off either end of an array. Walking off either end of an array or commonly referred to as an array out of bounds is the situation where the programmer is referring to indexes that are not valid.

For example, the declaration `int a[3]={0,1,2}`, the programmer referring to `a[-1]` or `a[3]` is walking of either ends[end refers to positive or negative scale movement] exceeding the defined limits of the array. Such errors are not caught at compilation stage and hence the programmer should be careful enough not to indulge in such poor programming practices. If not, he is in for serious logical errors. Readers may wonder how array indices can proceed in the negative direction. Yes, it is a real possibility when looping through an array; the loop counter variable should never fall below 0 and should always be less than the number of array elements.

4.2 Sorting an Array of Numbers

In the following example, we shall develop a program to sort an array of 5 numbers with compile time initialization and then sort them either in ascending order. The sorting algorithm we will employ is the naive bubble sort algorithm. It is naive in terms of a higher time complexity associated with it. But then this is the most natural form of sorting and which humans are more comfortable.

Bubble sort is based on the principle of comparing of adjacent pair of numbers and exchange or swap them if they are out of order or not in the ascending or descending order as required. This comparison is repeated for the next pair until the last number in the sequence is reached at which stage one pas is over. For an array of n elements there are `n-1` comparisons within each pass and `n-1` passes on the whole required to sort the input array. For better understanding let us assume we have the following 5 element input array that is to be sorted in ascending order. The various passes are as shown in Table 4.1.

Table 4.1: Bubble sort passes

Input:
5 4 3 2 1
Expected output: 1 2 3 4 5
Pass 1: 4 3 2 1 5
Pass 2: 3 2 1 4 5
Pass 3: 2 1 3 4 5
Pass 4: 1 2 3 4 5

Thus, in all 4 passes and 4 comparisons within each pass are required to sort the initial input array.

4.2.1 The Code

We shall not modularize this code, it is with purpose. At a later stage with discussions of pointers, we shall explore how the same code can be modularized when passing by reference [using pointers] would be required. The code to perform bubble sort is shown in Figure 4.2.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  const int arrsize=5;
5.  int input[arrsize]={5,4,3,2,1},pass,temp;
6.  printf("Bubble Sort \n");
7.  for (pass=0;pass<arrsize-1;pass++)
8.  for(j=0;j<arrsize-1;j++)
9.  if (input [j]>input [j+1])
10. {
11. temp = input [j];
12. input[j]=input[j+1];
13. input[j+1]=temp;
14. }
15. printf("The Sorted Array is \n");
16. for(i=0;i<arrsize-1;i++)
17. printf("%d \t",input[i]);
18. return 0;
19. }
```

Figure 4.2: C code to perform bubble sort.

We adopt an array initialization purely to avoid additional code. Readers who wish to read input from the user they should have appropriate `scanf` within `for loop`. It is in fact lines 7–14 that actually perform the sorting in ascending order. Here the swapping is done if the adjacent elements of the array are not in the required ascending or descending order.

Note: As an interesting coding logic swapping can also be achieved without using a temporary variable (third variable). The following two code snippets shown in Tables 4.2 and 4.3 perform swapping without using a temporary variable. Let us assume that the numbers to be swapped are in variables a and b and their values are 1 and 2.

Table 4.2: Swapping without temporary variables (code1)

First code:	*Effect*
a = a + b;	a = 1 + 2 = 3;
b = a - b;	b = 3 - 2 = 1;
a = a - b;	a = 3 - 1 = 2;

The ^carat symbol stands for Exclusive OR operation. This is also referred to as the logical bitwise XOR operator. To understand the above logic we need the bitwise representations of

Table 4.3: Swapping without temporary variables (code 2)

Second code:
a = a^b;
b = a^b;
a = a^b;
Trace through:
a = 1 = 0001 ; b = 2 = 0010
a = a^b = 0001^0010 = 0011 ;
b = a^b = 0011^0010 = 0001; (b = 1)
a = a^b = 0011^0001 = 0010; (a = 2)

a and b. Let us assume that it is a 4-bit representation [this suffices for our input] that is followed. The effect of the above statements is as shown in Table 4.3.

4.2.2 XOR

The output is high or 1 if and only if one of its input is high [exclusive highness with respect to the input]. This differs from the inclusive OR operation [+] in that the condition to be satisfied for + is that at least one of its input should be high for the output to be high. For easier understanding, the truth tables of both logical operations, exclusive OR and inclusive OR are shown in Tables 4.4 and 4.5.

Table 4.4: Truth table for exclusive OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.5: Truth table for inclusive OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

4.3 String Manipulations

A string is formally defined as a sequence of characters or an array of characters. C++ provides a string class. But it will be explored when we actually get into OOP. Here, in structured programming we shall manipulate strings treating them as an array of characters. Possible string manipulations are finding the length of a string, string reversal, string concatenation, palindrome checking, etc.

In fact, for most string manipulations predefined functions from the standard library are available. But then, in this section we will be more interested in defining these functions. Definitely not to contradict our earlier statement that one should not reinvent the wheel. It is from a learning point of view rather from performance that we define these functions. In the process one gets exposed to the various issues or niceties in structured programming. In fact, end users here are readers and not software users. The following code in Figure 4.3 performs the various string manipulations of concatenation, string comparison and length of string.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int strlength(char []);
void stringcopy(char [],char[]);
void stringconcat(char [], char []);
void strrev(char []);
int equal(char[],char[]);
char gloarr[200];
int main ()
{
int yes,concatchoice,choice,strlen=0,again=1;
char str1[200],str2[200],str[200];
char dest[50];
while (again)
{
system("clear");
printf("\n What Do U want to Do");
printf("\n 1. Determine the String Length \n 2. Copy a String \n
3. Concatenation \n 4. String Reversal \n 5. String Equality Check \n
printf("Enter your choice (1 -6 any number) \n");
scanf ("%d",&choice);
switch(choice)
{
case 1: system("clear");
printf("String Length Determination ...\n");
printf("Enter the String for which the length is to be determiend \n");
scanf ("%s",str);
strlen = strlength(str);
printf("The length of %s is %d",str,strlen);
printf("\n Exiting String Length Determination");
printf("\n Do U want to Continue with Other Operations(1-Yes,0-No)");
scanf("%d",&again);
break;
case 2: system("clear");
printf("Trying to Copy a String \n);
printf("\n Enter the Source String \n");
scanf("%s",str1);
stringcopy(str1,str2);
printf("The Copied String is %s",str2);
break;
```

Figure 4.3: Continued

```
case 3: system("clear");
printf("\n Enter the First String \n");
scanf("%s",str1);
printf("\n Enter the Second String \n");
scanf("%s",str2);
printf("What do you want to do \n");
printf("Concatenate First to Second(Input 1) Or
Second to First (Input 2 \n");
printf("Enter Your Concatenation Choice \n");
scanf("%d",&concatchoice);
if(concatchoice==1)
{
stringconcat(str1,str2);
}
else if(concatchoice==2)
{
stringconcat(str2,str1);
}
else
{
printf("\n Not A Valid Sub Choice");
printf("Returning to Main Menu \n");
}
break;
case 4: system("clear");
printf("String Reversal \n");
printf("Enter The String to be Reversed \n");
scanf("%s",str);
strrev(str);
break;
case 5: printf("\n Equality Checking");
printf("Enter the string 1");
scanf("%s",str1);
printf("\n Enter the string 2");
scanf("%s",str2);
yes = equal(str1,str2);
if(yes==1)
printf("Equal");
else
printf("Not Equal");
break;
default:printf("\n Invalid Option");
printf("\n Quitting.... Pgm Execution");
}
}
return 0;
}
int strlength(char forstr[])
{
int i=0;
while (forstr[i]!= '\0')
```

Figure 4.3: Continued

```
i++;
return i;
}
void stringcopy(char source[],char dest[])
{
int i=0;
while (source[i]!='\0')
{
dest[i] = source[i];
i++;
}
dest[i]='\0';
return;
}
void stringconcat(char string1[],char string2[])
{
int loclength,destindex;
int i=0,j=0;
loclength = strlength(string2);
destindex = loclength+1;
printf("\n Local Length is %d",loclength);
printf("\n Dest Index is %d",destindex);
while (string1[i]!='\0')
{
string2[destindex]=string1[i];
printf("\n Copied Char is %c \n",string2[destindex]);
i++;
destindex++;
}
string2[destindex]='\0';
printf("\n The Concatenated String is \n");
for(j=0;j<destindex;j++)
printf("%c",string2[j]);
return;
}
void strrev(char input[])
{
int i=0,j=0;
int loclen = strlength(input);
printf("Rev Length is %d",loclen);
while (loclen¿=0)
{
gloarr[i]=input[loclen];
i++;
loclen-;
}
gloarr[i+1]='\0';
printf("The Reversed String is \n");
for(j=0;j<i;j++)
printf("%c",gloarr[j]);
```

Figure 4.3: Continued

```
return;
}
int equal(char s1[],char s2[])
{
int i;
int len1 = strlength(s1);
int len2=strlength(s2);
if(len1!=len2)
return 0;
for(i=0;i<len1;i++)
if(s1[i]!=s2[i])
return 0;
return 1;
}
```

Figure 4.3: C code for string manipulations.

Storage classes are applicable with array declarations as well. Local automatic array variables are created and destroyed each time the block in which they exist are entered and exited. Also, automatic arrays if not initialized by the programmer or user get initialized with junk or garbage values. Static arrays are created when the block in which they are declared is entered first. The next time program control enters the same block; allocations or initializations are not performed from scratch. All these concepts which we explored in storage classes (Chapter 3) remain intact with respect to arrays as well. Arrays that will have write (assignment) operation performed on them only once should be declared as constant. Constant variables can be assigned or initialized with values once. Such array variables or identifiers should be suitably declared as constant arrays.

4.4 Passing Arrays to Functions

Arrays can be passed to functions by specifying the array name without any brackets (`[ ]`). An array with the following declaration `int a [3]` is passed to a function (let us assume the function is `fn1`) with the following syntax of `fn1(a,3);` where `3` represents the size of the array passed to `fn1()`. If the array size is not passed as an argument, then there should be a mechanism of making this information known to the called function, because the called function will need this information to process specific number of elements of the array within its definition. One inefficient solution would be to make the array size a global variable, but then this would have its own bottlenecks of being susceptible to accidental changes.

C++ automatically passes arrays to functions using simulated call by reference. The called function can modify the array element values in the caller functions original array. The name of the array refers to the address of the first element of the array (base address), since the called function is aware of the arrays starting address, manipulating the array element values within the called function, manipulates or modifies the actual elements of the array in the calling function (original memory location).

The reasoning behind allowing automatic calls by reference is that if they were allowed to be passed by value, then the entire array (with all its elements) needs to be duplicated. This can be acceptable when the array size is small. But for realistic large arrays, this would

consume or occupy storage space. However, individual array elements if required can be passed by value as is done with simple variables for which the subscripted name of array element is passed as an argument to the called function. The function header for a function that is to receive arrays should explicitly indicate that an array is being passed. For example the function header for function `fn1()` that is to receive an array `a` of size `arr_size` would be as follows:

```
void fn1 (int a [ ], int arr_size);
```

The size of the array if specified between the square brackets is simply ignored by the compiler. The function header for the above function can also be specified excluding the array name or array identifier and the `arr_size` identifier because these serve only as placeholders and it is the data type that is important. For code integrity and functions that need only read permissions over the array, the array and the array size can be passed as constant values. Such arrays remain non-modifiable within the function definition, attempting to do so causes syntax errors. Let us now implement Linear and Binary Search using arrays. Even before we get on with the code, let us briefly review the Linear and Binary Search Algorithms.

4.5 Linear Search

Linear search this is also referred to as sequential search. The entire array starting from the first position is searched for if the element for which the user wants to know the position is present or not. If the element is found, the search is terminated and the index or position is reported. This procedure is repeated in a sequential fashion till the end of the array is reached. If the comparison yields no match when the array end is reached, it means that the element is not finding the list and the same is reported. As is clear from the strategy the search time could be high if the element that is being for is in the last position of the array. The time complexity of linear search is high. [Beginners need not get disturbed]. We shall towards the end of this section establish the time complexities of linear and binary search. Thus, linear search cannot always be the choice.

4.6 Binary Search

The binary search algorithm is efficient in comparison with linear search in terms of time complexity. The binary search algorithm assumes or expects the input array to be sorted in ascending order. The binary search algorithm maintains three indices low, middle and high throughout the strategy. To perform binary search over the entire array, the starting values of low and high are set to 0[first position] and array size 1 [last position]. The middle position or index is computed as follows:

```
middle = (low + high)/2
```

Binary search is based on the principle of reducing the search space [by a factor of 2] after a comparison is done. If the element being searched for matches with the middle element of the array, then the middle position is returned and the algorithm terminates. If the element being searched for is less than the middle element of the array, then it means that the element, which the user is searching for, is lying on the LHS of the middle position. Now, the further

searching operations are performed only on the LHS of the middle position of the array, which means that the searching space should be resized.

The low and high indices which normally denote the first and last position of the array would be chanted to reflect the reduced subarray. The low index that is pointing to the first position should remain the same and the high should be set to position one less than the middle position [high = mid-1] [since the element being searched for is less than middle element of the array]. The last condition is when the element being searched for is greater than the middle element of the array, then it means that the element, which the user is looking for, is lying on the RS of the middle position of the array.

The high index which is pointing to the last position should remain the same and the low index should be one greater than the middle position or low = mid + 1. The resizing of the array and binary search over the resized array is repeated till the element being searched for is found or until the subarray consists of only one element [length1] and the element in subarray is not the search element. [The case when the element being searched for is not found on the array.] Let us assume that we start with an initial array of length or size 10 and the element being searched for is in the last position. The number of comparisons required with linear search will be 10 to locate the element, whereas with binary search it is only 4 comparisons. [Readers are advised to trace through the algorithm.] When it is the case with a small size of n=10, one can imagine the drastic reduction in the search space and hence the number of comparisons required with really large arrays. For better understanding of let us assume the initial input array is as follows:

1 2 3 4 5 6 7 8 9 10

and the element being searched for is 10. [last position 9]. The various iterations/passes of binary search over the array are shown in Figure 4.4. Hence the element 10 is found at the 9th position in the array. That is to say, the binary search algorithm returns 9 as the position of the element being searched.

1. Pass 1: low =0, high = 9, middle = 4, input[middle] = 5; search element (10) is greater than input [middle]. Thus low = 4 + 1 and high = 9;
2. Pass 2:low =5, high = 9, middle = 7, input[middle] = 8; search element (10) is greater than input [middle]. Thus, low = 7 + 1 and high = 9;
3. Pass 3:low =8, high = 9, middle = 8, input[middle] = 9; search element (10) is greater than input [middle]. Thus low = 8 + 1 and high = 9;
4. Pass 4:low =9, high = 9 , middle = 9, input[middle] = 10; search element (10) is equal to input [middle].

Figure 4.4: Binary search algorithm passes over the input array.

4.7 Time Complexity

Algorithms get evaluated based on time and space complexity. Time complexity represents the total time involved in the execution of an algorithm or to be literal a program. Time complexity is normally dependent on the size of the input being processed by the algorithm. But then there is no golden rule that time complexity will always be dependent on the input size. In certain cases such as number multiplication, it is the number of bits used to represent the numbers that determine the time complexity and not the number (magnitude). For our searching problem, it is the array size that has a direct role to play in the final complexity.

Thus, the time complexity of linear and binary search is directly proportional to the number of comparisons made or required to locate the element being searched. Three cases that are most often considered with time complexity analysis are: the best, average, and worst cases. The best case denotes the input sequence or input for which the algorithm executes in the minimal time possible. The worst case as obvious denotes the input for which the algorithm executes in the maximal time possible. The average case denotes the input combination (sequence) for which the time is somewhere near the average of the best and worst cases.

Symbolic notations associated with time complexities are O (Big O), ω(Omega) and θ(Theta). They are used to represent the worst, best and average time complexities respectively. In terms of values they can assume, they are also referred to as upper, lower and average bounds. Most often it is the worst-case complexities that are computed for this is indicatory of the fact that the algorithm will not take any more than the computed complexity or time (for this is the worst possible input sequence). It is in fact an upper bound on the number of comparison required. But then a true algorithm analysis would also compute average and best case complexities as well. The best case denotes that the algorithm will take at least the computed time when the input is the best possible.

Average case as we have already stated is an intermediate input sequence which falls between the best and the worst cases or one can interpret that the average case almost 50% of the input resembles the best case input sequence while the remaining input resembles the worst case input sequence. In terms of bound, best case represents the at least (>=), worst case represents the at most (<=) and average case represents the intermediate (=) time required to execute the algorithm. Best case for searching is when the element being searched is the first array element. Worst case is when the element being searched is the last element in the array. Average case is intermediate case between best and average or the element being searched is after the first and before the last element.

4.7.1 Linear Search

Best case: 1 comparison required.
Worst Case: n comparisons required.
Average case: In terms of a generic input size, this would also be some integer (>1 and < n).

4.7.2 Binary Search

O $(\log_2 n)$: Logarithmic representations are used to represent time complexities of problems for which the problem size gets reduced by a sizeable factor or to be more precise the reduction follows an exponential distribution. Here, 2 represents the factor by which the problem size

gets reduced with each pass. For an array of size $\wedge$ assume K is the number of times 'repeated division' is performed, binary search could be expressed as $\frac{\wedge}{2k} > 1$. (The array is repeatedly divided until array size is 1). Therefore $\wedge > 2^k \Rightarrow \log_2^n > k \Rightarrow K \leq \log_2^n$. Readers are advised to refer to *Introduction to Algorithms by Thomas*, H. Cormen for a detailed treatment on *Algorithm Analysis*. Now let us get into the coding of Linear and Binary Search algorithms, details of which are shown in Figure 4.5.

4.8 Two and Multidimensional Arrays

Arrays in C and C++ can have multiple subscripts. To start with the basic case of array subscript = 2 which represents two-dimensional arrays are used to represent tabular information consisting of m rows and n columns. Two-dimensional arrays are also referred to as double subscripted arrays. The upper bound on the number of arrays subscripts in modern C++ compilers is around 12. In general, any two-dimensional array consisting of m rows and n columns is represented as m*n or m by n array. The following declaration statement declares a two-dimensional array consisting of 5 rows and 4 columns:

```
int a[5][4]
```

Again in multiple subscripted arrays as well array indexing is from 0. Every element in array a is referenced by a generic syntax of `a[i][j]; i & j` being the subscripts that uniquely identify an element in `a`; `i` normally denotes the row and `j` represents the column subscript.

All the elements of the first row have a first or row subscript of 0. Similarly, all the elements of the third column have a second or column subscript of 2. A multiple subscripted or double subscripted array can be initialized in its declaration almost similar to single subscript arrays. For example a two-dimensional array `a[3][2]` could be declared and initialized as follows:

```
int a[3][2] = 1,2,0,5,6,7.
```

The values are grouped by rows in braces. Not enough initializers within a brace or a row initializes the remaining elements to 0. Let us look at each of the following three two-dimensional array declarations. Irrespective of the number of array subscripts the compiler stores these array elements in contiguous memory locations.

With two-dimensional arrays, the compiler proceeds in a row major fashion (one row after another). When multiple subscripted arrays are passed to function, the sizes of all array subscripts excluding the first array subscript is required. For with a two-dimensional array `a[3][5]` the array when passed to a function `fn1()`, the corresponding function header of `fn1` would be as follows:

```
void fn1(int a[][5])
```

The compiler uses these subscripts sizes to determine the memory locations of elements in multiple subscripted arrays. As we have already stated, all array elements are stored contiguously in memory.

In double subscripted arrays, the first row is stored in the first set of contiguous blocks of memory available and then followed up by the second row. These subscript values available in function declaration or header enable the compiler to decide on the position or location of an array element with double subscripted or two-dimensional arrays. Each row is a single subscript or one-dimensional array. To locate an element of a particular row, the number of

```
#include<stdio.h>
int linearserach(const int [ ], int, int);
int binarysearch(const int [ ], int , int , int ,int);
int main()
{
const int arrsize=10,i;
int input[arrsize], search element, position, search choice;
for(i=0;i¡arrsize;i++)
scanf("%d",&input[i]);
printf("Enter the element for which u want ot find the position \n");
scanf ("%d",&search element);
printf("Enter your choice \n 1.Linear Search \n 2. Binary Serach \n 3.Exit
\n');
scanf("%d",&search choice);
switch (search choice)
{
case 1: printf("Linear Search in Progress \n");
position = linearsearch ( input,search element,arr size);
if(position>= 0)
printf("Element %d found at position %d \n",search element,position);
else
printf("Element %d not found in the list",search element);
break;
case 2: printf("Binary Search in progress");
position = binarysearch (input,search element,arr size);
if(position>= 0)
printf("Element %d found at position %d \n",search element,position);
else
printf("Element %d not found in the list",search element);
break;
case 3: printf("Quitting ..\n");
default: printf("Invalid Choice \n");
}
return 0;
}
int linearsearch (const int proarr[], int search key,int arsize) {
int i=0;
for(i=0;i<arsize;i++)
if(proarr[i]==search key)
return i;
return -1;
}
int binarysearch(const int prarr[],int serachk, int low, int high, int size)
{
int mid;
while (low <=high)
{
mid = (low + high )/2;
if(searchk==prarr[mid])
return mid;
else if (searchk< prarr[mid])
high = mid-1;
else
low = mid + q;
}
return -1;
}
```

Figure 4.5: C code to perform linear and binary search.

elements of each row should be known to skip the appropriate number of elements (memory locations) when accessing the particular element.

For example, accessing `a[1][2]` one should skip the first-three memory locations [skipping the first row's 3 elements] to reach the second row after which the third element or column of second to access element `a[1][2]`. In certain situations, two-dimensional or multidimensional arrays might be required to accept values or subject to further processing. The earlier syntax of looping adopted is applicable here as well. Have as many nested for or any other looping construct as you have array subscripts. For example, a two-dimensional array to have values input from the keyboard, the following syntax can be adopted. Let us assume that the array declaration is

```
int a[3][2];
for(i=0;i<3;i++)
for(j=0;j<2;j++)
scanf(''%d'',&a[i][j])
```

The loops execute in an inside out fashion or for an index i the inner loop runs from 0 to 1 [2 columns]. That is, the first row (0) elements consisting of 2 columns are initialized or assigned values that are input via the keyboard. Then the outer loop proceeds to the next i or a syntax matching our row major processing of the array.

4.9 Pointers in C/C++

Pointers are one of the most important and powerful features of C/C++. Pointers are normally used to create dynamic data structures (structures containing data that grow and shrink in size). Examples of such data structures include linked lists, queues, stacks. Pointers are also used to simulate call by reference. Pointer variables contain memory addresses or locations as their values. A variable (non-pointer) contains a value stored or assigned to it, whereas a pointer variable contains the address of another variable that has a particular value.

A variable name directly refers to or directly references a value, whereas a pointer variable indirectly references a value. On this basis, referencing a value via a pointer is also referred to as indirection. As has been the trend to declare variable before their use, pointer variables also have to be declared before they are used. The following declaration

```
int a=8,*b;
```

declares the variable a of integer data type and b to be a pointer to some integer data type.

Pointer declarations are better off to be read from right to left for better understanding. Every variable that should act as a pointer should have a * preceding its name and the * should be repeated with every variable name that is to act as a pointer variable. The * operator does not distribute itself with the variables following it, it is applicable only for the immediate variable, delimiter being the comma or the semicolon. For a better understanding of pointer concept, let us go through the following pictorial notation.

Let us assume the declaration is as follows:

```
int * apointer, b=8:
```

Let us assume that the memory locations assigned to `apointer` and `b` are 4000 and 5000. Thus, pictorially the above statement looks like as shown in Figure 4.6.

As we have already stated that the above declaration statement (pointer declaration) is interpreted right to left, as apointer is a variable that is pointing to some integer data type.

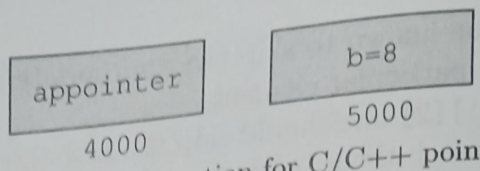

Figure 4.6: Illustration for C/C++ pointers.

But then at some stage or the other, pointer variables will be required to point to specific values. Let us assume in the above case apointer should point to b or apointer should point to a specific integer data type of b, the following assignment statement achieves it:

```
apointer = & b
```

Remember that apointer is a pointer variable and will have as its value the memory address of another variable of the declared data type. In this case apointer, which all along we said will point to some integer data type should point to the specific integer data type b or in other words, apointer should contain the address in memory where b resides. We have already seen that the & operator (ampersand) when applied with variable identifiers or names returns the address of the respective identifier or variable. By now the assignment statement should be clear. To summarize the assignment statement which evaluates right to left assigns or stores the address of b (&b) in the pointer variable apointer. Now that we have seen the syntax to make a pointer point to a specific value, the next requirement would be to refer that indirectly pointed value or what is called as referencing a pointer variable. Getting back to our pictorial representation which would now look like as shown in Figure 4.7.

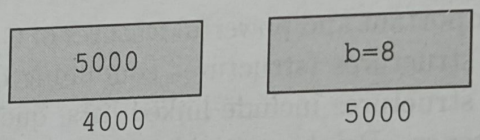

Figure 4.7: Illustration for C/C++ pointers.

To refer to the variables value pointed to by, the * operator is applied before the pointer variable name at places wherever we require the value pointed to. Lets assume we need to sum up the value pointed to by apointer and another non-pointer variable c=7 and store the result in sum; the following statement

```
sum = *apointer + c
```

would achieve the task where *apointer retrieves the value of b=8 pointed to by the apointer. This is referred to as indirection and the * operator is also called the indirection operator or dereferencing operator. The dereferenced or indirected pointer (*apointer) can also be used on the LHS of an assignment statement [the dereferenced pointer is infact an lvalue].

Pointers should be initialized either when they are declared or in an assignment statement. A pointer may be initialized with 0 or keyword NULL [to denote that is a null pointer or it is currently not pointing to a specific value] or with an address. The value 0 is the only integer value that can be assigned directly to apointer variable without casting the integer to the pointer data type. That is, a 0 being assigned to a pointer data type [which can be a char or float or int pointer] is converted to the appropriate pointer type automatically. The keyword NULL is defined in the standard library header files.

Dereferencing a pointer that has not been properly initialized might cause serious run time errors. The dereferencing operator cannot be applied with non-pointer variables and if applied causes syntax error. An interesting syntactic exploitation to be noted with pointer variables is

that the * (dereferencing) and & (address of) operator are inverses of one another, when they are both applied consecutively to a pointer variable in either order, the result is the address stored in or pointed to by the pointer variable. For example, `int *aptr,b=8; aptr = &b;` `*(&aptr)`, and `&(*aptr)` both yield the address of `b` that is stored in the pointer variable `aptr`. How come? The parenthesizing has been done for clarity

`*(&aptr)`: The address of operator returns the address of the `aptr` variable. The dereferencing operator dereferences the value contained at the specific address. We have already explained that a pointer variable contains the address of the variable to which it should point to. The result of `*(&aptr)` returns the address of `b`.

In the second expression of the form `&(*aptr)`, the * operator dereferences the value pointed to by `aptr` or value of `b=8`. The & operator then applied over this is nothing but the address of `b` getting returned. Thus, both expressions are the same.

4.10 Calling Functions by Reference

In the third Chapter on Functions, we have explained two methods of calling functions, one by reference using reference arguments and the other with actual values; or call by value and call by reference (using reference arguments). We shall now look at another mechanism of calling by reference (using pointers). Return statement is used to return a value to the caller function or simply return control back to caller function.

Passing arguments to called functions using reference arguments enabled or provided privilege for the called functions to modify the original values of arguments in the caller functions and thereby avoid the duplicating or copy creation overhead. In this section, we shall use pointer to simulate the reference passing of arguments to functions. Pointers and the indirection operator are used in combination to simulate the call by reference effect. Arrays are not passed using & operator as is done with normal variables since the array name refers to the starting or base address of the array. That is array name is equivalent to `&a[0]`, which implies that the array name is a pointer by itself. The indirection operator can be used in the function called to create an alias and can be used to modify the value (original) in the caller function and thereby simulate the pass by reference to exchange the values of two variables as done by the code shown in Figure 4.8. A function receiving an address as a argument defines a pointer parameter to receive the address [`a,b` in function swap receives the address of `no1` and `no2` declared in `main()`]. This is specified by the function header void `swap (int *, int *);` which denotes that the function swap would receive two arguments that are pointing to some integer data type or receive precisely address as argument. As has been stated previously pointer variable names are not compulsory in function headers as they serve only as place holders. The same syntactic exploitation can be used to pass arrays as well.

Array names, which are pointers to the base, address, such pointers being non-modifiable or constant entities. Thus, the syntax of `int a []` to pass a two-dimensional array automatically is converted by the compiler as `int * const a;` readers are advised to interpret all pointer declarations right to left; that is, `a` is a constant pointer to an integer data type. The address contained in `a` is constant or non-modifiable, i.e. the base address of the array is constant. However, unless explicitly required by the caller function to provide modifications of values define in its scope, within the scope of the called function it is advisable to pass arguments by value, which is the principle of least privilege.

```
1. #include<stdio.h>
2. void swap(int *,int *);
3. int main ()
4. {
5. int no1, no2;
6. printf("Enter the two numbers \n");
7. scanf("%d %d",&no1,&no2);
8. swap(&no1,&no2);
9. printf("Numbers after swapping are %d %d \n",no1,no2)
10. return 0;
11. }
12. void swap(int *a, int *b)
13. {
14. int temp;
15. temp =*a;
16. *a=*b;
17. *b=temp;
18. }
```

Figure 4.8: Swap by reference using pointers.

4.11 Constant Pointers

The constant qualifier (keyword `constant`) when applied with normal or non-pointer variables makes them non-modifiable or constant entities. In fact, no longer can they be called as variables! It is better of to refer them as constant identifiers since a variable by its inherent nature should be modifiable. The constant qualifier can as well be used with pointer variables. In fact, constant qualifiers when applied with normal variables or pointer variables to great extent enforce the concept of least privilege, [the minimum amount of right required is granted] and thereby avoid any unintended or accidental side effects. But then the constant qualifiers can be syntactically exploited to create differing pointer types or types of pointer variables or identifiers in terms of privileges to modify. We shall consider the following four pointer declarations for understanding these pointer types:

```
int * a;
const int * a;
int * const a;
const int * const a;
```

The first declaration `int * a` has been explained already. To briefly review `a` is a pointer to some integer data type. The second declaration `int * const a` declares that `a` is a constant pointer to an integer data type. Again we interpret pointer declaration in right to left fashion for better understanding. A is a constant pointer to some integer data type, means that the content in a (which is the address of another integer variable) is non-modifiable, however, the value pointed to by a which is a constant pointer is modifiable or assigned multiple values at different stages of the program execution. But the pointer (address) remains constant or

non-modifiable. That is with the following declaration:

```
int b =8, c= 5;
int * const a = &b;
*a = 6; // allowed
a = &c ; // not allowed
```

a's content or precisely the address cannot be modified but the value pointed to by a can be changed, or in other words, the address is non-modifiable, but the content is modifiable or can be changed. Constant pointers can be assigned the address of another variable, but only once on the basis that constant identifiers [non-pointer or pointer variables] can be initialized [assigned value] once at the point of declaration or once after the declaration statement. Further, the identifiers cannot be modified. For the third declaration `const int * a`, consider the following example:

```
int b=5,c=6;
a=&b; //allowed
*a=7; //not allowed
```

The statement is interpreted as a being a pointer to an integer constant data type or a points to some integer data type that is a constant. The value which a is pointing to is constant or non-modifiable or the content is constant, while the address is modifiable as shown in the above example. In other words, address of the pointer is modifiable, but the content of the pointer remains constant.

The fourth declaration `const int * const a;` is a constant pointer to a constant integer data type. As is obvious, both the address and content of the pointer variable remains constant. Maximum privilege is provided with the first declaration while minimum privilege is provided with the last declaration. Programmers need to judiciously choose the appropriate declarations for identifiers, arguments [identifiers that are passed to functions].

4.12 Variable Declarations and Their Associated Sizes Occupied

A program can declare variables of integer or character or float or double data type. A variable as per our earlier discussions has a name and a value associated with it, in fact, these are the basic attributes of variables. The memory requirement in terms of storage size is on the data type that is being stored in the variable. An integer data type on modern machines consumes 4 bytes (long) of memory. However, earlier systems had the demarcation between a short and long integers.

Short integers are 2 bytes long and long integers are 4 bytes long. Mostly integer data types are used with a singed interpretation. Thus, an integer array of size 10 would actually consume 40 bytes of memory. The size of operator available with C/C++ can be used to determine the specific system byte consumption for the respective data types. The size of operator when applied with the array names reflects the memory requirement for the entire array, while with pointer variables it reflects the byte consumption of the values pointed to or referred to by the pointer variable. Also, the number of elements in an array can be determined using two size

of operations as follows:

$$\text{Number of elements} = \frac{\text{Size of array}}{\text{Size of array data types}}$$

For example, for the array declaration we had earlier the number of elements = 80/4 = 20 [assuming it is an integer array consuming 80 bytes]. However, the byte consumption for a specific data type may vary across systems. Programs that shall be data type-specific and in turn be system-dependent should explicitly make use of size of operator to check for the system settings. It is just good and efficient coding practice. Size of is an operator and not a function. Its computation is at compile time and there is no overhead resulting from it [in terms of execution time]. In fact, it is included in the keyword list of C/C++. As we have stated an operator `size of` can be applied to any variable name or type name or constant value. The `size of` operator requires parentheses when used with type name [`integer` or `char` or `float`] but does not require parentheses when used with other variable names.

4.13 Pointer Expressions or Arithmetic

Pointer variables as normal non-pointer variables can be used in arithmetic expressions or assignment statements. But then not all operators can be applied with pointers. The number of operators that can be applied with pointers is quite limited. Pointers may be incremented (`++`) or decremented (`-`). Additive arithmetic of pointers with integers as well as with another pointer is allowed. For better understanding let us consider the following example. Pointer arithmetic is entirely machine-dependent as the byte consumption for data types differ across machines, `int a[5]`, (assume location 3000H for `a[0]`), `int *vptr = &a[0]`. In normal arithmetic addition of 3 to 3000 yields 3003, but this will not be the case with the pointer arithmetic.

When an integer is added to or subtracted from a pointer, the pointer gets incremented or decremented by that many times the size of the data type to which the pointer refers. The number of bytes incremented or decremented depends on the data type to which the pointer is actually pointing to. Let us assume that the pointers address us currently 3000. Addition of 1 to it is actually referring to the next valid memory address, (since 0–3 totals to 4 bytes]. But then 3000 which is pointing to an integer would occupy 4 bytes. Thus, 3000–3003 is actually consumed and thus the next available location is 3004 which would be the result of 3000+1, the pointer is incremented to make it point to the next location (not necessarily immediate!).

`vptr= vptr+2` would yield `3000+2*4 = 3008 H`. Thus, `vptr` now points to `v[2]`. In general, the above statement for some generic data type would increment the pointer by twice the number of bytes it takes to store a variable of that generic data type. If a pointer is to be incremented or decremented, then the operators `++`,`--` can be made use of as follows:

`++vptr, vptr++;--vptr and vptr - -`

Each of the above statements either increments or decrements, the pointer points to the next or previous array location. Pointer variables pointing to the same array may be subtracted from one another. For example, if `vptr` contains the location `3000` and `v2ptr` contains the address `3008`, the statement `x = v2ptr-vptr` assigns the number of array elements from `3000` to `3008` which equals to `2`. Pointer arithmetic is meaningful only when performed on

an array. One cannot assume that two variables of the same type are stored contiguously in memory unless they are adjacent array elements.

Pointer arithmetic on pointer variables that does not refer to an array of values is a logical error. Again subtracting or comparing two pointers that does not refer to elements of the same array is also a logical error. As has been stated, running or walking off either ends of an array during pointer arithmetic is also logical error. As far as assignment statements are concerned a pointer can be assigned to another pointer if both pointers are of the same type. If two pointers that are not of the same data type are to be assigned, cast operator can be used to convert the right-hand side pointer (pointer on the right of the = statement) to type of the pointer on the LHS of the = sign so that the two pointers are of the same type and the assignment can be performed.

However, a `void *` (pointer to void) or a generic pointer which can point to any data type can be assigned to all pointer types without any explicit casting. That is `void * = int * / char *` is allowed, whereas `int * /char * = void *` is not allowed. A void pointer (*) cannot be dereferenced since the compiler to dereference a pointer needs to know the number of bytes associated with a data type and with respect to a generic pointer `(void *)` which contains a memory location for an unknown data type and hence the number of bytes to which the pointer refers to is unknown for the compiler. For pointer variables dereferencing to be possible, the compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer.

In the case of a `void *`, the number of bytes cannot be determined from the type. Assigning a pointer of one type to another pointer other than the `void *` is a syntax error and as has been stated dereferencing a `void *` is also a syntax error. We have already stated that pointers can be compared with one another using equality and relational operators, but then such comparison should be made on pointers that point to members of the same array. Such comparisons actually compare the addresses stored in the respective pointers and can be used to determine whether a pointer is 0 or not. Arrays and pointers are closely related with each other and can be used interchangeably. Array names are viewed as constant pointers.

Pointers can be used for array subscripting. Let us assume the following declaration:

```
int a[5],*aptr; aptr
```

can be made to point the first element of the array `a` by `aptr = a`; since the array name without a subscript operator is actually a pointer to the first array element which is equivalent to `aptr =&a[0]`. Array element `a[5]` can be alternatively referenced as `*(aptr+5)`. 5 is actually the displacement or offset to the pointer. A pointer that points to the array beginning, the offset is identical to the array subscript and indicates the element of the array that is to be referenced. This notation is also called pointer or offset notation.

The parentheses are mandatory since * carries higher precedence over +. If the parentheses is not used then the expression is parsed as `*aptr + 5` which would compute the sum of 5 and dereferenced value of `aptr [i.e.a[0]`. Therefore, `*aptr + 5` is actually equal to `a[0] + 5`. The statement `&a[3]` is the same as `bptr+3` [pointer arithmetic] and also `a[3]` is equal to `*(a+3)`. Thus, all array subscriptions can be performed alternatively using a pointer and an offset. Similarly, pointers can also be subscripted as arrays. For example, `aptr[1]` actually refers to the element `a[1]`. This is referred to as pointer or subscript notation. An expression such as `a = a+ 5;` is invalid because array names are not just pointers but constant pointers and thus address modification is not possible. Even though pointer arithmetic-wise it is valid,

thus array names cannot be modified in arithmetic expressions. A programming example to understand array subscripting, pointer/offset with array name as pointer, pointer subscripting and pointer offset with pointer to access the elements of an integer array a is explained using the code in Figure 4.9.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  int a [ ] = {5,10,15,20,25 };
5.  int i, offset;
6.  int * aptr = a; //Array subscript notation
7.  for(i=0;i<5;i++)
8.  printf("%d",a[i]); //Pointer offset notation with array name as pointer
9.  for(offset = 0;offset< 5;offset ++)
10. printf("%d",*(a +offset)); //Pointer subscript notation
11. for(i=0;i<5;i++)
12. printf("%d",aptr[i]); //pointer offset notation with pointer
13. for(offset=0;offset¡5;offset++)
14. printf("%d",*(aptr + offset ));
15. return 0;
16. }
```

Figure 4.9: Array subscription using pointers.

Let us now look at another example regarding pointer–array interchangeable use, that performs string copy operation. The code is as shown in Figure 4.10.

The above example aims at developing the code to perform string copying operation. There are two functions copy1 and copy2 to perform the same operation of copying strings. Copy1 manipulates the strings with array notation while copy2 manipulates strings with the pointer notation. The source strings passed to copy1 and copy2 are made as constant strings since it is only read operation that shall be performed over the strings. The direction of the copy operation in the function call would be in the following order of (destination, source). The driver function creates four arrays of characters. Str2 and Str4 are initialized with the strings "NIT" and "Trichy". Aim of the above program is to copy the contents of str2 to str1 and str3 to str4.

The function copy1 manipulates the strings with array notation. The for loop within the definition iterates thru the string viewing it as an array of characters. Each character of the source string is assigned to the destination string after checking for if it is a '\0' [null or string terminator.] The loop iterates till the end of source string ['\0'] is encountered. Observe that the copying operation has been achieved as a part of the for header structure itself. Thus the definition of the for loop is empty. Similarly the function copy2 copies the strings manipulating them with pointer notation. Again the definition of the for is an empty statement as the copying operation is accomplished in the for header structured itself by the statement *s1=*s2. The loop iterates until the de referenced character from s2 is not a null character. The test driver function main creates two strings NIT and Trichy and two more empty strings. The functions copy1 and copy2 are invoked to test the validity of the copying operation.

```
1. #include<stdio.h>
2. void copy1(char *, const char *);
3. void copy2(char *, const char *);
4. int main()
5. {
6. char str1[10],*str2="NIT";
7. char str3[10],*str4[]="Trichy";
8. copy1(str1,str2);
9. printf("Copied string in str1 is %s",str1);
10. copy2(str3,str4);
11. printf("Copied string is str3 is %s",str3);
12. return 0;
13. }
14. void copy1(char *s1,const char *s2)
15. {
16. int i=0;
17. for(i=0;((s1[i]=s2[i])'='\0');i++)
18. ;
19. }
20. void copy2(char *s1,const char *s2)
21. {
22. for(;((*s1=*s2)'='\0';s1++;s2++)
23. ;
24. }
```

Figure 4.10: String copy using pointers.

4.14 Arrays of Pointers

Arrays can also contain pointers in them. One such example is string array and as we have already seen a string by itself is an array of character. Thus, each string is actually a pointer to the first character. Thus, a string array or an array of string is actually an array of pointers, each pointer referring to the first character of each string. An array of four student names would be declared as follows:

```
char Name_Array[4][50]={"Sudarshan.G", "SriKrishnan", "SaiSiddarth", "Siddarth.J"}
```

`const char * Name_Array[4]` is another equivalent syntax for creating an array of four names. Each name is a pointer to `char` [constant] or each name is by itself an array of characters [array of constant characters]. Although it appears as these names are placed in the array `name_array`, it is only pointers that are actually stored in the `name_array` and each pointer points to the first character of the corresponding string or name. The equivalent syntax of above can as well be used with the first subscript (rows) for the number of strings or names and the second subscript [columns] for the largest string. Both rows and columns being fixed is one disadvantage of the above declaration(1), that is the column size being set to the size of the largest string, considerable memory may be wasted when a large number of names is

stored and each name is shorter than the longest size string declared. Hence, the declaration 2 is a more efficient form of declaring string arrays.

4.15 Function Pointers

In the previous section, we have dealt with pointers to data members. As we declared data pointers, we can also have pointers to functions or what is called as function pointers. Function pointers contain pointers to functions or addresses of functions in memory. As an array name is actually the address in memory of the first array element, a function name is similarly the starting address in memory of the code that performs the function's task. As arrays or pointers were passed to functions, so function pointers can be passed to functions, returned from functions and assigned to other function pointers.

For a better understanding of the function pointers, we shall develop a program to sort an array of numbers either in descending or ascending order. In fact, an earlier code performed the same tasks, but in a redundant fashion. Whether it is ascending or descending order sorting, the sorting algorithm [assuming it is bubble sort] remains the same. Bubble sort is based on the overall principle of comparing adjacent numbers and swapping them if they are not in the right or expected ordering required. Instead of duplicating the bubble sort logic twice [once each for ascending and descending], a solution that implements bubble sort logic only once and takes care of swapping suitably for ascending or descending order would be a better choice than have the bubble sort logic repeated twice just because the swapping condition is different in each case.

In this program, we shall define a function called `bubble_sort` which in addition to the normal arguments of the array to be sorted, array size also accepts a pointer to either of the two functions ascending or descending. Each of these functions return a value 1 (true) if swapping needs to be performed and a 0 (false) if swapping need not be performed. For a better interpretation of the above function declarations, an ascending function returning a value indicates that the adjacent numbers to be compared are not in the required ascending order and further swapping should be done to ensure that they are in the correct ascending or strictly increasing order.

A 1/0 return has been adopted to indicate that 1 represents the TRUE condition of ascending ordering or increasing ordering required for the current pair of numbers being compared and 0 represents a FALSE condition of no ordering required for the current pair of numbers[they are already in the order required]. In other words, the TRUE condition represents the fact that the adjacent numbers or current pair of numbers being compared or checked are not in the required ordering and hence swapping is required. The FALSE condition represents that the numbers are as per the ordering required and hence no swapping is required for the current pair of numbers being processed which s based on bubble sorting logic.

A true or false value indicates whether swapping is required or not for the current ordering desired. Precisely, the task of checking whether the current or adjacent pair of numbers being processed by the bubble sorting logic are already in ascending or descending order or not is taken care of by two functions, namely ascending and descending. If the numbers are not in the required order the functions return a 1 to and then swap the values invoking another helper function called exchange that swaps the values passed to it and this swapping should get reflected in the caller function. Hence it accepts the arguments by reference or pass by

reference is used. To briefly suggest a solution, the various functions and their corresponding declarations are as follows:

```
int ascending(int,int);
int descending(int,int);
void exchange(int *const, int * const);
void bubble (int *,const int, int (*) (int,int));
```

The first-two function headers have been explained previously. The functions accept only two integers because at all stages of the ordering task, the bubble sort logic would process only two, in fact, two immediate or adjacent numbers. The function exchange which does the swapping of values by reference needs to modify only the value or content at an address and not the address where these values reside. That is to say address modifications should be prevented. This is taken care of by declaring the arguments to be of type `int * const` or constant pointers to integers in which case it is only the pointer actual content or value and not the address to which it points to that can be changed. This enforces the principle of least privilege which is so very crucial to good OO software development as we shall see in the later OOP chapters.

The last and as often they say but not the least which is so very true here, the bubble sort function is where the bubble sort sorting logic is implemented. This function accepts the array to be sorted, array size and pointer to either of the two functions ascending or descending. These functions when defined will have separate memory allocations for their definitions and the function name actually refers to the starting address in memory of the above function code. This is now passed as an argument to function `bubble_sort`, which means that the ascending or descending function would get invoked within the definition of function bubble sort to decide on the swapping requirement. In fact, the very purpose of having arguments passed and a formal name for them is to refer them within the function definition.

In this case, it is a pointer to another function that is passed and referring to this pointer within the function definition would actually be invoking the respective function definition whose address is available in the function pointer or to which it is actually pointing to. We have almost gone through the entire program logic with the exception of the syntax of defining or declaring a function pointer. The statement `int (*fname) (int a,int b)` identifies to the compiler that the function fname is actually a pointer to some function which accepts two integer arguments and return s an integer at the point of invocation. Let us know look at the code that performs the above tasks which is shown in Figure 4.11.

In the above code, the declared function pointer gets its address required (address where another function definition resides in memory) only in the statements in lines 14 or 20 when the decision for ascending or descending ordering requirement is made. The function pointer which was only declared as a part of the `bubble_sort` header gets a name identifier for it in the function header of `bubble_sort` function definition [compares the function to which it would refer to, actually compares the two numbers passed]. It is a formal parameter for reference within the function definition and gets related to the function's address to which it should refer or point to based on users ordering choice of ascending or descending. The logic for bubble sort is implemented only once and the comparison task is left to a function that accepts a function pointer which actually does the comparing logic.

The main application of function pointers is in building menu-driven systems. Each option is serviced by a different function. Pointers to each function are stored in array of function

```
#include<stdio.h>
void bubblesort(int [],const int,int (*) (int,int));
int ascending(int,int);
int descending(int,int);
void exchange (int * const, int * const);
int main()
{
const int arraysize=5;
int order required, count, input[arraysize];
printf("\n Ascending(input 1) or Descending Order(input 2) Required \n");
scanf("%d",&order required);
if(order required ==1)
{
bubblesort(input,arraysize,ascending);
printf("The Ascending Ordering of the input array is \n");
}
else
if (ordering required ==2)
{
bubblesort(input,arraysize,descending);
printf("The Descending Ordering of the input array is \n");
}
for(count=0;count<arraysize;count++)
printf("%d \t",input[count]);
return 0;
}
void bubblesort(int pro arr[], const int size, int (*comp) (int,int))
{
for (pass=0;pass<size-1;pass++)
for (int j=0;j<size-1;j++)
if((*comp)(pro arr[j],pro arr[j+1]))
exchange(&pro arr[j],&pro arr[j+1]);
}
int ascending(int a,int b)
{
if (b<a)
return 1;
else
return 0;
}
int descending(int a, int b)
{
if (b>a)
return 1;
else
return 0;
}
void exchange(int *const ptr1, int * const ptr2)
{
int temp;
temp = *ptr1;
ptr1 = * ptr2;
ptr2 = temp;
}
```

Figure 4.11: Bubble sort using function pointers.

pointers. The user's choice is used as a subscript into the array and the pointer in the array is used to call the function. We shall look at an example to understand this application of function pointers, code for which is shown in Figure 4.12. The number of options available for the end user is three.

```
1. #include<stdio.h>
2. void fn1(int);
3. void fn2(int);
4. void fn3(int);
5. int main ()
6. {
7. void (*f[3])(int) = fn1,fn2,fn3;
8. int choice;
9. printf("Enter Choice 1 -fn1, 2- fn2, 3- fn3 \n");
10. scanf("%d",&choice);
11. while (choice >= 0 && choice < 3)
12. {
13. (*f[choice]) (choice);
14. printf("Enter Choice 1 -fn1, 2- fn2 ,3- fn3 \n");
15. scanf("%d",&choice);
16. }
17. return 0;
18. }
19. void fn1(int a)
20. {
21. printf("Function fn1 called with value %d",a);
22. }
23. void fn2(int a)
24. {
25. printf("Function fn2 called with value %d",a);
26. }
27. void fn3(int a)
28. {
29. printf("Function fn3 called with value %d",a);
30. }
```

Figure 4.12: Function pointer's application in menu-driven systems.

The declaration in is read as `f` is an array of 3 function pointers that each take an integer as an argument and return void. The array is initialized with the three functions that are to associate with the user choice. The user's choice is used to subscript into the array of function pointers. The function call is made as follows:

```
(*f[choice])(choice)
```

in which `f[choice]` selects the pointer at location choice in the array. The pointer is dereferenced to call the function and choice is passed as the argument to the function. For the sake of program flow understanding, each function pointer prints its argument values and its function name to indicate that the intended function was called correctly.

4.16 Character and String Processing

Characters are the fundamental blocks of a C or C++ program. Every program is composed of a sequence of characters that when grouped together meaningfully forms a string. A string is a series of characters treated as a single unit. A string can contain letters or digits or special characters. String literals or string constants are written in double quotation as

```
char * str = "constant"
```

Strings in C/C++ which are treated as array of characters end with a null character or a string terminator `['\0']`. It is the string terminator that identifies that the end of the string is reached with the current character array that is being processed. A string may be assigned in a declaration statement to a character array or a variable of type `char *`. The following declarations are allowed:

```
char exarray[] = "Sivaselvan";
const char * exarray="Sivaselvan";
```

The first declaration creates a 10 element character array containing the individual characters and the last character stored is `'\0'`. Array indexes ex-array[0–9] contains the individual characters. The compiler reserves an additional index for the null character or a string terminator `['\0']`. This, is an additional index allocated and does not consume the allocations for the character array ex-array. Thus for all string or character arrays the compiler allocates one index in addition to whatever is required by the programmer and it is the index or location that is used to store the strings termination character. When storing a string of characters in a character array, the array should be large enough to hold the largest string that can be stored. A string that is longer than the character array in which it is to be stored, characters beyond the array end overwrite or overflow data in memory locations following the array.

The string handling library makes available functions for string manipulations such as comparison of strings, copying strings, string searching for specific characters, string tokenization [splitting] and length of the string determination. The list of all the functions is made available in the `string.h` header file that should be included in the program for the string manipulation functions mentioned below to be made use of or called are as shown in Table 4.6. The code for string manipulations is as shown in Figure 4.13.

Table 4.6: String manipulation functions

Function prototype	*Description*
`char * strcpy(char *s1,const char * s2)`	Copies the string s2 into the character array s1. The value of s1 is returned.
`char * strcat(char * s1,const char *s2)`	Appends string s2 to s1. The first character of s2 overwrites the '\0' of s1. Finally, the value of s1 is returned.
`char * strncpy(char *s1,const *s2,int n)`	Copies at most n characters of string s2 into character array s1. Value of s1 is returned.
`char * strncat(char *s1,const *s2,int n)`	Appends at most n characters of string s2 to s1. The first character of s2 overwrites the '\0'' of s1. Finally, the value of s1 is returned.
`int strcmp(const char*s1,const char *s2)`	Compares the string s1 with string s2. The function returns a value of zero, if the string are equal, <0 if string 1 is less than string 2 , and >0 if string1 is greater than string2 respectively.
`int strncmp(const char *s1,const char*s2, int n)`	Similar to the above comparison function, except that it is only the n characters that are compared for equality.
`int strlen(const char *s)`	Excluding the '\0', it returns the number of characters preceding the \0.

Function strcpy copies its second argument a string into the first argument, a character array that should be large enough to hold the string being copied. The '\0' is also copied. Function strncpy is similar to strcpy except that the number of characters to be copied from s2 is also specified in the call. The terminating null character is not copied unless the length of the string or the number of characters to be copied from s2 is at least 1, greater than the length of the string. If the size is less than that of the string length, then the terminating character is not copied. Only if this size is greater than the number of characters to be copied, that many additional '\0's are appended which means that in the earlier case a \0 should be physically inserted in the last position, otherwise, the program gets into serious logical errors.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[]="National",str2[10],str3[15],
str4[]="Institute",str5[]="Inst";
char input[]="This is to test strtok", *sep;
int choice,i=0;
printf("\n String Manipulations...");
printf("\n 1. strcpy (string copy)");
printf("\n 2. strncpy (string copy))";
printf("\n 3. strcat (concatenation)");
printf("\n 4. strncat (concatenation)");
printf("\n 5. strcmp (comparison)");
printf("\n 6. strncmp (comparison");
printf("\n 7. strtok (separation)");
printf("\n Enter Your Choice :");
scanf("%d",&choice);
switch(choice)
{
case 1: strcpy(str2,str1);
printf("\n Copied String is %s",str2);
break;
case 2: strncpy(str3,str1,6);
str3[6]='\0';
printf("\n Copied String is %s",str3);
break;
case 3: strcat(str1,str4);
printf("\n Concatenated String is %s",str1);
break;
case 4: strncat(str1,str4,5);
printf("\n Concatenated String is %s",str1);
break;
case 5: if(strcmp(str4,str5)==0)
printf("\n Strings are equal");
else
printf("\n Strings are not equal");
break;
case 6: if(strncmp(str4,str5,4)==0)
printf("\n Strings are equal");
else
printf("\n Strings are not equal");
break;
case 7: sep=strtok(input," ");
while(sep[i]!='\0')
{
printf("%c",sep[i]);
i++;
}
break;
default:break;
}
return 0;
}
```

Figure 4.13: String manipulations using built-in functions shown in Table 4.6.

Review Questions

1. Justify the need for arrays construct in any programming situation.
2. Create an integer array and initialize it with values of 0,2,4,6 and 8 using a single assignment statement.
3. Repeat the earlier question, this time accepting the values from the keyboard.
4. Develop a C program to perform linear search over an input array. (Search if a user-specified element is present in the input array or not. If element found returns its position, otherwise, display the message element not found.)
5. Develop a C program to sort an array of numbers using selection sort.
6. Develop a C program to sort an array of numbers using insertion sort.
7. Develop a C program to implement binary search over an input array.
8. Develop a C program to generate the mean, median and mode of elements in an array.
9. Develop a C program using pointers to perform string manipulations of string copy, concatenation and comparison.
10. Justify the need for function pointers.
11. Develop a C program using functions to generate the reverse of a string. Use the function to test if an input string is a palindrome or not. A palindrome is a string that reads the same in the forward and the reverse direction.
12. What do you understand by the term time complexity of an algorithm?
13. Derive the time complexity of binary search algorithm (refer external sources as well).
14. Derive the time complexity of insertion sort algorithm (refer Thomas H Cormen, *Introduction to Algorithms*).
15. Differentiate the declarations `int * a` and `const int * a`.
16. Discuss the unique aspect in C++ with respect to passing arguments to functions which is not supported in traditional C.
17. Why is not mandatory to pass arrays as references?
18. Justify the argument behind the & operator not being mandatory when reading in strings.
19. Identify the different programming paradigms and compare them.(refer external sources).
20. Argue the case for object-oriented programming over structured programming.