POST-GRADUATE DIPLOMA IN COMPUTER APPLICATIONS

PAPER : PGDCA-4
PROBLEM SOLVING USING C

SECTION-B

Department of Distance Education
Punjabi University, Patiala

Lesson No. :

2.1 : Arrays
2.2 : Strings
2.3 : Pointers
2.4 : Structures
2.5 : Searching Techniques
2.6 : Sorting Techniques

Lesson No. 2.1 **Author : Dharam Veer Sharma**

Arrays

2.1.1 Introduction
2.1.2 Objectives of the Lesson
2.1.3 Array Basics
2.1.4 Array Initialization
2.1.5 Arrays and functions
2.1.6 Multi-dimensional Arrays
2.1.7 Summary
2.1.8 Short Answer Type Questions
2.1.9 Long Answer Type Questions
2.1.10 Suggested Books

2.1.1 Introduction

An Array is a collection of elements having the same data type. When multiple elements of the same data type are to be used, then we need some such identifier which can store these multiple elements. In the C language array is one such data structure that can store groups of similar data type elements. These elements are stored in contiguous memory area. The array elements are accessed by providing the name of the storage area or the array and a subscript representing the position of the element within array.

2.1.2 Objectives of the Lesson

We shall discuss the declaration, initialization, printing and manipulation of array elements in this lesson. We shall see how arrays are passed to functions. We shall also discuss multi-dimensional array and using pointers with arrays.

2.1.3 Array basics

Array is a collection of same type elements under the same variable identifier referenced by index number.

Now let's start by looking at a single variable used to store a person's age.

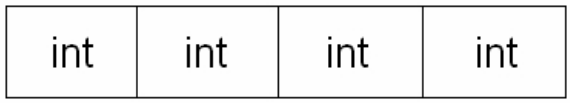
```
1: #include <stdio.h>
2:
3: int main()
4: {
5:   short age;
6:   age=23;
7:   printf("%d\n", age);
```

```
8:   return 0;
9: }
```

Not much to it. The variable age is created at line (5) as a short. A value is assigned to age. Finally, age is printed to the screen.

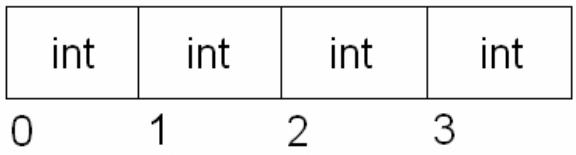
Now let's keep track of 4 ages instead of just one. We could create 4 separate variables, but 4 separate variables have limited appeal. (If using 4 separate variables **is** appealing to you, then consider keeping track of 1000 ages instead of just 4). Rather than using 4 separate variables, we'll use an array which can store a group of similar data type elements as single entity and whose each element is accessed by providing its offset with in the array.

An array is a simple sequence of objects. All of the objects in the sequence are of the same type. The following example presents an array of four integers.



Array of four integers

Each cell of the array is accessed through its index number. Arrays are zero-based. Thus the first cell is defined by index 0, the second by index 1 and so on.



Arrays are declared in the following manner:

```
type variable_name[array_size];
```

The following examples show different ways to declare various arrays.

```
int my_int_array[4]; // An array of 4 integers
double my_double_array[10]; // An array of 10 doubles
```

Accessing an element within an array

Elements within an array can be accessed (for reading or writing) with the subscript operator **[ ]**.

Example -

```
int my_array[10]; // create an array of 10 integers
my_array[3] = 15; // store the number 15 in the 4th element of my-array
cout << my_array[3]; // display the number 15 we just put in
```

NOTE: In C, arrays do not have boundary checking. This means that the programmer is responsible for knowing the number of cells in the array and thus, the last valid index which can be referenced. Accessing an element past the end of the array bounds will not cause a compiler error, but will crash your program at an unpredictable (but usually the worst possible) time. This is a very common bug which is found even in some of the most popular commercial software packages. Be careful – this is one of the most difficult bugs to track down.

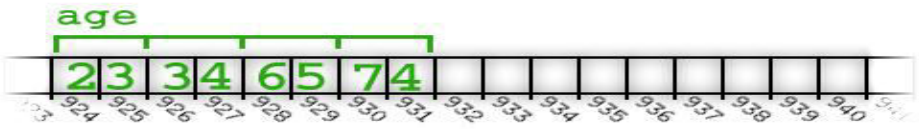
Example -

```
int the_array[2];          //array of 2 integers
the_array[0] = 42;         //valid access
the_array[1] = 1776; //valid access
the_array[3] = 1492; //oops! Error but not reported by compiler.
```

Here's how to create an array and one way to initialize an array:

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:   short age[4];
6:   age[0]=23;
7:   age[1]=34;
8:   age[2]=65;
9:   age[3]=74;
10:   return 0;
11: }
```

On line (5), an array of 4 shorts is created. Values are assigned to each variable in the array on line (6) through line (9).



Accessing any single short variable, or element, in the array is straightforward. Simply provide a number in square braces next to the name of the array. The number identifies which of the 4 elements in the array you want to access.

The program above shows that the first element of an array is accessed with the

number 0 rather than 1. Later We'll discuss why 0 is used to indicate the first element in the array.

2.1.4 Array Initialization

When an array of any type is created, the cells within the array are not empty. Depending on your setup, they will either have an already initialized value depending on the type of array you created or the cells will contain garbage values left over from previous use of that piece of memory. In either case, it is usually a good idea to initialize the elements of the array prior to use. Two common ways of initializing an array is through an initializer list or through a loop. Array elements can be initialized at the time of declaration of the array as following.

Example (initializer list) -

```
int int_arr[] = { 34, 68, 7, 9, 20 };
double double_arr[] = { 22.78, 9.7, 3.1415, 2.71 };
char name[10] = "John";
char name[10]= { 'J', 'o','h','n','\0'};
```

In all the above declaration the array has been initialized there itself. If an array is partially initialized then the remaining elements are automatically initialized to 0 in case of numeric arrays.

```
int a[10] = {0};
```

The above example is a handy way of initializing all elements of an array to 0.

NOTE: When initializer lists are used, the size of the array inside the brackets is not needed. You are free to explicitly place the number there but if you do not, the compiler will know the size based on the number of elements you initialized the array with.

Example (loop) -

```
const int ARRAY_SIZE = 10;
int my_array[ARRAY_SIZE];
for (int i = 0; i < ARRAY_SIZE; i++)
{
        my_array[i] = 0;
}
```

This method is useful when arrays are to be initialized with in the program.

NOTE: A 'for' loop is used by convention to initialize the array. When the value to be used to initialize the array is the same for all the cells, a loop is usually the way to go.

Arrays and Loops

Most of the useful work done with an array requires some sort of searching through the array. While searching through the array, you are accessing every cell in that array. This is done with the same kind of 'for' loop. The following example 'traverses'

(runs through) the entire array in order to check if the number 2 appears anywhere within the array.

Example:

```
const int ARR_SIZE = 7;
int my_array[ARR_SIZE] = { 60, 3, 2, 8, 19, 2, 9 };
for (int i = 0; i < ARR_SIZE; i++)
{
        if (my_array[i] == 2)
        {
                cout << "Number 2 appears in index " << i;
        }
}
```

2.1.5 Arrays and Functions

Arrays, by default, are not passed to functions in the same way as regular variables are. In C, regular variables are passed by value, meaning that any changes you make to those variables in that function will not persist after you leave the function. Arrays are passed by reference (this isn't technically true, but the effect is the same), meaning that any changes you make to array cells in the function are still in effect when the function exits.

NOTE: (For the more inquisitive of you): The name of the array is actually a pointer to the first cell of the array. When you pass an array to a function, what is passed is actually the address of the first cell. Because of this, access to other cells of the array is freely available from inside the function through the use of the subscript operator or through pointer arithmetic.

What follows is a complete program which declares an array, initializes its cells and increments the value in each cell in a separate function.

Example -

```
void increment_all(int an_array[], int size)
{
        for (int i = 0; i < size; i++)
                an_array[i] += 1;
}
int main()
{
        const int ARR_SIZE = 10;
        int my_arr[ARR_SIZE];
        for (int i = 0; i < ARR_SIZE; i++)
                my_arr[i] = i;
        increment_all(my_arr, ARR_SIZE);
```

```
    return 0;
}
```

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all of the members, with the individual members being selected by an index.

Here's an array being declared:

```
double ar[100];
```

The name of the array is ar and its members are accessed as ar[0] through to ar[99] inclusive, as the following figure shows.

100 element array

Each of the hundred members is a separate variable whose type is double. Without exception, all arrays in C are numbered from 0 up to one less than the bound given in the declaration. This is a prime cause of surprise to beginners—watch out for it. For simple examples of the use of arrays, look back at earlier lessons where several problems are solved with their help.

One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. For example, this function incorrectly tries to use its argument in the size of an array declaration:

```
f(int x){
    char var_sized_array[x];        /* FORBIDDEN */
}
```

It's forbidden because the value of x is unknown when the program is compiled; it's a run-time, not a compile-time, value.

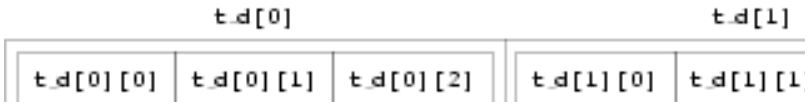
To tell the truth, it would be easy to support arrays whose first dimension is variable, but neither Old C nor the Standard permits it, although we do know of one Very Old C compiler that used to do it.

2.1.6 Multi-dimensional Arrays

Multidimensional arrays can be declared like this:

```
int three_dee[5][4][2];
int t_d[2][3]
```

The use of the brackets gives a clue to what is going on. The first declaration gives us a five-element array called three_dee. The members of that array are each a four element array whose members are an array of two ints. We have declared arrays of arrays, as the following figure shows for two dimensions.



Two-dimensional array, showing layout

In the diagram, you will notice that t_d[0] is one element, immediately followed by t_d[1] (there is no break). It so happens that both of these elements are themselves arrays of three integers. Because of C's storage layout rules, t_d[1][0] is immediately after t_d[0][2]. It would be possible (but very poor practice) to access t_d[1][0] by making use of the lack of array-bound checking in C, and to use the expression t_d[0][3]. That is not recommended—apart from anything else, if the declaration of t_d ever changes, then the results will be likely to surprise you.

That's all very well, but does it really matter in practice? Not much it's true; but it is interesting to note that in terms of actual machine storage layout the rightmost subscript 'varies fastest'. This has an impact when arrays are accessed via pointers. Otherwise, they can be used just as would be expected; expressions like these are quite in order:

```
three_dee[1][3][1] = 0;
three_dee[4][3][1] += 2;
```

The second of those is interesting for two reasons. First, it accesses the very last member of the entire array—although the subscripts were declared to be [5][4][2], the highest usable subscript is always one less than the one used in the declaration. Second, it shows where the combined assignment operators are a real blessing. For the experienced C programmer, it is much easier to tell that only one array member is being accessed, and that it is being incremented by two. Other languages would have to express it like this:

```
three_dee[4][3][1] = three_dee[4][3][1] + 2;
```

It takes a conscious effort to check that the same array member is being referenced on both sides of the assignment. It makes things easier for the compiler too: there is only one array indexing calculation to do and this is likely to result in shorter, faster code. (Of course a clever compiler would notice that the left- and right-hand sides look alike and would be able to generate equally efficient code—but not all compilers are clever and there are lots of special cases where even clever compilers are unable to make use of the information.)

It may be of interest to know that although C offers support for multidimensional arrays, they aren't particularly common to see in practice. One-dimensional arrays are present in most programs, if for no other reason than that's what strings are. Two dimensional arrays are seen occasionally and arrays of higher order than that are most uncommon. One of the reasons is that the array is a rather inflexible data structure and the ease of building and manipulating other types of data structures in

C means that they tend to replace arrays in the more advanced programs. We will see more of this when we look at pointers.

2.1.7 Summary

Arrays are used for storing multiple elements of the same data type. Elements of array can be accessed by providing a subscript enclosed in square brackets. Arrays can be initialized at the time of declaration or using a for loop. If array is partially initialized then the remaining elements are automatically initialized to 0, in case of numeric arrays. Arrays can be passed as arguments to functions. Multi-dimensional arrays are used for storing matrix and other higher dimensional data.

2.1.8 Short Answer Type Questions

1. Define arrays.
2. How array can be initialized at the time of declaration?
3. Define multi-dimensional array.

2.1.9 Long Answer Type Questions

1. What are the methods of arrays initialization? Explain giving examples.
2. How arrays are passed as arguments to functions? Explain.
3. What is the purpose of multi-dimensional arrays? Give an example of these.

2.1.10 Suggested Books

1.	Let Us C	Yashavant Kanetkar
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	The C Programming language	Brian W. Kernigham Dennis M. Ritchie

Web Resources

www.cprogramming.com
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

Lesson No. 2.2 **Author : Dharam Veer Sharma**

Strings

2.2.1 Introduction
2.2.2 Objectives of the Lesson
2.2.3 Representing Strings and String I/O
2.2.4 Defining Strings Within a Program
2.2.5 Inbuilt String Functions
2.2.6 Summary
2.2.7 Short Answer Type Questions
2.2.8 Long Answer Type Questions
2.2.9 Suggested Books

2.2.1 Introduction

The character string is one of the most useful and important data types in C. You have been using character strings all along, but there still is much to learn about them. The C library provides a wide range of functions for reading and writting strings, copying strings, comparing strings, combining strings, searching strings, and more. This lesson will add these capabilities to your programming skills.

2.2.2 Objectives of the Lesson

You will learn about the following in this chapter:

- Functions: gets(), puts(), strcat(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), sprintf(), strchr()
- Creating and using strings
- Using several string and character functions from the C library and creating your own string functions
- Using command-line arguments

2.2.3 Representing Strings and String I/O

Of course, you already know the most basic fact: A character string is a char array terminated with a null character ('\O'). Therefore, what you've learned about arrays and pointers carries over to character strings. But because character strings are so commonly used, C provides many functions specifically designed to work with strings. This lesson discusses the nature of strings, how to declare and initialize strings, how to get them into and out of programs and how to manipulate strings.

The following listing presents a busy program that illustrates several ways to set up, read, and print strings. It uses two new functions—gets(), which reads a string, and puts(), which prints a string. (You probably notice a family resemblance to getchar() and putchar().) The rest of the program should look fairly familiar.

The strings.c Program

```
// strings.c -- stringing the user along
#include <stdio.h>
#define MSG "You must have many talents. Tell me some."
                                // a symbolic string constant
#define LIM  5
#define LINELEN 81              // maximum string length + 1
int main(void)
{
        char name[LINELEN];
        char talents[LINELEN];
        int i;
        // initializing a one dimensioned char array
        const char m1[40] = "Limit yourself to one line's worth.";
                                // letting the compiler compute the
                                // array size
        const char m2[] = "If you can't think of anything, fake it.";
                                // initializing a pointer
        const char *m3 = "\nEnough about me - what's your name?";
                                // initializing an array of
                                // string pointers
        const char *mytal[LIM] = {  // array of 5 pointers
                "Adding numbers swiftly",
                "Multiplying accurately", "Stashing data",
               "Following instructions to the letter",
                "Understanding the C language"
        };
        printf("Hi! I'm Clyde the Computer."
                " I have many talents.\n");
        printf("Let me tell you some of them.\n");
        puts("What were they? Ah, yes, here's a partial list.");
        for (i = 0; i < LIM; i++)
```

```
            puts(mytal[i]);   // print list of computer talents
        puts(m3);
        gets(name);
        printf("Well, %s, %s\n", name, MSG);
        printf("%s\n%s\n", m1, m2);
        gets(talents);
        puts("Let's see if I've got that list:");
        puts(talents);
        printf("Thanks for the information, %s.\n", name);
        return 0;
}
```

To show you what this program does, here is a sample run:

```
Hi! I'm Clyde the Computer. I have many talents.
Let me tell you some of them.
What were they? Ah, yes, here's a partial list.
Adding numbers swiftly
Multiplying accurately
Stashing data
Following instructions to the letter
Understanding the C language
Enough about me – what's your name?
Nigel Barntwit
Well, Nigel Barntwit, You must have many talents. Tell me some.
Just limit yourself to one line's worth.
If you can't think of anything, fake it.
Fencing, yodeling, malingering, cheese tasting, and sighing.
Let's see if I've got that list:
Fencing, yodeling, malingering, cheese tasting, and sighing.
Thanks for the information, Nigel Barntwit.
```

Rather than going through the listing line-by-line, let's take a more encompassing approach. First, you will look at ways of defining a string within a program. Then you will see what is involved in reading a string into a program. Finally, you will study ways to output a string.

2.2.4 Defining Strings Within a Program

As you probably noticed when you read the above listing, there are many ways to define a string. The principal ways are using string constants, using char arrays,

using char pointers and using arrays of character strings. A program should make sure there is a place to store a string, and we will cover that topic, too.

Character String Constants (String Literals)

A string constant, also termed a string literal, is anything enclosed in double quotation marks. The enclosed characters, plus a terminating '\O' character automatically provided by the compiler are stored in memory as a character string. The program uses several such character string constants, most often as arguments for the printf() and puts() functions. Note, too, that you can use #define to define character string constants.

Recall that ANSI C concatenates string literals if they are separated by nothing or by whitespace. For example,

```
char greeting[50] = "Hello, and"" how are"  " you"
                                " today!";
```

is equivalent to this:

```
char greeting[50] = "Hello, and how are you today!";
```

If you want to use a double quotation mark within a string, precede the quotation mark with a backslash, as follows:

```
printf("\"Run, Spot, run!\" exclaimed Dick.\n");
```

This produces the following output:

```
"Run, Spot, run!" exclaimed Dick.
```

Character string constants are placed in the static storage class, which means that if you use a string constant in a function, the string is stored just once and lasts for the duration of the program, even if the function is called several times. The entire quoted phrase acts as a pointer to where the string is stored. This action is analogous to the name of an array acting as a pointer to the array's location. If this is true, what kind of output should the program in the following listing produce?

The quotes.c Program

```
/* quotes.c -- strings as pointers */
#include <stdio.h>
int main(void)
{
        printf("%s, %p, %c\n", "We", "are", *"space farers");
        return 0;
}
```

The %s format should print the string We. The %p format produces an address. So if the phrase "are" is an address, then %p should print the address of the first character in the string. (Pre-ANSI implementations might have to use %u or %lu

instead of %p.) Finally, *"space farers" should produce the value of the address pointed to, which should be the first character of the string "space farers". Does this really happen? Well, here is the output:

We, 0x0040c010, s

Character String Arrays and Initialization

When you define a character string array, you must let the compiler know how much space is needed. One way is to specify an array size large enough to hold the string. The following declaration initializes the array m1 to the characters of the indicated string:

```
const char m1[40] = "Limit yourself to one line's worth.";
```

The const indicates the intent to not alter this string.

This form of initialization is short for the standard array initialization form:

```
const char m1[40] = {  'L', 'i', 'm', 'i', 't', ' ', 'y', 'o', 'u', 'r', 's', 'e', 'l',
                       'f', ' ', 't', 'o', ' ', 'o', 'n', 'e', ' ',
                       'l', 'i', 'n', 'e', '\'', 's', ' ', 'w', 'o', 'r',
                       't', 'h', '.', '\0'
};
```

Note the closing null character. Without it, you have a character array, but not a string.

When you specify the array size, be sure that the number of elements is at least one more (that null character again) than the string length. Any unused elements are automatically initialized to 0 (which in char form is the null character, not the zero digit character).

Often, it is convenient to let the compiler determine the array size; recall that if you omit the size in an initializing declaration, the compiler determines the size for you:

```
const char m2[] = "If you can't think of anything, fake it.";
```

Initializing character arrays is one case when it really does make sense to let the compiler determine the array size. That's because string-processing functions typically don't need to know the size of the array because they can simply look for the null character to mark the end.

Note that the program had to assign a size explicitly for the array name:

```
#define LINELEN 81        // maximum string length + 1
...
char name[LINELEN];
```

Because the contents for name are to be read when the program runs, the compiler has no way of knowing in advance how much space to set aside unless you

tell it. There is no string constant present whose characters the compiler can count, so we gambled that 80 characters would be enough to hold the user's name. When you declare an array, the array size must evaluate to an integer constant. You can't use a variable that gets set at runtime. The array size is locked into the program at compile time. (Actually, with C99 you could use a variable-length array, but you still have no way of knowing in advance how big it has to be.)

```
int n = 8;
char cakes[2 + 5];  /* valid, size is a constant expression
char crumbs[n];     /* invalid prior to C99, a VLA after C99
```

The name of a character array, like any array name, yields the address of the first element of the array. Therefore, the following holds for the array m1:

m1 == &m1[0] , *m1 == 'L', and *(m1+1) == m1[1] == 'i'

Indeed, you can use pointer notation to set up a string. For example, the following declaration uses the pointer:

```
const char *m3 = "\nEnough about me -- what's your name?";
```

This declaration is very nearly the same as this one:

```
char m3[] = "\nEnough about me -- what's your name?"
```

Both declarations amount to saying that m3 is a pointer to the indicated string. In both cases, the quoted string itself determines the amount of storage set aside for the string. Nonetheless, the forms are not identical.

Array Versus Pointer

What is the difference, then, between an array and a pointer form? The array form (m3[]) causes an array of 38 elements (one for each character plus one for the terminating '\0') to be allocated in the computer memory. Each element is initialized to the corresponding character. Typically, what happens is that the quoted string is stored in a data segment that is part of the executable file; when the program is loaded into memory, so is that string. The quoted string is said to be in static memory. But the memory for the array is allocated only after the program begins running. At that time, the quoted string is copied into the array. Hereafter, the compiler will recognize the name m3 as a synonym for the address of the first array element, &m3[0]. One important point here is that in the array form, m3 is an address constant. You can't change m3, because that would mean changing the location (address) where the array is stored. You can use operations such as m3+1 to identify the next element in an array, but ++m3 is not allowed. The increment operator can be used only with the names of variables, not with constants.

The pointer form (*m3) also causes 38 elements in static storage to be set aside for the string. In addition, once the program begins execution, it sets aside one more storage location for the pointer variable m3 and stores the address of the string in the pointer variable. This variable initially points to the first character of the string, but the value can be changed. Therefore, you can use the increment operator. For instance, ++m3 would point to the second character (E).

In short, initializing the array copies a string from static storage to the array, whereas initializing the pointer merely copies the address of the string.

Are these differences important? Often they are not, but it depends on what you try to do. See the following discussion for some examples.

Array and Pointer Differences

Let's examine the differences between initializing a character array to hold a string and initializing a pointer to point to a string. (By "pointing to a string," we really mean pointing to the first character of a string.) For example, consider these two declarations:

```
char heart[] = "I love Tillie!";
char *head = "I love Millie!";
```

The chief difference is that the array name heart is a constant, but the pointer head is a variable. What practical difference does this make?

First, both can use array notation:

```
for (i = 0; i < 6; i++)
        putchar(heart[i]);
putchar('\n');
for (i = 0; i < 6; i++)
        putchar(head[i]));
putchar('\n');
```

This is the output:

```
I love
I love
```

Next, both can use pointer addition:

```
for (i = 0; i < 6; i++)
        putchar(*(heart + i));
putchar('\n');
for (i = 0; i < 6; i++)
        putchar(*(head + i));
putchar('\n');
```

Again, the output is as follows:

```
I love
I love
```

Only the pointer version, however, can use the increment operator:

```
while (*(head) != '\0')  /* stop at end of string          */
putchar(*(head++));  /* print character, advance pointer */
```

This produces the following output:

```
I love Millie!
```

Suppose you want head to agree with heart. You can say

```
head = heart;  /* head now points to the array heart */
```

This makes the head pointer point to the first element of the heart array.

However, you cannot say

```
heart = head;  /* illegal construction */
```

The situation is analogous to x = 3; versus 3 = x;. The left side of the assignment statement must be a variable or more generally, an lvalue, such as *p_int. Incidentally, head = heart; does not make the Millie string vanish; it just changes the address stored in head. Unless you've saved the address of "I love Millie!" elsewhere, however, you won't be able to access that string when head points to another location.

There is a way to alter the heart message—go to the individual array elements:

```
heart[7]= 'M';  or  *(heart + 7) = 'M';
```

The elements of an array are variables (unless the array was declared as const), but the name is not a variable.

Let's go back to a pointer initialization:

```
char * word = "frame";
```

Can you use the pointer to change this string?

```
word[1] = 'l';  // allowed??
```

Your compiler probably will allow this, but, under the current C standard, the behavior for such an action is undefined. Such a statement could, for example, lead to memory access errors. The reason is that a compiler can choose to represent all identical string literals with a single copy in memory. For example, the following statements could all refer to a single memory location of string "Klingon":

```
char * p1 = "Klingon";
p1[0] = 'F';    // ok?
printf("Klingon");
```

```
printf(": Beware the %ss!\n", "Klingon");
```

That is, the compiler can replace each instance of "Klingon" with the same address. If the compiler uses this single-copy representation and allows changing p1[0] to 'F', that would affect all uses of the string, so statements printing the string literal "Klingon" would actually display "Flingon":

```
Flingon: Beware the Flingons!
```

In fact, several compilers do behave this rather confusing way, whereas others produce programs that abort. Therefore, the recommended practice for initializing a pointer to a string literal is to use the const modifier:

```
const char * pl = "Klingon";  // recommended usage
```

Initializing a non-const array with a string literal, however, poses no such problems, because the array gets a copy of the original string.

2.2.5 Inbuilt String Functions

The C library supplies several string-handling functions; ANSI C uses the string.h header file to provide the prototypes. We'll look at some of the most useful and common ones: strlen(), strcat(), strncat(), strcmp(), strncmp(), strcpy() and strncpy(). We'll also examine sprintf(), supported by the stdio.h header file. For a complete list of the string.h family of functions.

The strlen() Function

The strlen() function, as you already know, finds the length of a string. It's used in the next example, a function that shortens lengthy strings:

```
/* fit.c — truncation function */
void fit(char * string, unsigned int size)
{
        if (strlen(string) > size)
                *(string + size) = '\0';
}
```

This function does change the string, so the function header doesn't use const in declaring the formal parameter string.

The ANSI string.h file contains function prototypes for the C family of string functions, which is why this example includes it.

The strcat() Function

The strcat() (for string concatenation) function takes two strings for arguments. A copy of the second string is tacked onto the end of the first and this combined version becomes the new first string. The second string is not altered. Function strcat() is type char * (that is, a pointer-to-char). It returns the value of its first

argument—the address of the first character of the string to which the second string is appended.

```
/* str_cat.c -- joins two strings */
#include <stdio.h>
#include <string.h>  /* declares the strcat() function */
#define SIZE 80
int main(void)
{
        char flower[SIZE];
        char addon[] = "s smell like old shoes.";
        puts("What is your favorite flower?");
        gets(flower);
        strcat(flower, addon);
        puts(flower);
        puts(addon);
        return 0;
}
```

This is the output:

```
What is your favorite flower?
Rose
Roses smell like old shoes.
s smell like old shoes.
```

The strncat() Function

The strcat() function does not check to see whether the second string will fit in the first array. If you fail to allocate enough space for the first array, you will run into problems as excess characters overflow into adjacent memory locations. Of course, you can use strlen() to look before you leap. Note that it adds 1 to the combined lengths to allow space for the null character. Alternatively, you can use strncat(), which takes a second argument indicating the maximum number of characters to add. For example, strncat(bugs, addon, 13) will add the contents of the addon string to bugs, stopping when it reaches 13 additional characters or the null character, whichever comes first. Therefore, counting the null character (which is appended in either case), the bugs array should be large enough to hold the original string (not counting the null character), a maximum of 13 additional characters and the terminal null character. The following listing uses this information to calculate a

value for the available variable, which is used as the maximum number of additional characters allowed.

The join_chk.c Program

```
/* join_chk.c -- joins two strings, check size first */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13
int main(void)
{
        char flower[SIZE];
        char addon[] = "s smell like old shoes.";
        char bug[BUGSIZE];
        int available;
        puts("What is your favorite flower?");
        gets(flower);
        if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
                strcat(flower, addon);
        puts(flower);
        puts("What is your favorite bug?");
        gets(bug);
        available = BUGSIZE - strlen(bug) - 1;
        strncat(bug, addon, available);
        puts(bug);
        return 0;
}
```

Here is a sample run:

```
What is your favorite flower?
Rose
Roses smell like old shoes.
What is your favorite bug?
Aphid
Aphids smell
```

The strcmp() Function

Suppose you want to compare someone's response to a stored string, as shown in the following listing.

The nogo.c Program

```
/* nogo.c -- will this work? */
#include <stdio.h>
#define ANSWER "Grant"
int main(void)
{
        char try[40];
        puts("Who is buried in Grant's tomb?");
        gets(try);
        while (try != ANSWER)
        {
                puts("No, that's wrong. Try again.");
                gets(try);
        }
        puts("That's right!");
        return 0;
}
```

As nice as this program might look, it will not work correctly. ANSWER and try really are pointers, so the comparison try != ANSWER doesn't check to see whether the two strings are the same. Rather, it checks to see whether the two strings have the same address. Because ANSWER and try are stored in different locations, the two addresses are never the same and the user is forever told that he or she is wrong. Such programs tend to discourage people.

What you need is a function that compares string contents, not string addresses. You could devise one, but the job has been done for you with strcmp() (for string comparison). This function does for strings what relational operators do for numbers. In particular, it returns 0 if its two string arguments are the same. The revised program is shown in the following listing.

The compare.c Program

```
/* compare.c -- this will work */
#include <stdio.h>
#include <string.h>   /* declares strcmp() */
#define ANSWER "Grant"
```

```
#define MAX 40
int main(void)
{
        char try[MAX];
        puts("Who is buried in Grant's tomb?");
        gets(try);
        while (strcmp(try,ANSWER) != 0)
        {
                puts("No, that's wrong. Try again.");
                gets(try);
        }
        puts("That's right!");
        return 0;
}
```

One of the nice features of strcmp() is that it compares strings, not arrays. Although the array try occupies 40 memory cells and "Grant" only six (one for the null character), the comparison looks only at the part of try up to its first null character. Therefore, strcmp() can be used to compare strings stored in arrays of different sizes.

What if the user answers "GRANT" or "grant" or "Ulysses S. Grant"? The user is told that he or she is wrong. To make a friendlier program, you have to anticipate all possible correct answers. There are some tricks you can use. For example, you can use #define to define the answer as "GRANT" and write a function that converts all input to uppercase. That eliminates the problem of capitalization, but you still have the other forms to worry about. We leave that as an exercise for you.

The strcmp() Return Value

What value does strcmp() return if the strings are not the same? The following listing shows an example.

The compback.c Program

```
/* compback.c -- strcmp returns */
#include <stdio.h>
#include <string.h>
int main(void)
{
        printf("strcmp(\"A\", \"A\") is ");
        printf("%d\n", strcmp("A", "A"));
        printf("strcmp(\"A\", \"B\") is ");
```

```
    printf("%d\n", strcmp("A", "B"));
    printf("strcmp(\"B\", \"A\") is ");
    printf("%d\n", strcmp("B", "A"));
    printf("strcmp(\"C\", \"A\") is ");
    printf("%d\n", strcmp("C", "A"));
    printf("strcmp(\"Z\", \"a\") is ");
    printf("%d\n", strcmp("Z", "a"));
    printf("strcmp(\"apples\", \"apple\") is ");
    printf("%d\n", strcmp("apples", "apple"));
    return 0;
}
```

Here is the output on one system:

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 1
strcmp("Z", "a") is -1
strcmp("apples", "apple") is 1
```

Comparing "A" to itself returns 0. Comparing "A" to "B" returns -1, and reversing the comparison returns 1. These results suggest that strcmp() returns a negative number if the first string precedes the second alphabetically and that it returns a positive number if the order is the other way. Therefore, comparing "C" to "A" gives a 1. Other systems might return 2—the difference in ASCII code values. The ANSI standard says that strcmp() returns a negative number if the first string comes before the second alphabetically, returns 0 if they are the same, and returns a positive number if the first string follows the second alphabetically. The exact numerical values, however, are left open to the implementation. Here, for example, is the output for another implementation, one that returns the difference between the character codes:

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 2
strcmp("Z", "a") is -7
strcmp("apples", "apple") is 115
```

What if the initial characters are identical? In general, strcmp() moves along until it finds the first pair of disagreeing characters. It then returns the corresponding code. For

instance, in the very last example, "apples" and "apple" agree until the final s of the first string. This matches up with the sixth character in "apple", which is the null character, ASCII 0. Because the null character is the very first character in the ASCII sequence, s comes after it, and the function returns a positive value.

The last comparison points out that strcmp() compares all characters, not just letters, so instead of saying the comparison is alphabetic, we should say that strcmp() goes by the machine collating sequence. That means characters are compared according to their numeric representation, typically the ASCII values. In ASCII, the codes for uppercase letters precede those for lowercase letters. Therefore, strcmp("Z", "a") is negative.

Most often, you won't care about the exact value returned. You just want to know if it is zero or nonzero—that is, whether there is a match or not—or you might be trying to sort the strings alphabetically, in which case you want to know if the comparison is positive, negative or zero.

Incidentally, sometimes it is more convenient to terminate input by entering an empty line—that is, by pressing the Enter key or Return key without entering anything else. To do so, you can modify the while loop control statement so that it looks like this:

The strncmp() Variation

The strcmp() function compares strings until it finds corresponding characters that differ, which could take the search to the end of one of the strings. The strncmp() function compares the strings until they differ or until it has compared a number of characters specified by a third argument. For example, if you wanted to search for strings that begin with "astro", you could limit the search to the first five characters. The following listing shows how.

The starsrch.c Program

```
/* starsrch.c -- use strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 5
int main()
{
        const char * list[LISTSIZE] =
        {
                "astronomy", "astounding",
                "astrophysics", "ostracize",
                "asterism"
```

```
        };
        int count = 0;
        int i;
        for (i = 0; i < LISTSIZE; i++)
                if (strncmp(list[i],"astro", 5) == 0)
                {
                        printf("Found: %s\n", list[i]);
                        count++;
                }
        printf("The list contained %d words beginning with astro\n",
        count);
        return 0;
}
```

Here is the output:

```
Found: astronomy
Found: astrophysics
The list contained 2 words beginning with astro.
```

The strcpy() and strncpy() Functions

We've said that if pts1 and pts2 are both pointers to strings, the expression

```
pts2 = pts1;
```

copies only the address of a string, not the string itself. Suppose, though, that you do want to copy a string. Then you can use the strcpy() function. The following listing asks the user to enter words beginning with q. The program copies the input into a temporary array and if the first letter is a q, the program uses strcpy() to copy the string from the temporary array to a permanent destination. The strcpy() function is the string equivalent of the assignment operator.

The copy1.c Program

```
/* copy1.c -- strcpy() demo */
#include <stdio.h>
#include <string.h>  /* declares strcpy() */
#define SIZE 40
#define LIM 5
int main(void)
{
        char qwords[LIM][SIZE];
        char temp[SIZE];
```

```
        int i = 0;
        printf("Enter %d words beginning with q:\n", LIM);
        while (i < LIM && gets(temp))
        {
                if (temp[0] != 'q')
                        printf("%s doesn't begin with q!\n", temp);
                else
                {
                        strcpy(qwords[i], temp);
                        i++;
                }
        }
        puts("Here are the words accepted:");
        for (i = 0; i < LIM; i++)
                puts(qwords[i]);
        return 0;
}
```

Here is a sample run:

```
Enter 5 words beginning with q:
quackery
quasar
quilt
quotient
no more
no more doesn't begin with q!
quiz
Here are the words accepted:
quackery
quasar
quilt
quotient
quiz
```

Note that the counter i is incremented only when the word entered passes the q test. Also note that the program uses a character-based test:

```
if (temp[0] != 'q')
```

That is, is the first character in the temp array not a q? Another possibility is using a string-based test:

```
if (strncmp(temp, "q", 1) != 0)
```

That is, are the strings temp and "q" different from each other in the first element? Note that the string pointed to by the second argument (temp) is copied into the array pointed to by the first argument (qword[i]). The copy is called the target and the original string is called the source. You can remember the order of the arguments by noting that it is the same as the order in an assignment statement (the target string is on the left):

```
char target[20];
int x;
x = 50;                    /* assignment for numbers */
strcpy(target, "Hi ho!");  /* assignment for strings */
target = "So long";        /* syntax error           */
```

It is your responsibility to make sure the destination array has enough room to copy the source. The following is asking for trouble:

```
char * str;
strcpy(str, "The C of Tranquility"); /* a problem */
```

The function will copy the string "The C of Tranquility" to the address specified by str, but str is uninitialized, so the copy might wind up anywhere!

In short, strcpy() takes two string pointers as arguments. The second pointer, which points to the original string, can be a declared pointer, an array name or a string constant. The first pointer, which points to the copy, should point to a data object, such as an array, roomy enough to hold the string. Remember, declaring an array allocates storage space for data; declaring a pointer only allocates storage space for one address.

The sprintf() Function

The sprintf() function is declared in stdio.h instead of string.h. It works like printf(), but it writes to a string instead of writing to a display. Therefore, it provides a way to combine several elements into a single string. The first argument to sprintf() is the address of the target string. The remaining arguments are the same as for printf()—a conversion specification string followed by a list of items to be written.

The following listing uses sprintf() to combine three items (two strings and a number) into a single string. Note that it uses sprintf() the same way you would use printf(), except that the resulting string is stored in the array formal instead of being displayed onscreen.

The format.c Program

```
/* format.c -- format a string */
```

```
#include <stdio.h>
#define MAX 20
int main(void)
{
        char first[MAX];
        char last[MAX];
        char formal[2 * MAX + 10];
        double prize;
        puts("Enter your first name:");
        gets(first);
        puts("Enter your last name:");
        gets(last);
        puts("Enter your prize money:");
        scanf("%lf", &prize);
        sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
        puts(formal);
        return 0;
}
```

Here's a sample run:

```
Enter your first name:
Teddy
Enter your last name:
Behr
Enter your prize money:
2000
Behr, Teddy             : $2000.00
```

The sprintf() command took the input and formatted it into a standard form, which it then stored in the string formal.

Other String Functions

The ANSI C library has more than 20 string-handling functions and the following list summarizes some of the more commonly used ones:

- char *strcpy(char * s1, const char * s2);
  This function copies the string (including the null character) pointed to by s2 to the location pointed to by s1. The return value is s1.
- char *strncpy(char * s1, const char * s2, size_t n);

This function copies to the location pointed to by s1 no more than n characters from the string pointed to by s2. The return value is s1. No characters after a null character are copied and, if the source string is shorter than n characters, the target string is padded with null characters. If the source string has n or more characters, no null character is copied. The return value is s1.

- char *strcat(char * s1, const char * s2);
  The string pointed to by s2 is copied to the end of the string pointed to by s1. The first character of the s2 string is copied over the null character of the s1 string. The return value is s1.
- char *strncat(char * s1, const char * s2, size_t n);
  No more than the first n characters of the s2 string are appended to the s1 string, with the first character of the s2 string being copied over the null character of the s1 string. The null character and any characters following it in the s2 string are not copied, and a null character is appended to the result. The return value is s1.
- int strcmp(const char * s1, const char * s2);
  This function returns a positive value if the s1 string follows the s2 string in the machine collating sequence, the value 0 if the two strings are identical, and a negative value if the first string precedes the second string in the machine collating sequence.
- int strncmp(const char * s1, const char * s2, size_t n);
  This function works like strcmp(), except that the comparison stops after n characters or when the first null character is encountered, whichever comes first.
- char *strchr(const char * s, int c);
  This function returns a pointer to the first location in the string s that holds the character c. (The terminating null character is part of the string, so it can be searched for.) The function returns the null pointer if the character is not found.
- char *strpbrk(const char * s1, const char * s2);
  This function returns a pointer to the first location in the string s1 that holds any character found in the s2 string. The function returns the null pointer if no character is found.
- char *strrchr(const char * s, int c);
  This function returns a pointer to the last occurrence of the character c in the string s. (The terminating null character is part of the string, so it can be searched for.) The function returns the null pointer if the character is not found.
- char *strstr(const char * s1, const char * s2);
  This function returns a pointer to the first occurrence of string s2 in string s1. The function returns the null pointer if the string is not found.

- size_t strlen(const char * s);

  This function returns the number of characters, not including the terminating null character, found in the string s.

Note that these prototypes use the keyword const to indicate which strings are not altered by a function. For example, consider the following:

char *strcpy(char * s1, const char * s2);

It means s2 points to a string that can't be changed, at least not by the strcpy() function, but s1 points to a string that can be changed. This makes sense, because s1 is the target string, which gets altered and s2 is the source string, which should be left unchanged.

2.2.6 Summary

The C language does not support string data type. String must be terminated by appending a null character. But string can be simulated either using character array or pointer to character. The C language has a rich set of string functions, which operate on both representations of the string.

2.2.7 Short Answer Type Questions

1. What is the difference between character array and character pointer?
2. What is the purpose of strlen() function?
3. What is the purpose of strcat() function?

2.2.8 Long Answer Type Questions

1. Define and distinguish between character array and character pointer modes of string representation.
2. What are the various ways of storing strings?
3. What are the different ways of copying string?
4. What are the different ways of comparing strings?

2.2.9 Suggested Books

1.	Let Us C	Yashavant Kanetkar
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	The C Programming language	Brian W. Kernigham Dennis M. Ritchie

Web Resources

www.cprogramming.com
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

Lesson No. 2.3 **Author : Dharam Veer Sharma**

Pointers

2.3.1 Introduction
2.3.2 Objectives of the Lesson
2.3.3 Pointer and Addresses
2.3.4 Pointer and Function Arguments
2.3.5 Pointers and Arrays
2.3.6 Address Arithmetic
2.3.7 Character Pointers and Functions
2.3.8 Pointer Arrays; Pointers to Pointers
2.3.9 Pointes to Functions
2.3.10 Summary
2.3.11 Short Answer Type Questions
2.3.12 Long Answer Type Questions
2.3.13 Suggested Books

2.3.1 Introduction

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the goto statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type void * (pointer to void) replaces char * as the proper type for a generic pointer.

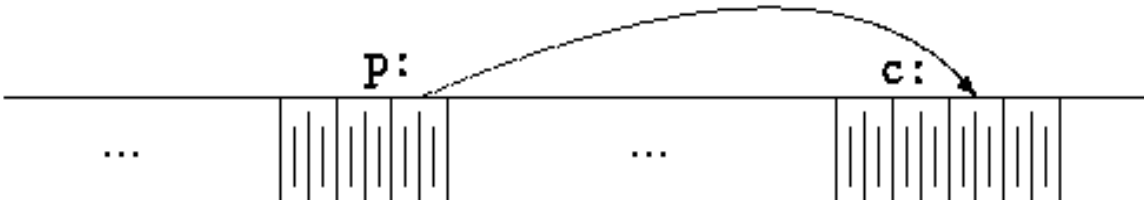
2.3.2 Objectives of the Lesson

Pointers play important role in efficient programming but are very dangerous if proper care is not taken while using them. Through pointers we can directly access the memory. In this lesson we shall discuss the concept of pointers and how to use pointers, including pointer arithmetic, passing pointers as function arguments,

relation between arrays and pointers.

2.3.3 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long. A pointer is a group of cells (often two or four) that can hold an address. So if c is a char and p is a pointer that points to it, we could represent the situation this way:



The unary operator & gives the address of an object, so the statement p = &c; assigns the address of c to the variable p and p is said to ``point to'' c. The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants or register variables. So, we can say that a **pointer** is a variable that points to (or contains) address of another variable.

The unary operator * is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip;/* ip is a pointer to int */
ip = &x;        /* ip now points to x */
y = *ip;        /* y is now 1 */
*ip = 0;        /* x is now 0 */
ip = &z[0];     /* ip now points to z[0] */
```

The declaration of x, y, and z are what we've seen all along. The declaration of the pointer ip,

```
int *ip;
```

is intended as a mnemonic; it says that the expression *ip is an int. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression *dp and atof(s) have values of double and that the argument of atof is a pointer to char.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a ``pointer to void'' is used to hold any type of pointer but cannot be dereferenced itself.)

If ip points to the integer x, then *ip can occur in any context where x could, so

```
*ip = *ip + 10;
```

increments *ip by 10.

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1 and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if iq is another pointer to int,

```
iq = ip
```

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

2.3.4 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called swap. It is not enough to write

```
swap(a, b);
```

where the swap function is defined as

```
void swap(int x, int y)  /* WRONG */
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

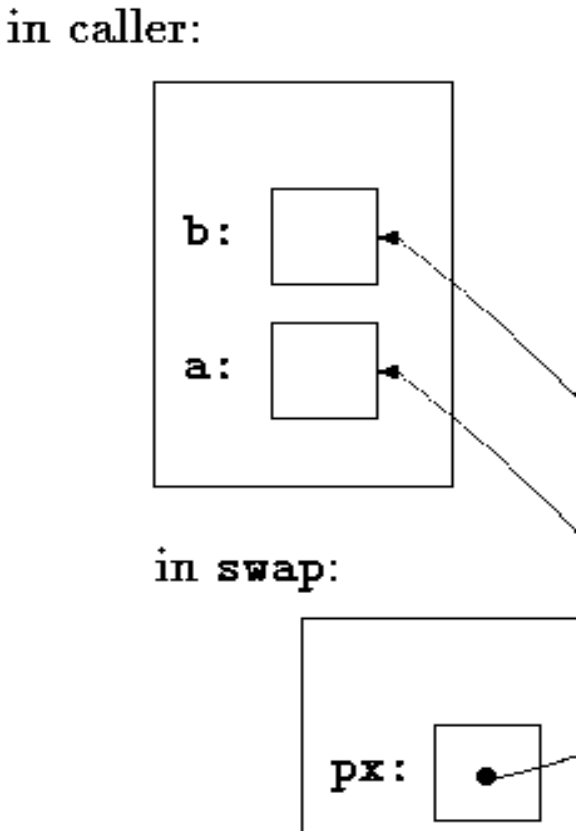
Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps *copies* of a and b.
The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

```
swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers and the operands are accessed indirectly through them.

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
```

Pictorially:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function getint that performs free-format input

conversion by breaking a stream of characters into integer values, one integer per call. getint has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for EOF, that could also be the value of an input integer.

One solution is to have getint return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by scanf as well.

The following loop fills an array with integers by calls to getint:

```
int n, array[SIZE], getint(int *);
for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
;
```

Each call sets array[n] to the next integer found in the input and increments n. Notice that it is essential to pass the address of array[n] to getint. Otherwise there is no way for getint to communicate the converted integer back to the caller.

Our version of getint returns EOF for end of file, zero if the next input is not a number and a positive value if the input contains a valid number.

```
#include <ctype.h>
int getch(void);
void ungetch(int);
/* getint:  get next integer from input into *pn */
int getint(int *pn)
{
        int c, sign;
        while (isspace(c = getch()))    /* skip white space */
    ;
        if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
                ungetch(c);  /* it is not a number */
                return 0;
        }
        sign = (c == '-') ? -1 : 1;
        if (c == '+' || c == '-')
                c = getch();
        for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
        *pn *= sign;
        if (c != EOF)
                ungetch(c);
        return c;
}
```

Throughout getint, *pn is used as an ordinary int variable. We have also used getch and ungetch so the one extra character that must be read can be pushed back onto the input.

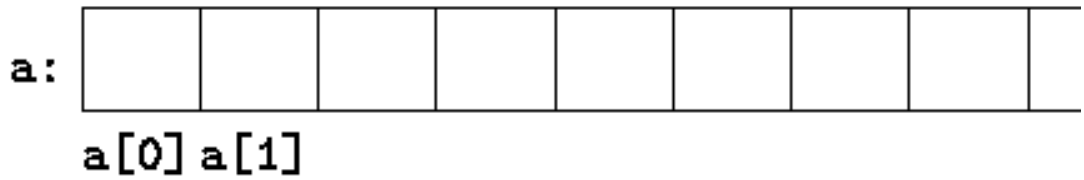
2.3.5 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9].



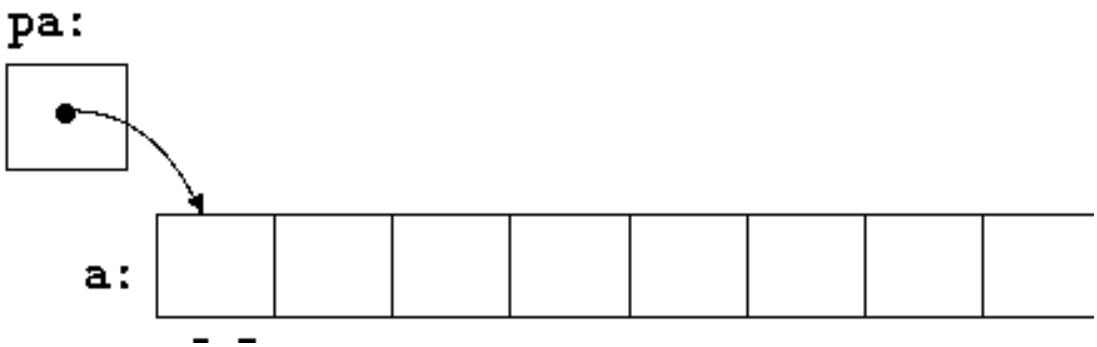
The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets pa to point to element zero of a; that is, pa contains the address of a[0].



Now the assignment

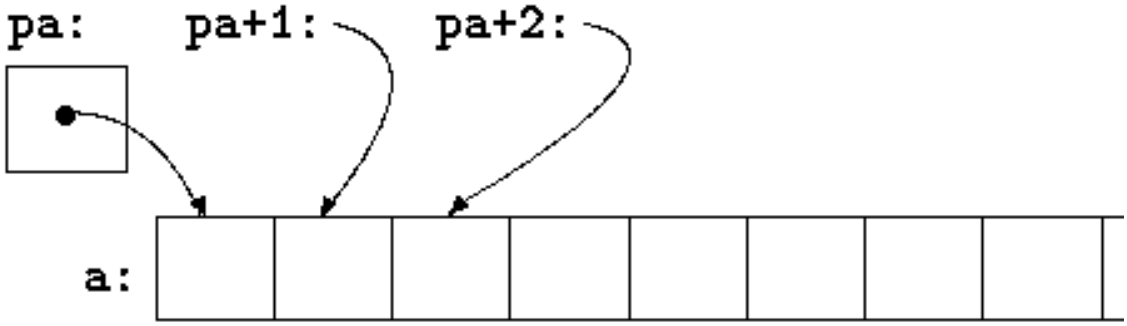
```
x = *pa;
```

will copy the contents of a[0] into x.

If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa and pa-i points i elements before. Thus, if pa points to a[0],

*(pa+1)

refers to the contents of a[1], pa+i is the address of a[i] and *(pa+i) is the contents of a[i].



These remarks are true regardless of the type or size of the variables in the array a. The meaning of ``adding 1 to a pointer,'' and by extension all pointer arithmetic, is that pa+1 points to the next object and pa+i points to the i-th object beyond pa.

The correspondence between indexing and pointer arithmetic is very close. **By definition, the value of a variable or expression of type array is the address of element zero of the array**. Thus after the assignment

pa = &a[0];

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment pa=&a[0] can also be written as

pa = a;

Rather more surprising, at first sight, is the fact that a reference to a[i] can also be written as *(a+i). In evaluating a[i], C converts it to *(a+i) immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. As the other side of this coin, if pa is a pointer, expressions might use it with a subscript; pa[i] is identical to *(pa+i). In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. **A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; constructions like a=pa and a++ are illegal**.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable and so an

array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of strlen, which computes the length of a string.

```
/* strlen:  return length of string s */
int strlen(char *s)
{
        int n;
        for (n = 0; *s != '\0', s++)
                n++;
        return n;
}
```

Since s is a pointer, incrementing it is perfectly legal; s++ has no effect on the character string in the function that called strlen, but merely increments strlen's private copy of the pointer. That means that calls like

```
strlen("hello, world");   /* string constant */
strlen(array);            /* char array[100]; */
strlen(ptr);              /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the variable is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if a is an array,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function f the address of the subarray that starts at a[2]. Within f, the parameter declaration can read

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as f is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in

an array; p[-1], p[-2], and so on are syntactically legal, and refer to the elements that immediately precede p[0]. Of course, it is illegal to refer to objects that are not within the array bounds.

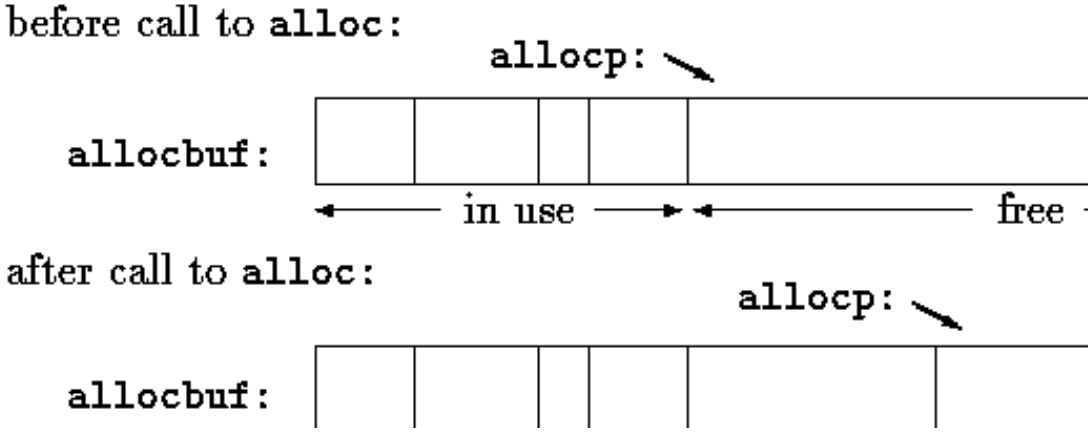
2.3.6 Address Arithmetic

We can perform certain arithmetic operations on pointer variables themselves. If p is a pointer to some element of an array, then p++ increments p to point to the next element and p+=i increments it to point i elements beyond where it currently does. Note that, p++ causes p to point to the next element, not to the next memory location. These and similar constructions are the simples forms of pointer or address arithmetic.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, alloc(n), returns a pointer to n consecutive character positions, which can be used by the caller of alloc for storing characters. The second, afree(p), releases the storage thus acquired so it can be re-used later. The routines are ``rudimentary'' because the calls to afree must be made in the opposite order to the calls made on alloc. That is, the storage managed by alloc and afree is a stack or last-in, first-out. The standard library provides analogous functions called malloc and free that have no such restrictions.

The easiest implementation is to have alloc hand out pieces of a large character array that we will call allocbuf. This array is private to alloc and afree. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared static in the source file containing alloc and afree, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling malloc or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of allocbuf has been used. We use a pointer, called allocp, that points to the next free element. When alloc is asked for n characters, it checks to see if there is enough room left in allocbuf. If so, alloc returns the current value of allocp (i.e., the beginning of the free block), then increments it by n to point to the next free area. If there is no room, alloc returns zero. afree(p) merely sets allocp to p if p is inside allocbuf.




```
#define ALLOCSIZE 10000 /* size of available space */
static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf;  /* next free position */
char *alloc(int n)    /* return pointer to n characters */
{
        if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
                allocp += n;
                return allocp - n; /* old p */
        } else      /* not enough room */
                return 0;
}

void afree(char *p)  /* free storage pointed to by p */
{
        if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
                allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines allocp to be a character pointer and initializes it to point to the beginning of allocbuf, which is the next free position when the program starts. This could also have been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element.

The test

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
```

checks if there's enough room to satisfy a request for n characters. If there is, the new value of allocp would be at most one beyond the end of allocbuf. If the request can be satisfied, alloc returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, alloc must return some signal that there is no space left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer and a pointer may be compared with the constant zero. The symbolic constant NULL is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. NULL is defined in <stdio.h>. We will use NULL henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If p and q point to members of the same array, then relations like ==, !=, <, >=, etc., work properly. For example,

```
p < q
```

is true if p points to an earlier element of the array than q does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the n-th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p. If an int is four bytes, for example, the int will be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and p<q, then q-p+1 is the number of elements from p to q inclusive. This fact can be used to write yet another version of strlen:

```
/* strlen:  return length of string s */
int strlen(char *s)
{
        char *p = s;
```

```
        while (*p != '\0')
                p++;
        return p - s;
}
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is the string length. (The number of characters in the string could be too large to store in an int. The header <stddef.h> defines a type ptrdiff_t that is large enough to hold the signed difference of two pointer values. If we were being cautious, however, we would use size_t for the return value of strlen, to match the standard library version. size_t is the unsigned integer type returned by the sizeof operator.

Pointer arithmetic is consistent: if we had been dealing with floats, which occupy more storage that chars, and if p were a pointer to float, p++ would advance to the next float. Thus we could write another version of alloc that maintains floats instead of chars, merely by changing char to float throughout alloc and afree. All the pointer manipulations automatically take into account the size of the objects pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers or to multiply or divide or shift or mask them or to add float or double to them or even, except for void *, to assign a pointer of one type to a pointer of another type without a cast.

2.3.7 Character Pointers and Functions

A *string constant*, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character '\0' so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; printf receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If pmessage is declared as

```
char *pmessage;
```

then the statement

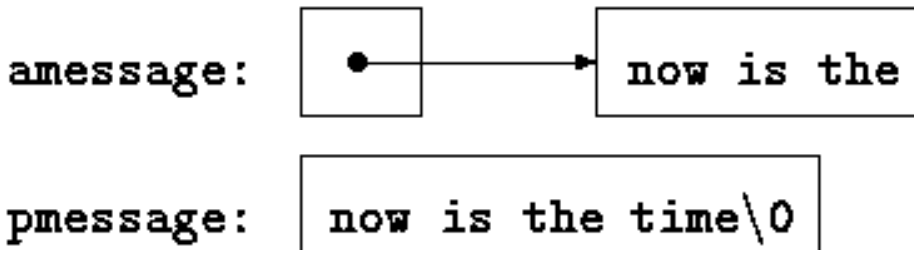
```
pmessage = "now is the time";
```

assigns to pmessage a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

amessage is an array, just big enough to hold the sequence of characters and '\0' that initializes it. Individual characters within the array may be changed but amessage will always refer to the same storage. On the other hand, pmessage is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is strcpy(s,t), which copies the string t to the string s. It would be nice just to say s=t but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version first:

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
        int i;
        i = 0;
        while ((s[i] = t[i]) != '\0')
                i++;
}
```

For contrast, here is a version of strcpy with pointers:

```
/* strcpy:  copy t to s; pointer version */
void strcpy(char *s, char *t)
{
        int i;
        i = 0;
        while ((*s = *t) != '\0') {
                s++;
```

```
        t++;
    }
```

Because arguments are passed by value, strcpy can use the parameters s and t in any way it pleases. Here, they are conveniently initialized pointers, which are matched along the arrays a character at a time, until the '\0' that terminates when t has been copied into s.

In practice, strcpy would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
        while ((*s++ = *t++) != '\0');
}
```

This moves the increment of s and t into the test part of the loop. The value of *t++ is the character that t pointed to before t was incremented; the postfix ++ doesn't change t until after this character has been fetched. In the same way, the character is stored into the old s position before s is incremented. This character is also the value that is compared against '\0' to control the loop. The net effect is that characters are copied from t to s, up and including the terminating '\0'.

As the final abbreviation, observe that a comparison against '\0' is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
/* strcpy:  copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
        while (*s++ = *t++)    ;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The strcpy in the standard library (<string.h>) returns the target string as its function value.

The second routine that we will examine is strcmp(s,t), which compares the character strings s and t and returns negative, zero or positive if s is lexicographically less than, equal to or greater than t. The value is obtained by subtracting the characters at the first position where s and t disagree.

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
```

```
        int i;
        for (i = 0; s[i] == t[i]; i++)
                if (s[i] == '\0')
                        return 0;
        return s[i] - t[i];
}
```

The pointer version of strcmp:

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
        for ( ; *s == *t; s++, t++)
                if (*s == '\0')
                        return 0;
        return *s - *t;
}
```

Since ++ and -- are either prefix or postfix operators, other combinations of * and ++ and -- occur, although less frequently. For example,

```
*--p
```

decrements p before fetching the character that p points to. In fact, the pair of expressions

```
*p++ = val;  /* push val onto stack */
val = *--p;  /* pop top of stack into val */
```

are the standard idiom for pushing and popping a stack.

The header <string.h> contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

2.3.8 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can.

If we have to deal with lines of text, which are of different lengths and which, unlike integers, can't be compared or moved in a single operation, we need a data representation that will cope efficiently and conveniently with variable-length text lines.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array. Two lines can be compared by passing their pointers to strcmp. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.

This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

```
char *lineptr[MAXLINES];  /* pointers to text lines */
```

The main new thing is the declaration for lineptr:

```
char *lineptr[MAXLINES]
```

says that lineptr is an array of MAXLINES elements, each element of which is a pointer to a char. That is, lineptr[i] is a character pointer and *lineptr[i] is the character it points to, the first character of the i-th saved text line.

Since lineptr is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples and writelines can be written instead as

```
/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
        while (nlines-- > 0)
                printf("%s\n", *lineptr++);
}
```

Initially, *lineptr points to the first line; each element advances it to the next line pointer while nlines is counted down.

2.3.9 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on.

Example

```
#include <stdio.h>
#include <conio.h>
main()
{
        int show();     /*Function prototype*/
        int (*p)();     /*Pointer to function declaration*/
        p=show;         /*Assigning address of show to p*/
        (*p)();         /Function call using pointer*/
        printf("%u",show);    /*Displays address of function*/
}
int show()
{
```

```
        clrscr();
        printf("Inside show function whose address is -> ")
}
```

In the above program the variable p is pointer to function. Address of show() is assigned to pointer p. Using function pointer, the function show() is invoked. The output of the program is as under:

Inside show function whose address is -> 554

2.3.10 Summary

C Language facilitates direct access to memory through pointers. A pointer store the address of the variable of type for which pointer is defined. C language also provides parameter passing by address for modifying the values of the actual parameters while making the function calls. Arrays and pointers have strong relation between themselves. Use of pointers makes the programs run faster. Pointer are of great help but should be used very cautiously. If some memory has been allocated to a pointer then it must be freed if it is no longer required because otherwise the memory allocated will not be available for other uses. It will be available only after rebooting of the system. C language does not support string data type but character pointers can solve the purpose. Each string must be terminated by a null otherwise there will be overlapping in the memory read using a character pointer.

2.3.11 Short Answer Type Questions

1. What is the difference between int *A[20] and int (*A)[10];
2. What is the difference between array and pointer?
3. What do you mean by pointer arithmetic?
4. What is the data type of a pointer variable?

2.3.12 Long Answer Type Questions

5. What is the relation between pointer and addressing? Explain.
6. What is the difference between parameter passing by address and by value?
7. What is the relation between pointer and array?
8. Write the code for string copy and string comparison functions using arrays?

2.3.13 Suggested Books

1.	Let Us C	Yashavant Kanetkar
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo	C Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	The C Programming language	Brian W. Kernigham Dennis M. Ritchie

Web Resources

www.cprogramming.com
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

Lesson No. 2.4 **Author : Dharam Veer Sharma**

Structures

2.4.1 Introduction
2.4.2 Objectives of the Lesson
2.4.3 Basics of Structures
2.4.4 Structures and Functions
2.4.5 Arrays of Structures
2.4.6 Pointers to Structures
2.4.7 Self-referential Structures
2.4.8 typedef
2.4.9 union
2.4.10 Summary
2.4.11 Short Answer Type Questions
2.4.12 Long Answer Type Questions
2.4.13 Suggested Books

2.4.1 Introduction

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called ``records'' in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinate, a rectangle is a pair of points, and so on.

The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions and returned by functions. This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.

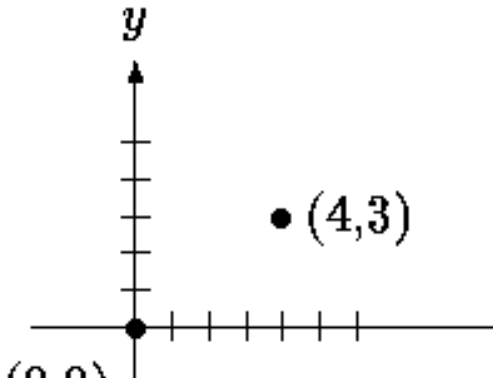
2.4.2 Objectives of the Lesson

Structures and unions are the subject matter of this lesson. We shall delve into details of the declaration, use and importance of structures and unions.

2.4.3 Basics of Structures

A structure is a collection of variable referenced under are name providing a cavendent means of keeping related information together.

Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an *x* coordinate and a *y* coordinate, both integers.



The two components can be placed in a structure declared like this:

```
struct point {
        int x;
        int y;
};
```

The keyword struct introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a *structure tag* may follow the word struct (as with point here). The tag names this kind of structure and can be used subsequently as a shorthand for the part of the declaration in braces.

The variables named in a structure are called *members*. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

A struct declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares x, y and z to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged,

however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of point above,

```
struct point pt;
```

defines a variable pt which is a structure of type struct point. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type.

A member of a particular structure is referred to in an expression by a construction of the form

structure-name.member

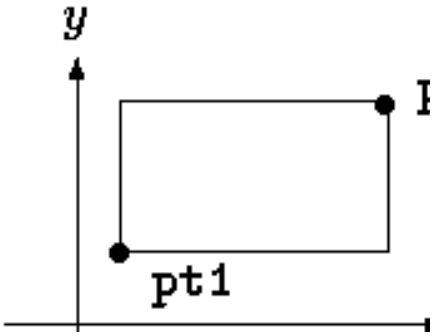
The structure member operator ``." connects the structure name and the member name. To print the coordinates of the point pt, for instance,

```
printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin (0,0) to pt,

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:




```
struct rect {
        struct point pt1;
        struct point pt2;
};
```

The rect structure contains two point structures. If we declare screen as

```
struct rect screen;
```

then

```
screen.pt1.x
```

refers to the *x* coordinate of the pt1 member of screen.

2.4.4 Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members. Copy and assignment inpclude passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure or pass a pointer to it. Each has its good points and bad points.

The first function, makepoint, will take two integers and return a point structure:

```
/* makepoint:  make a point from x and y components */
struct point makepoint(int x, int y)
{
        struct point temp;
        temp.x = x;
        temp.y = y;
        return temp;
}
```

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship.

makepoint function can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
(screen.pt1.y + screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoints:  add two points */
struct addpoint(struct point p1, struct point p2)
{
        p1.x += p2.x;
        p1.y += p2.y;
        return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in p1 rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

As another example, the function ptinrect tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect:  return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
{
        return p.x >= r.pt1.x && p.x < r.pt2.x && p.y >=
        r.pt1.y &&     p.y < r.pt2.y;
}
```

This assumes that the rectangle is presented in a standard form where the pt1 coordinates are less than the pt2 coordinates. The following function returns a rectangle guaranteed to be in canonical form:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect: canonicalize coordinates of rectangle */
struct rect canonrect(struct rect r)
{
        struct rect temp;
        temp.pt1.x = min(r.pt1.x, r.pt2.x);
        temp.pt1.y = min(r.pt1.y, r.pt2.y);
        temp.pt2.x = max(r.pt1.x, r.pt2.x);
        temp.pt2.y = max(r.pt1.y, r.pt2.y);
        return temp;
}
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members. To use pp, we might write, for example,

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher then *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer.

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

```
p->member-of-structure
```

refers to the particular member. So we could write instead

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Both . and -> associate from left to right, so if we have

```
struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly. For example, given the declaration

```
struct {
        int len;
        char *str;
} *p;
```

then

```
++p->len
```

increments len, not p, because the implied parenthesization is ++(p->len). Parentheses can be used to alter binding: (++p)->len increments p before accessing len, and (p++)->len increments p afterward. (This last set of parentheses is unnecessary.)

In the same way, *p->str fetches whatever str points to; *p->str++ increments str after accessing whatever it points to (just like *s++); (*p->str)++ increments whatever str points to; and *p++->str increments p after accessing whatever str points to.

2.4.5 Arrays of Structures

Consider writing a program to count the occurrences of each C keyword. We need an array of character strings to hold the names and an array of integers for the counts. One possibility is to use two parallel arrays, keyword and keycount, as in

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each keyword is a pair:

```
        char *word;
        int count;
```

and there is an array of pairs. The structure declaration

```
        struct key {
                char *word;
                int count;
        } keytab[NKEYS];
```

declares a structure type key, defines an array keytab of structures of this type, and sets aside storage for them. Each element of the array is a structure. This could also be written

```
        struct key {
                char *word;
                int count;
        };
        struct key keytab[NKEYS];
```

Since the structure keytab contains a constant set of names, it is easiest to make it an external variable and initialize it once and for all when it is defined. The structure initialization is analogous to earlier ones - the definition is followed by a list of initializers enclosed in braces:

```
        struct key {
                char *word;
                int count;
        } keytab[] = {
                        "auto", 0,
                        "break", 0,
                        "case", 0,
                        "char", 0,
                        "const", 0,
                        "continue", 0,
                        "default", 0,
                        /* ... */
                        "unsigned", 0,
                        "void", 0,
                        "volatile", 0,
                        "while", 0
        };
```

The initializers are listed in pairs corresponding to the structure members. It would be more precise to enclose the initializers for each "row" or structure in braces, as in

```
        { "auto", 0 },
        { "break", 0 },
        { "case", 0 },
        ...
```

but inner braces are not necessary when the initializers are simple variables or character strings and when all are present. As usual, the number of entries in the array keytab will be computed if the initializers are present and the [] is left empty.

The keyword counting program begins with the definition of keytab. The main routine reads the input by repeatedly calling a function getword that fetches one word at a time. Each word is looked up in keytab with a version of the binary search function. The list of keywords must be sorted in increasing order in the table.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
int binsearch(char *, struct key *, int);
/* count C keywords */
main()
{
        int n;
        char word[MAXWORD];
        while (getword(word, MAXWORD) != EOF)
                if (isalpha(word[0]))
                        if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                                keytab[n].count++;
        for (n = 0; n < NKEYS; n++)
                if (keytab[n].count > 0)
                        printf("%4d %s\n",
        keytab[n].count, keytab[n].word);
        return 0;
}
/* binsearch:  find word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
        int cond;
        int low, high, mid;
        low = 0;
        high = n - 1;
```

```
        while (low <= high) {
                mid = (low+high) / 2;
                if ((cond = strcmp(word, tab[mid].word)) < 0)
                        high = mid - 1;
                else if (cond > 0)
                        low = mid + 1;
                else
                        return mid;
        }
        return -1;
}
```

We will show the function getword in a moment; for now it suffices to say that each call to getword function finds a word, which is copied into the array named as its first argument.

The quantity NKEYS is the number of keywords in keytab. Although we could count this by hand, it's a lot easier and safer to do it by machine, especially if the list is subject to change. One possibility would be to terminate the list of initializers with a null pointer, then loop along keytab until the end is found.

But this is more than is needed, since the size of the array is completely determined at compile time. The size of the array is the size of one entry times the number of entries, so the number of entries is just

size of keytab / *size of* struct key

C provides a compile-time unary operator called sizeof that can be used to compute the size of any object. The expressions

sizeof *object*

and

sizeof (*type name*)

yield an integer equal to the size of the specified object or type in bytes. (Strictly, sizeof produces an unsigned integer value whose type, size_t, is defined in the header <stddef.h>.) An object can be a variable or array or structure. A type name can be the name of a basic type like int or double or a derived type like a structure or a pointer.

In our case, the number of keywords is the size of the array divided by the size of one element. This computation is used in a #define statement to set the value of NKEYS:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

This has the advantage that it does not need to be changed if the type changes.

A size of can not be used in a #if line, because the preprocessor does not parse type names. But the expression in the #define is not evaluated by the preprocessor, so the code here is legal.

Now for the function getword. We have written a more general getword than is necessary for this program, but it is not complicated. getword fetches the next ``word'' from the input, where a word is either a string of letters and digits beginning with a letter or a single non-white space character. The function value is the first character of the word, or EOF for end of file or the character itself if it is not alphabetic.

```
/* getword:  get next word or character from input */
int getword(char *word, int lim)
{
        int c, getch(void);
        void ungetch(int);
        char *w = word;
        while (isspace(c = getch()))
        ;
        if (c != EOF)
                *w++ = c;
        if (!isalpha(c)) {
                *w = '\0';
                return c;
        }
        for ( ; --lim > 0; w++)
                if (!isalnum(*w = getch())) {
                        ungetch(*w);
                        break;
                }
        *w = '\0';
        return word[0];
}
```

getword uses the getch and ungetch. When the collection of an alphanumeric token stops, getword has gone one character too far. The call to ungetch pushes that character back on the input for the next call. getword also uses isspace to skip whitespace, isalpha to identify letters, and isalnum to identify letters and digits; all are from the standard header <ctype.h>.

2.4.6 Pointers to Structures

To illustrate some of the considerations involved with pointers to and arrays of structures, let us write the keyword-counting program again, this time using pointers instead of array indices.

The external declaration of keytab need not change, but main and binsearch do need modification.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
struct key *binsearch(char *, struct key *, int);
/* count C keywords; pointer version */
main()
{
        char word[MAXWORD];
        struct key *p;
        while (getword(word, MAXWORD) != EOF)
                if (isalpha(word[0]))
                        if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                                p->count++;
        for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
                printf("%4d %s\n", p->count, p->word);
        return 0;
}
/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struck key *tab, int n)
{
        int cond;
        struct key *low = &tab[0];
        struct key *high = &tab[n];
        struct key *mid;
        while (low < high) {
                mid = low + (high-low) / 2;
                if ((cond = strcmp(word, mid->word)) < 0)
                        high = mid;
                else if (cond > 0)
                        low = mid + 1;
```

```
            else
                return mid;
        }
        return NULL;
}
```

There are several things worthy of note here. First, the declaration of binsearch must indicate that it returns a pointer to struct key instead of an integer; this is declared both in the function prototype and in binsearch. If binsearch finds the word, it returns a pointer to it; if it fails, it returns NULL.

Second, the elements of keytab are now accessed by pointers. This requires significant changes in binsearch.

The initializers for low and high are now pointers to the beginning and just past the end of the table.

The computation of the middle element can no longer be simply

```
mid = (low+high) / 2    /* WRONG */
```

because the addition of pointers is illegal. Subtraction is legal, however, so high-low is the number of elements, and thus

```
mid = low + (high-low) / 2
```

sets mid to the element halfway between low and high.

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that &tab[-1] and &tab[n] are both outside the limits of the array tab. The former is strictly illegal and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, &tab[n]) will work correctly.

In main we wrote

```
for (p = keytab; p < keytab + NKEYS; p++)
```

If p is a pointer to a structure, arithmetic on p takes into account the size of the structure, so p++ increments p by the correct amount to get the next element of the array of structures and the test stops the loop at the right time.

Don't assume, however, that the size of a structure is the sum of the sizes of its members. Because of alignment requirements for different objects, there may be unnamed ``holes'' in a structure. Thus, for instance, if a char is one byte and an int four bytes, the structure

```
struct {
    char c;
    int i;
};
```

might well require eight bytes, not five. The sizeof operator returns the proper value.

Finally, an aside on program format: when a function returns a complicated type like a structure pointer, as in

```
struct key *binsearch(char *word, struct key *tab, int n)
```

the function name can be hard to see, and to find with a text editor. Accordingly an alternate style is sometimes used:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

This is a matter of personal taste; pick the form you like and hold to it.

2.4.7 Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of *all* the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is likely to grow quadratically with the number of input words.) How can we organize the data to copy efficiently with a list or arbitrary words?

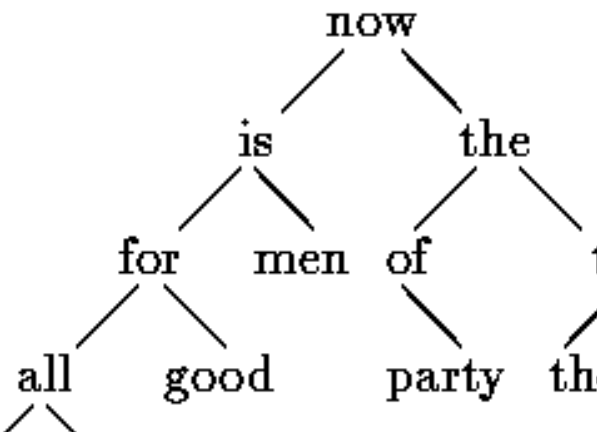
One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though - that also takes too long. Instead we will use a data structure called a **binary tree.**

The tree contains one ``node'' per distinct word; each node contains

- A pointer to the text of the word,
- A count of the number of occurrences,
- A pointer to the left child node,
- A pointer to the right child node.

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence ``now is the time for all good men to come to the aid of their party'', as built by inserting each word as it is encountered:



To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new record is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is most conveniently represented as a structure with four components:

```
struct tnode {     /* the tree node: */
        char *word;          /* points to the text */
        int count;           /* number of occurrences */
        struct tnode *left;   /* left child */
        struct tnode *right;  /* right child */
};
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares left to be a pointer to a tnode, not a tnode itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
        ...
        struct s *p;   /* p points to an s */
};
struct s {
        ...
```

```
        struct t *q;   /* q points to a t */
};
```

2.4.8 typedef

C provides a facility called typedef for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name Length a synonym for int. The type Length can be used in declarations, casts, etc., in exactly the same ways that the int type can be:

```
Length len, maxlen;
Length *lengths[];
```

Similarly, the declaration

```
typedef char *String;
```

makes String a synonym for char * or character pointer, which may then be used in declarations and casts:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Notice that the type being declared in a typedef appears in the position of a variable name, not right after the word typedef. Syntactically, typedef is like the storage classes extern, static, etc. We have used capitalized names for typedefs, to make them stand out.

As a more complicated example, we could make typedefs for the tree nodes shown earlier in this lesson:

```
typedef struct tnode *Treeptr;
typedef struct tnode { /* the tree node: */
        char *word;          /* points to the text */
        int count;           /* number of occurrences */
        struct tnode *left;   /* left child */
        struct tnode *right;  /* right child */
} Treenode;
```

This creates two new type keywords called Treenode (a structure) and Treeptr (a pointer to the structure). Then the routine talloc could become

```
Treeptr talloc(void)
{
        return (Treeptr) malloc(sizeof(Treenode));
}
```

It must be emphasized that a typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as

variables whose declarations are spelled out explicitly. In effect, typedef is like #define, except that since it is interpreted by the compiler, it can cope with textual substitutions that are beyond the capabilities of the preprocessor. For example,

```
typedef int (*PFI)(char *, char *);
```

creates the type PFI, for ``pointer to function (of two char * arguments) returning int,'' which can be used in contexts like

```
PFI strcmp, numcmp;
```

in the sort program.

Besides purely aesthetic issues, there are two main reasons for using typedefs. The first is to parameterize a program against portability problems. If typedefs are used for data types that may be machine-dependent, only the typedefs need change when the program is moved. One common situation is to use typedef names for various integer quantities, then make an appropriate set of choices of short, int, and long for each host machine. Types like size_t and ptrdiff_t from the standard library are examples.

The second purpose of typedefs is to provide better documentation for a program - a type called Treeptr may be easier to understand than one declared only as a pointer to a complicated structure.

2.4.9 union

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program. They are analogous to variant records in pascal.

As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an int, a float, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a union - a single variable that can legitimately hold any of one of several types. The syntax is based on structures:

```
union u_tag {
        int ival;
        float fval;
        char *sval;
} u;
```

The variable u will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to u and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of

which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.

Syntactically, members of a union are accessed as

union-name.member

or

union-pointer->member

just as for structures. If the variable utype is used to keep track of the current type stored in u, then one might see code such as

```
if (utype == INT)
        printf("%d\n", u.ival);
if (utype == FLOAT)
        printf("%f\n", u.fval);
if (utype == STRING)
        printf("%s\n", u.sval);
else
        printf("bad type %d in utype\n", utype);
```

Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures. For example, in the structure array defined by

```
struct {
        char *name;
        int flags;
        int utype;
        union {
                int ival;
                float fval;
                char *sval;
        } u;
} symtab[NSYM];
```

the member ival is referred to as

```
symtab[i].u.ival
```

and the first character of the string sval by either of

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the ``widest'' member and the alignment is appropriate for all of the types in the union. The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address and accessing a member.

A union may only be initialized with a value of the type of its first member; thus union u described above can only be initialized with an integer value.

The storage allocator in shows how a union can be used to force a variable to be aligned on a particular kind of storage boundary.

2.4.10 Summary

Structures facilitate declaration of composite data types of heterogeneous variable. A structure may contain variables of different data types. These represent the logical organization of distinct variables as one unit. Structures may be self referential, where one structure may contain instances of itself. Unions are like structure. The only difference is that for union the memory allocated is equal to the largest sized data types declared in that union.

2.4.11 Short Answer Type Questions

1. Define structure.
2. What is the purpose of using a structure?
3. How an array of structures is declared and used?
4. How a pointer to a structure is declared and used?

2.4.12 Long Answer Type Questions

1. How structures are declared and used ? Explain by giving examples.
2. Give an example of array of structures.
3. What are self referential structures?
4. What is the difference between structure and union?

2.4.13 Suggested Books

1.	Let Us C	Yashavant Kanetkar
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	The C Programming language	Brian W. Kernigham Dennis M. Ritchie

Web Resources

www.cprogramming.com
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

Lesson No. 2.5 **Author : Dharam Veer Sharma**

Searching Techniques

2.5.1 Introduction
2.5.2 Objectives of the Lesson
2.5.3 Linear Search
2.5.4 Binary Search
2.5.5 Summary
2.5.6 Short Answer Type Questions
2.5.7 Long Answer Type Questions
2.5.8 Suggested Books

2.5.1 Introduction

In large data sets, it becomes difficult to find one data item. Automated systems for search generally rally on two basic searching techniques namely: linear and binary search. Linear search is comparatively slow and binary search requires ordered data.

2.5.2 Objectives of the Lesson

In this lesson we shall learn two searching techniques. Dry runs of algorithms will be provided for easier understanding of the procedures of performing search.

2.5.3 Linear Search

As the name suggests this is a sequential search algorithm in which the elements of a set are searched in linear or sequential manner for finding existence of a particular value. If the value to be searched is found then its index or location can be returned. Linear search is applicable when there is no information about the relationship of the values with in a set of values. In such situations the value to be searched is compared with all the values of the set for finding a match. The whole set of values is traversed for the search of the item.

The following example demonstrates how this technique is implemented:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], i, n, item;
        clrscr();
        printf("Enter the number of values in the set -> ");
        scanf("%d",&n);
```

```
        if (n > MAX)
        {
                printf("\nSet size can not be more than 100\n",n);
                exit(1);
        }
        printf("\nEnter the %d items of the set\n");
        for (i=0;i<n;i++)
        {
                printf("Enter the value for element no %d -> ",i+1);
                scanf("%d",&A[i]);
        }
        printf("\nEnter the element to be searched -> ");
        scanf("%d",&item);
        for (i=0;i<n;i++)
        {
                if (A[i] == item)
                        break;
        }
        if (i > n)
                printf("\n%d is not present in the array\n",item);
        else
                printf("\n%d is present at location %d", item, i);
        getch();
}
```

In the above program if the element is present in the array then at the location of the element the loop will break abnormally and location of the element will be known. However if the element is not present then the loop will continue till item has been compared with all elements of the array and it is not present in the array.

The above algorithm can be modified by using a while loop as under:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], i=0, n, item;
        clrscr();
        printf("Enter the number of values in the set -> ");
        scanf("%d",&n);
```

```
        if (n > MAX)
        {
                printf("\nSet size can not be more than 100\n",n);
                exit(1);
        }
        printf("\nEnter the %d items of the set\n");
        for (i=0;i<n;i++)
        {
                printf("Enter the value for element no %d -> ",i+1);
                scanf("%d",&A[i]);
        }
        printf("\nEnter the element to be searched -> ");
        scanf("%d",&item);
        while ((i < n) && (A[i] != item))
                i++
        if (i > n)
                printf("\n%d is not present in the array\n",item);
        else
                printf("\n%d is present at location %d", item, i);
        getch();
}
```

The problem with the linear search is the number of comparisons to be made which are equal to number of elements in the array if the item is not present in the array. This is called worst case. But since elements are unordered in the array, we are left with the choice of sequential search only.

2.5.4 Binary Search

If the elements of the array are ordered, means they have some relation between them, then binary search can be applied. If the numeric or alphanumeric data is sorted in ascending or descending order then the array can be partitioned and search can be concentrated in one partition based on the value of the item to be searched. This is very much similar to the dictionary searching. For searching some word we do not traverse dictionary page by page rather we partition the dictionary pages and then the word lies in either of the two partitions. The other partition is discarded. And we refine our search by following the same procedure on the selected partition. The procedure is repeated till we find the word or we may conclude that the word is missing in the dictionary.

Binary search algorithm can be illustrated by the following example.

If there are 10 elements stored in ordered manner in some array A

15 23 34 45 55 67 78 89 92 97

And we want to search element 47 then the positions of the elements with in array are as follows:

15	23	34	45	55	67	78	89	92	97
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

To begin with the procedure we take start = 0 and end = 9. The algorithm will be executed iteratively till either the element is found or start becomes greater than end i.e. start > end.

Find the value of the mid based on (start+end)/2 therefore (0+9)/2 = 4 (as per integer division).

Now compare the value lying at position mid with the item to be searched i.e. if A[4] == 47 which is not true since A[4] is 55. It means item lies in the lower partition of the array as 47 is less than 55. Therefore we will set the value of end as 3 i.e. end = mid – 1 and start remains unchanged.

Again find the value of mid which is (0+3)/2 = 1 (as per integer division).

Again compare the value lying at position mid with the item to be searched i.e. A[1] == 47 which is not true since A[1] is 23. It means now item lies in the upper partition of the array as 47 is greater than 23. This time we shall modify the value of start and it will be calculated as follows:

start = mid + 1 = 1 + 1 = 2

This time value of start has become greater than end which mean item is not present in the array.

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], i, start, end, mid, item, n;

        clrscr();
        printf("\nEnter the number of values in the array -> ");
        scanf("%d",&n);
        if (n > MAX)
        {
                printf("\nValue can not be more than %d\n",MAX);
                return;
        }
        printf("\nEnter the %d values\n",n);
        for (i = 0;i < n;i++)
```

```
        {
                printf("\nEnter value for element no %d -> ",i+1);
                scanf("%d",&A[i]);
        }
        printf("\nEnter the element to be searched -> ");
        scanf("%d",&item);
        start = 0;
        end = n - 1;
        while (start <= end)
        {
                mid = (start + end)/2;
                if (A[mid] == item)
                        break;
                if (A[mid] > item)
                        end = mid - 1;
                if (A[mid] < item)
                        start = mid + 1;
        }
        if (start <= end)
                printf("\n%d found at location %d",item,mid+1);
        else
                printf("\nElement %d not found",item);
        getch();
}
```

In the above program the terminating condition is when start becomes greater than end. Otherwise the loop breaks when value is found. The nature of binary search algorithm is recursive with the terminating point when either the element is found or start becomes greater than end. Therefore a recursive function can be designed for binary search which is given as below:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], i, start, end, loc, item, n;
        int BSearch(int Array[],int s, int e, int value);
        clrscr();
        printf("\nEnter the number of values in the array -> ");
        scanf("%d",&n);
```

```
        if (n > MAX)
        {
                printf("\n Value can not be more than %d\n",MAX);
                return;
        }
        printf("\nEnter the %d values\n",n);
        for (i = 0;i < n;i++)
        {
                printf("\nEnter value for element no %d -> ",i+1);
                scanf("%d",&A[i]);
        }
        printf("\nEnter the element to be searched -> ");
        scanf("%d",&item);
        start = 0;
        end = n - 1;
        loc = BSearch(A,start,end,item);
        if (loc > 0)
                printf("\nElement %d found at location %d",item,loc+1);
        else
                printf("\nElement %d not found",item);
        getch();
}
int BSearch(int Array[],int s, int e, int value)
{
        int mid = (s + e)/2;
        if (s > e)
                return -1;
        if (Array[mid] == value)
                return mid;
        if (Array[mid] > value)
                        e = mid - 1;
        if (Array[mid] < value)
                        s = mid + 1;
        BSearch(Array, s, e, value);
}
```

In the above example of recursive function, the terminating points are when start becomes greater than end or if the element is found in the array. Otherwise the values of start or end are modified and function is called recursively for finding the item in the array.

As compared to linear search, binary search performs far better. The only precondition is that the array should be ordered, ascending or descending. Binary search performs better because it is based on divide and conquer technique, under which the problem is first divided and then the value is searched in the divided portions. For each iteration the size of the array to be searched in halved. Therefore comparison is not made with all the elements of the array, even if the item is not present.

2.5.5 Summary

Searching is a very important function in information technology. Efficient search algorithms facilitate faster search and results in savings in terms of time. Linear and binary search are the two most common searching techniques. In case of linear search, item is searched sequentially in the array, where as in case of binary search, the array needs to be ordered and then the array is partitioned and search is applied depending on the relationship between of the middle value and the value of the item to be searched. Binary search is recursive in nature.

2.5.6 Short Answer Type Questions

1. Define linear search.
2. Define binary search.
3. Which search algorithm is better?
4. What is the precondition for applying binary search?

2.5.7 Long Answer Type Questions

1. What is the difference between linear and binary search?
2. How linear search is performed? Explain.
3. Discuss the functioning of recursive binary search algorithm.

2.5.8 Suggested Books

1. Application Programming in C R. S. Salaria

Web Resources

www.cprogramming.com
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

Lesson No. 2.6 **Author : Dharam Veer Sharma**

Sorting Techniques

2.6.1 Introduction
2.6.2 Objectives of the Lesson
2.6.3 Bubble Sort
2.6.4 Insertion Sort
2.6.5 Selection Sort
2.6.6 Quick Sort
2.6.7 Comparison of Sorting Techniques
2.6.8 Summary
2.6.9 Short Answer Type Questions
2.6.10 Long Answer Type Questions
2.6.11 Suggested Books

2.6.1 Introduction

Sorting is the process of arranging the data or information in some logical order. This logical order may be ascending or descending in case of numeric values or dictionary order in case of alphanumeric values. Ordered data is easier to manage and searching in the ordered data is better as compared to unordered data, as discussed in lesson 14. For sorting we assume that a list of values is given with the value of total number of elements in the list, which is array for our considerations.

2.6.2 Objectives of the Lesson

In this lesson we shall discuss various sorting algorithms along with some examples for better understanding of the algorithms.

2.6.3 Bubble Sort

This is probably the simplest way to sort an array of objects. Unfortunately it is also the slowest way! The basic idea is to compare two neighboring objects and to swap them if they are in the wrong order.

The idea of bubble sort is to repeatedly move the largest element to the highest index position of the array. Each iteration reduces the effective size of the array. Bubble sort focuses on successive adjacent pairs of elements in the array, compares them and either swaps them or not. In either case, after such a step, the larger of the two elements will be in the higher index position. The focus then moves to the next higher position, and the process is repeated. When the focus reaches the end of the effective array, the largest element will have ``bubbled'' from whatever its original position to the highest index position in the effective array.

Given an array A of numbers, with length n, here's a snippet of C code for bubble sort:

```
for (i=0; i<n-1; i++)
        for (j=0; j<n-1-i; j++)
                if (a[j+1] < a[j])
                { /* compare the two neighbors */
                        tmp = a[j];         /* swap a[j] and a[j+1]      */
                        a[j] = a[j+1];
                        a[j+1] = tmp;
                }
```

As we can see, the algorithm consists of two nested loops. The index j in the inner loop travels up the array, comparing adjacent entries in the array (at j and j+1), while the outer loop causes the inner loop to make repeated passes through the array. After the first pass, the largest element is guaranteed to be at the end of the array, after the second pass, the second largest element is in position and so on. That is why the upper bound in the inner loop (n-1-i) decreases with each pass, we don't have to re-visit the end of the array.

To illustrate the functioning of bubble sort, consider the following array or unordered numbers:

15 47 12 7 20

Following are the steps performed for sorting the above array in ascending order:

Pass1	15	47	12	7	20
	15	12	47	7	20
	15	12	7	47	20
	15	12	7	20	47

Pass 2	12	15	7	20	47
	12	7	15	20	47
	12	7	15	20	47

Pass3	7	12	15	20	47
	7	12	15	20	47

Pass 4	7	12	15	20	47

The output of a given pass becomes input for the next pass. The input of pass 1 is the given array. The sorted part of the array is shown as shaded text.

Following is an important property of the bubble sort algorithm:

Once there is no swapping of elements in a particular pass, there will be no further swapping of elements in the subsequent passes.

This property can be exploited to reduce the unnecessary i.e. redundant passes. For this purpose, we can use a flag to determine if any change has taken place. If there is any change only then we need to proceed with the next pass otherwise stop. The implementation of the algorithm is given as under:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], i, n, pass = 0, temp, flag=1;
        clrscr();
        printf("Enter the number of values in the set -> ");
        scanf("%d",&n);
        if (n > MAX)
        {
                printf("\nSet size can not be more than 100\n");
                exit(1);
        }
        printf("\nEnter the %d items of the set\n");
        for (i=0;i<n;i++)
        {
                printf("Enter the value for element no %d -> ",i+1);
                scanf("%d",&A[i]);
        }
        while ((pass < n-1) && flag)
        {
                flag = 0;
                for (i = 0;i < n-pass-1;i++)
                {
                        if (A[i] > A[i+1])
                        {
                                flag = 1;
                                temp = A[i];
                                A[i] = A[i +1];
                                A[i +1] = temp;
                        }
                }
```

```
            pass++;
        }
        printf("\n Elements after sorting are \n");
        for (i=0;i<n;i++)
            printf("%d ",A[i]);
        getch();
    }
```

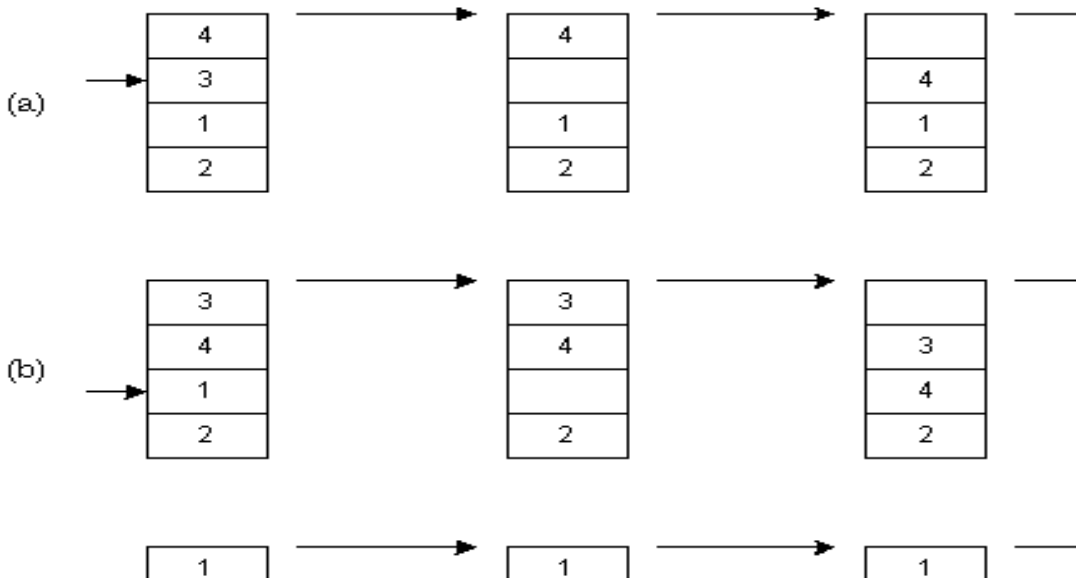
In the above code, swapping is performed only if array elements are not sorted.

2.6.4 Insertion Sort

One of the simplest methods to sort an array is an insertion sort. An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.

Insertion sort is well suited for sorting small data sets or for the insertion of new elements into a sorted sequence.

Starting near the top of the array in figure, we extract the 3. Then the above elements are shifted down until we find the correct place to insert the 3. This process repeats with the next number. Finally we complete the sort by inserting 2 in the correct place.



Insertion Sort

Assuming there are *n* elements in the array, we must index through *n* - 1 entries. For each entry, we may need to examine and shift up to *n* - 1 other entries, resulting in a $O(n^2)$ algorithm. The insertion sort is an *in-place* sort. That is, we sort the array in-place. No extra memory is required. The insertion sort is also a *stable* sort. Stable sorts retain the original ordering of keys when identical keys are present in the input data.

```
void insertionSort(int numbers[], int array_size)
{
        int i, j, index;
        for (i=1; i < array_size; i++)
        {
                index = numbers[i];
                j = i;
                while ((j > 0) && (numbers[j-1] > index))
                {
                        numbers[j] = numbers[j-1];
                        j = j - 1;
                }
                numbers[j] = index;
        }
}
```

Insertion sort maintains a sorted front section of the array [1..i-1]. At each stage, a[i] is inserted at the appropriate point in this sorted section and i is increased.

Part way through sorting, a[1..i-1] is sorted. Consider a[i], and how to insert it into a[1..i-1] so as to make a[1..i] sorted:

```
sorted
 ----------
a:  1  2  3  6  5  4
          ^
          |
          |
          i
```

Examine the elements a[i-1], a[i-2], ..., a[1], moving each of them one place right, and stopping when an element less than or equal to (the original) a[i] is found, or at the left-hand end of the array if no such element is found. Consider the following element :

```
 a[i] = 5
a:  1  2  3  -  6  4
             ^
             |
```

```
                i
```

Place the original a[i] in the vacant location:

```
 sorted
  ------------
a: 1  2  3  5  6  4
      ^
      |
      i
```

Now a[1..i] is sorted. Repeat until i=N.

The worst case occurs when in every step the proper position for the element that is inserted is found at the beginning of the sorted part of the sequence. i.e. in the **while**-loop sequences of length 1, 2, 3, ..., n-1 are scanned. Altogether, these are $(n-1)\cdot n / 2 \in \Theta(n^2)$ operations. This case occurs when the original sequence is sorted in decreasing order.

It is possible to find the inserting position of element a_i faster, namely by binary search. However, moving the elements to the right in order to make room for the element to be inserted takes linear time anyway.

The exact number of steps required by insertion sort is given by the number of inversions of the sequence.

2.6.5 Selection Sort

The selection sort method also requires (N-1) passes to sort the array. The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

From the following explanation it will become clearer:

Pass 1:

Find location LOC of the largest element in the array A[0], A[1] … A[N-1] and interchange A[N-1] with A[LOC] and then A[N-1] is trivially sorted.

Pass 2:

Find location LOC of the largest element in the array A[0], A[1] … A[N-2] and interchange A[N-2] with A[LOC] and then A[N-2] and A[N-1] are trivially sorted.

.
.
.

Pass k:

.
.
.

Pass N-1:

Find location LOC of the largest element in the array A[0], A[1] and interchange A[1] with A[LOC] and then A[0], A[1], A[2], …, A[N-1] are sorted.

To better understand the selection sort algorithm consider the following array A of unsorted numbers

21, 34, 41, 99, 4, 11, 15

Given Array A	21	34	41	99	4	11	15
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Pass1	21	34	41	99	4	11	15
LOC = 3, interchange A[6] with A[LOC] i.e. 99 and 15							
Pass 2	21	34	41	15	4	11	99
LOC = 2, interchange A[5] with A[LOC] i.e. 41 and 11							
Pass 3	21	34	11	15	4	41	99
LOC = 1, interchange A[4] and A[LOC] i.e. 34 and 4							
Pass 4	21	4	11	15	34	41	99
LOC = 0, interchange A[3] and A[LOC] i.e. 21 and 15							
Pass 5	15	4	11	21	34	41	99
LOC = 0, interchange A[2] and A[LOC] i.e. 15 and 11							
Pass 6	11	4	15	21	34	41	99
LOC = 0, interchange A[1] and A[LOC] i.e. 11 and 4							
Final Sorted Array	4	11	15	21	34	41	99

Following is the program for implementing selection sort:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], largest, loc, i, n, temp, pass;
        clrscr();
        printf("\nEnter the number of value in the array -> ");
        scanf("%d",&n);
        if (n > MAX)
        {
                printf("\nValues can not be more than %d\n",MAX);
```

```
            return;
        }
        printf("\nEnter the %d values\n",n);
        for (i = 0;i < n;i++)
        {
            printf("\nEnter value for element no %d -> ",i+1);
            scanf("%d",&A[i]);
        }
        for (pass = n-1;pass >= 0;pass--)
        {
            largest = A[pass];
            loc = pass;
            for (i = pass;i >= 0;i--)
            {
                if (A[i] > largest)
                {
                    largest = A[i];
                    loc = i;
                }
            }
            if (loc != pass)
            {
                temp = A[pass];
                A[pass] = A[loc];
                A[loc] = temp;
            }
        }
        printf("\nThe sorted array is -> \n");
        for (i = 0;i < n;i++)
            printf("%d ",A[i]);
        getch();
    }
```

The above program looks for the largest element in the array and sets at its right position. An alternative is to look for the smallest element and set it at its right place, as explain in the following example

Given Array A	21	34	41	99	4	11	15
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Pass1	21	34	41	99	4	11	15
LOC = 4, interchange A[0] with A[LOC] i.e. 4 and 21							
Pass 2	4	34	41	99	21	11	15
LOC = 5, interchange A[1] with A[LOC] i.e. 11 and 34							
Pass 3	4	11	41	99	21	34	15
LOC = 6, interchange A[2] and A[LOC] i.e. 15 and 41							
Pass 4	4	11	15	99	21	34	41
LOC = 4, interchange A[3] and A[LOC] i.e. 21 and 99							
Pass 5	4	11	15	21	99	34	41
LOC = 5, interchange A[4] and A[LOC] i.e. 34 and 99							
Pass 6	4	11	15	21	34	99	41
LOC = 6, interchange A[5] and A[LOC] i.e. 41 and 99							
Final Sorted Array	4	11	15	21	34	41	99

The program for the finding smallest element in the array and setting it at its right place is as follows:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void main()
{
        int A[MAX], smallest, loc, i, n, temp, pass;
        clrscr();
        printf("\nEnter the number of value in the array -> ");
        scanf("%d",&n);
        if (n > MAX)
        {
                printf("\nValues can not be more than %d\n",MAX);
                return;
        }
        printf("\nEnter the %d values\n",n);
        for (i = 0;i < n;i++)
        {
                printf("\nEnter value for element no %d -> ",i+1);
                scanf("%d",&A[i]);
        }
        for (pass = 0;pass < n-1;pass++)
```

```
        {
                smallest = A[pass];
                loc = pass;
                for (i = pass;i < n;i++)
                {
                        if (A[i] < smallest)
                        {
                                smallest = A[i];
                                loc = i;
                        }
                }
                if (loc != pass)
                {
                        temp = A[pass];
                        A[pass] = A[loc];
                        A[loc] = temp;
                }
                for (i = 0;i < n;i++)
                        printf("%d ",A[i]);
                printf("\n");

        }
        printf("\nThe sorted array is -> \n");
        for (i = 0;i < n;i++)
                printf("%d ",A[i]);
        getch();
}
```

2.6.6 Quick Sort

Quick sort is a sorting algorithm that is based on divide and conquer technique. In order to sort an array, this algorithm first selects a pivot element, which is usually the first element, this element is used to divide the array in two parts in such a way that elements smaller than the pivot element are in the left part and elements greater than the pivot element are in the right part. Then the above procedure is applied on the so divided arrays. This procedure is inherently recursive and terminates when only one element is left in the sub array.

The main task in quick sort is to find the elements that divides the array into two halves and to place it at its proper place in the array. Usually, the procedure places the first element (pivot element) in the array at its final location. This task is

performed as below:

1. To begin with, set the index of the first element of the array to LOC and LEFT variable, and index of the last element of the array to RIGHT variable.
2. Beginning with the element pointer to by RIGHT, the array is scanned from right to left, comparing each element on the way with the element pointed to by LOC, till either
   a. Element smaller than the pivot element pointed to by LOC is found. In this case, the elements are interchanged and procedure continues with step 3.
   b. If the value of RIGHT variable becomes equal to the value of LOC, the procedure terminates here. This condition indicates that the element is placed in its final position LOC.
3. Beginning with the element pointer to by LEFT, the array is scanned from left to right, comparing each element on the way with the element pointed to by LOC (pivot element), till either
   a. Element greater than the element pointer to by LOC is found. In this case, the elements are interchanged and procedure continues with step 1.
   b. If the value of the LEFT variable becomes equal to the value of LOC, the procedure terminates here. This condition indicates that the element is placed in its final position LOC.

As this procedure terminates, the first element of original array will be placed at LOC, its final location in the sorted array. The elements to left of it will be lesser than this element and elements to its right will be greater than this element. Then these two partitioned array become target of sorting and the above procedure is applied to the sub arrays formed after partition. If only one element is left in a partitioned array then the procedure terminated for that sub array. The following illustration explains the above procedure:

Let the array be: 25 10 30 15 20 28

To begin with set LOC = 0, LEFT = 0, RIGHT = 5

25	10	30	15	20	28
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Set LOC = 4, LEFT = 0, RIGHT = 4

20	10	30	15	25	28

Set LOC = 4, LEFT = 1, RIGHT = 4

20	10	30	15	25	28

Set LOC = 4, LEFT = 2, RIGHT = 4

20	10	30	15	25	28

Set LOC = 2, LEFT = 2, RIGHT = 4

20	10	25	15	30	28

Set Loc = 2, LEFT = 2, RIGHT = 3

20	10	25	15	30	28

Set LOC = 3, LEFT = 2, RIGHT = 3

20	10	15	25	30	28

At this point the procedure terminates. Element 25 is placed in its final location in the array, and it divides the array into two sub arrays

20	10	15	and	30	28

Elements in the left sub array are smaller than the pivot element 25 and elements in the right sub array are greater than the pivot element 25.

The recursive procedure for splitting and sorting can be implemented as follows:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 100
void Qsort(int *a, int low, int high);
void main()
{
        int i,n;
        int A[MAX];
        clrscr();
        printf("Enter the number of elements in the array -> ");
        scanf("%d",&n);
        randomize();
        for (i = 0;i < n;i++)
        {
                A[i] = random(100);
```

```
                printf("\nThe element no <%3d> is -> %3d",i+1,A[i]);
        }
        Qsort(A,0,n-1);
        printf("\nThe sorted array is -> \n");
        for (i = 0;i < n;i++)
                printf("%d ",A[i]);
        getch();
}
void Qsort(int *a, int low, int high)
{
        int p = a[low];
        int tmp;
        int l = low, h = high;
        if (low >= high)
          return;
        while (l < h)
        {
                while (p <= a[h])
                {
                        h--;
                        if (l == h)
                                break;
                }
                tmp = a[l];
                a[l] = a[h];
                a[h] = tmp;
                while (p > a[l])
                        l++;
                tmp = a[l];
                a[l] = a[h];
                a[h] = tmp;
        }
        Qsort(a,low,l-1);
        Qsort(a,h+1,high);
}
```

2.6.7 Comparison of Sorting Techniques

Bubble sort is the easiest algorithm for implementation, however it is not the best in terms of performance. Quick sort on the other hand is difficult to implement but is far superior to bubble sort in terms of performance.

Following is the comparison of the above discussed sorting algorithms for best, average and worst cases. Worst means when order of a sorted array is to be reversed, which generally does not happen and all other cases fall under average case. The performance of these algorithms is judged by the number of comparisons made for sorting.

Sorting Algorithm	Average Case	Worst Case
Bubble	N^2	N^2
Insertion	N^2	N^2
Selection	N^2	N^2
Quick	$(n)\log_2 n$	N^2

Quick sort performs better than all other algorithms under average case. In case of 100 element array sorting is done by making 100*7 = 700 comparisons only where as in other algorithms the comparisons are near 10000.

2.6.8 Summary

Sorting is the process of arranging the data in ascending or descending order. Bubble sort is the easiest to implement algorithm but is slowest of all. The basic idea is to compare two neighboring objects, and to swap them if they are in the wrong order.

Insertion sort is implemented by selecting the first smaller number and then placing it at its position after shifting the other elements. The process is repeated for all the numbers in this way. Insertion sort is well suited for sorting small data sets or for the insertion of new elements into a sorted sequence.

Selection sort is based on selecting the smallest or the largest element and placing it at its right place. We repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array and subsequently reducing the effective array size.

Quick sort is the fastest algorithm and is based on recursion. Under this a pivot element is selected and the array is divided in two halves by placing elements, smaller than the pivot element, on its left and larger on its right. Then the process is repeated in the sub array till only one element is left in a sub array.

2.6.9 Short Answer Type Questions

1. What are the various sorting algorithms available?
2. Why bubble sorting is slow?
3. What are the characteristics of insertion sort?
4. Why recursion is the natural way of implementing quick sort?

2.6.10 Long Answer Type Questions

1. Discuss in detail the bubble sort algorithm.
2. Discuss in detail the insertion sort algorithm.
3. Discuss in detail the selection sort algorithm.

4. Discuss in detail the quick sort algorithm.

2.6.11 Suggested Books

1. Application Programming in C R. S. Salaria

Web Resources

www.cprogramming.com
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/