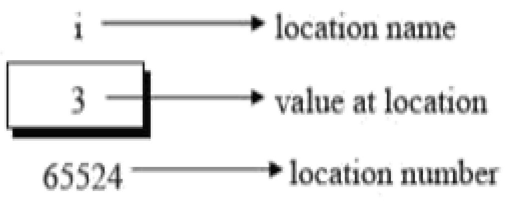
UNIT-III

Unit-3	
Pointer	A pointer is a variable which contains the address in memory of another variable.
Pointer Indirection	An indirection in C is denoted by the operand * followed by the name of a pointer variable. Its meaning is “access the content the pointer points to”.
Function	A function is a group of statements that together perform a task. Every C program has at least one function, which is main()
Storage classes	A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify.
Function prototype	A function prototype or function interface is a declaration of a function that specifies the function's name and type, but omits the function body.
Scope	A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed
Lifetime	The lifetime of a variable or object is the time period in which the variable/object has valid memory.
Static memory allocation	Static memory allocation is the allocation of memory at compile time, before the associated program is executed
Dynamic Memory Allocation	It is a process of allocating or de-allocating the memory at run time it is called as dynamically memory allocation.

Pointer Variables

- If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an *****, and the variable name.

- The general form for declaring a pointer variable is ***type *name***; where *type* is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by *name*. The base type of the pointer defines the type of object to which the pointer will point.
- For example, when you declare a pointer to be of type **int** *, the compiler assumes that any address that it holds points to an integer— whether it actually does or not.
- Therefore, when you declare a pointer, you must make sure that its type is compatible with the type of object to which you want to point.



The Pointer Operators

There are two pointer operators: ***** (at address) and **&** (the address of).

The **&** is a unary operator that returns the memory address of its operand.

For example, m = &count; places into **m** the memory address of the variable **count**. This address is the computer's internal location of the variable.It has nothing to do with the value of **count.** Therefore, the preceding assignment statement can be verbalized as "**m** receives the address of **count.**" To understand the above assignment better, assume that the variable **count** uses memory location 2000 to store its value. Also assume that **count** has a value of 100. Then, after the preceding assignment, **m** will have the value 2000. The second pointer operator, *, is the complement of **&**. It is a unary operator that returns the value located at the address that follows. For example, if **m** contains the memory address of the variable **count**, q = *m; places the value of **count** into **q**. Thus, **q** will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in **m**. You can think of * as "." In this case, the preceding statement can be verbalized as "**q** receives the value at address **m**."

Pointer Expressions

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions, such as assignments, conversions, and arithmetic.

Pointer Assignments

You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward. For example:

```
#include <stdio.h>
int main(void)
{
int x = 99;
int *p1, *p2;
p1 = &x;
p2 = p1;
/* print the value of x twice */
printf(''Values at p1 and p2: %d %
d\n", *p1, *p2);
/* print the address of x twice */
printf("Addresses pointed to by p1 and p2: %p %p", p1, p2);
return 0;
}
```

After the assignment sequence

```
p1 = &x;
p2 = p1;
```

p1 and **p2** both point to **x**. Thus, both **p1** and **p2** refer to the same object. Sample output from the program, which confirms this, is shown here.

```
Values at p1 and p2: 99 99
Addresses pointed to by p1 and p2: 0063FDF0 0063FDF0
```

Notice that the addresses are displayed by using the **%p printf( )** format specifier, which causes **printf( )** to display an address in the format used by the host computer.

It is also possible to assign a pointer of one type to a pointer of another type. However, doing so involves a pointer conversion, which is the subject of the next section.

Pointer Conversions

One type of pointer can be converted into another type of pointer. There are two general categories of conversion: those that involve **void** * pointers, and those that don't. Each is examined here. In C, it is permissible to assign a **void** * pointer to any other type of pointer. It is also permissible to assign any other type of pointer to a **void** * pointer. A **void** * pointer is called a *generic pointer*. The **void** * pointer is used to specify a pointer whose base type is unknown. The **void** * type allows a function to specify a parameter that is capable of receiving any type of pointer argument without reporting a type mismatch. It is also used to refer to raw memory (such as that returned by the **malloc( )** function described later in this chapter) when the semantics of that memory are not known. No explicit cast is required to convert to or from a **void** * pointer. Except for **void** *, all other pointer conversions must be performed by using an explicit cast. However, the conversion of one type of pointer into another type may create undefined behavior. For example, consider the following program that attempts to assign the value of **x** to **y**, through the pointer **p**. This program compiles without error, but does not produce the desired result.

```
#include <stdio.h>
int main(void)
{

double x = 100.1, y;
int *p;
/* The next statement causes p (which is an
integer pointer) to point to a double. */
p = (int *) &x;
/* The next statement does not operate as expected. */
y = *p; /* attempt to assign y the value x through p */
/* The following statement won't output 100.1. */
printf(''The (incorrect) value of x is: %f", y);
return 0;
}
```

Notice that an explicit cast is used when assigning the address of **x** (which is implicitly a **double** * pointer) to **p**, which is an **int** * pointer. While this cast is correct, it does not cause the program to act as intended (at least not in most environments). To understand the problem, assume 4-byte **int**s and 8-byte **double**s. Because **p** is declared as an integer pointer, only 4 bytes of information will be transferred to **y** by this assignment statement, y = *p; not the 8 bytes that make up a **double**. Thus, even though **p** is a valid pointer, the fact that it points to a **double** does not change the fact that operations on it expect **int** values. Thus, the use to which **p** is put is invalid. The preceding example reinforces the rule stated earlier: Pointer operations are performed relative to the base type of the pointer. While it is technically permissible for a pointer to point to some other type of object, the pointer will still ''think" that it is pointing to an object of its base type. Thus, pointer operations are governed by the type of the pointer, not the type of the object being pointed to. One other pointer conversion is allowed: You can convert an integer into a pointer or a pointer into an integer. However, you must use an explicit cast, and the result of such a conversion is implementation defined and may result in undefined behavior. (A cast is not needed when converting zero, which is the null pointer.)

Pointers and Arrays

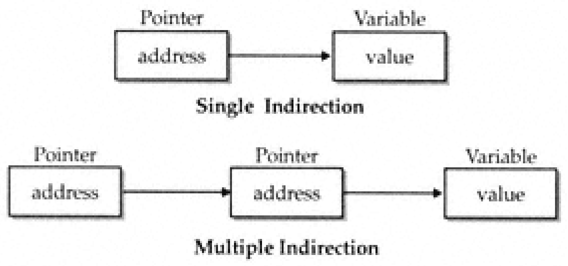
There is a close relationship between pointers and arrays. Consider this program fragment: char str[80], *p1; p1 = str; Here, **p1** has been set to the address of the first array element in **str**. To access the fifth element in **str**, you could write str[4] or *(p1+4) Both statements will return the fifth element. Remember, arrays start at 0. To access the fifth element, you must use 4 to index **str**. You also add 4 to the pointer **p1** to access the fifth element because **p1** currently points to the first element of **str**. (Recall that an array name without an index returns the starting address of the array, which is the address of the first element.) The preceding example can be generalized. In essence, C provides two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C programmers often use pointers to access array elements. These two versions of **putstr( )**— one with array indexing and one with pointers— illustrate how you can use pointers in place of array indexing. The **putstr( )** function writes a string to the standard output device one character at a time.

```
/* Index s as an array. */
void putstr(char *s)
{
register int t;
for(t=0; s[t]; ++t) putchar(s[t]);
}
/* Access s as a pointer. */
void putstr(char *s)
{
while(*s) putchar(*s++);
}
```

Most professional C programmers would find the second version easier to read and understand. Depending upon the compiler, it might also be more efficient. In fact, the pointer version is the way routines of this sort are commonly written in C.

Multiple Indirection

You can have a pointer point to another pointer that points to the target value. This situation is called *multiple indirection*, or *pointers to pointers*. Pointers to pointers can be confusing. Figure 5-3 helps clarify the concept of multiple indirection. As you can see, the value of a normal pointer is the address of the object that contains the desired value. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the desired value. Multiple indirection can be carried on to whatever extent desired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and prone to conceptual errors. A variable that is a pointer to a pointer must be declared as such. You do this by placing an additional asterisk in front of the variable name. For example, the following declaration tells the compiler that **newbalance** is a pointer to a pointer of type **float**: float **newbalance; You should understand that **newbalance** is not a pointer to a floating-point number but rather a pointer to a **float** pointer.



To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
#include <stdio.h>
int main(void)
{
int x, *p, **q;
x = 10;
p = &x;
q = &p;
printf("%d", **q); /* print the value of x */
return 0;
}
```

Initializing Pointers

After a nonstatic, local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global and static local pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program— and possibly your computer's operating system as well— a very nasty type of error! There is an important convention that most C programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is given the value null (which is zero). Null is used because C guarantees that no object will exist at the null address. Thus, any pointer that is null implies that it points to nothing and should not be used. One way to give a pointer a null value is to assign zero to it. For example, the following initializes **p** to null. char *p = 0; Additionally, many of C's headers, such as **<stdio.h>** , define the macro **NULL**, which is a null pointer constant. Therefore, you will often see a pointer assigned null using a statement such as this: p = NULL; However, just because a pointer has a null value, it is not necessarily ''safe." The use of null to indicate unused pointers is simply a convention that programmers follow. It is not a rule enforced by the C language. For example, the following sequence, although incorrect, will still be compiled without error: int *p = 0; *p = 10; /* wrong! */ In this case, the assignment through **p** causes an assignment at 0, which will usually cause a program crash. Because a null pointer is assumed to be unused, you can use the null pointer to make many of your pointer routines easier to code and more efficient. For example, you can use a null pointer to mark the end of a pointer array. A routine that accesses that array knows that it has reached the end when it encounters the null value. The **search( )** function shown in the following program illustrates this type of approach. Given a list of names, **search( )** determines whether a specified name is in that list.

```
#include <stdio.h>
#include <string.h>
int search(char *p[], char *name);
char *names[] = {
"Herb","Rex","Dennis",''John ",NULL}; /* null pointer constant ends the list */
int main(void)
{
```

```
if(search(names, "Dennis") != -1)
printf("Dennis is in list.\n");
if(search(names, "Bill") == -1)
printf("Bill not found.\n");
return 0;
}
/* Look up a name. */
int search(char *p[], char *name)
{
register int t;
for(t=0; p[t]; ++t)
if(!strcmp(p[t], name)) return t;
return -1; /* not found */
}
```

The **search( )** function is passed two parameters. The first, **p**, is an array of **char *** pointers that point to strings containing names. The second, **name**, is a pointer to a string that points to the name being sought. The **search( )** function searches through the list of pointers, seeking a string that matches the one pointed to by **name**. The **for** loop inside **search( )** runs until either a match is found or a null pointer is encountered. Assuming the end of the array is marked with a null, the condition controlling the loop is false when the end of the array is reached. That is, **p[t]** will be false when **p[t]** is null. In the example, this occurs when the name Bill is tried, since it is not in the list of names. C programmers commonly initialize **char *** pointers to point to string constants, as the previous example shows. To understand why this works, consider the following statement: char *p = "hello world"; As you can see, **p** is a pointer, not an array. This raises a question: Where is the string constant "hello world" being held? Since **p** is not an array, it can't be stored in **p**. Yet, the string is obviously being stored somewhere. The answer to the question is found in the way C compilers handle string constants. The C compiler creates what is called a *string table*, which stores the string constants used by the program. Therefore, the preceding declaration statement places the address of ''hello world", as stored in the string table, into the pointer **p**. Throughout a program, **p** can be used like any other string. For example, the following program is perfectly valid:

```
#include <stdio.h>
#include <string.h>
char *p = "hello world";
int main(void)
{
register int t;
/* print the string forward and backwards */
printf(p);
for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);
return 0;
}
```

Pointers to Functions

A particularly confusing yet powerful feature of C is the *function pointer*. A function has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions. You obtain the address of a function by using the function's name without any parentheses or arguments. (This is similar to the way an array's address is obtained when only the array name, without indexes, is used.) To see how this is done, study the following program, which compares two strings entered by the user. Pay close attention to the declarations of **check( )** and the function pointer **p**, inside **main( )**.

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int main(void)
{
```

```
char s1[80], s2[80];
int (*p)(const char *, const char *); /* function pointer */
p = strcmp; /* assign address of strcmp to p */
printf("Enter two strings.\n");
gets(s1);
gets(s2);
check(s1, s2, p); /* pass address of strcmp via p */
return 0;
}
void check(char *a, char *b,
int (*cmp) (const char *, const char *))
{
printf(''Testing for equality.\n");
if(!(*cmp)(a, b)) printf("Equal");
else printf("Not Equal");
}
```

Let's look closely at this program. First, examine the declaration for **p** in **main( )**. It is shown here:

```
int (*p)(const char *, const char *);
```

This declaration tells the compiler that **p** is a pointer to a function that has two **const char** * parameters, and returns an **int** result. The parentheses around **p** are necessary in order for the compiler to properly interpret this declaration. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ. Next, examine the **check( )** function. It declares three parameters: two character pointers, **a** and **b**, and one function pointer, **cmp**. Notice that the function pointer is declared using the same format as was **p** inside **main( )**. Thus, **cmp** is able to receive a pointer to a function that takes two **const char** * arguments and returns an **int** result. Like the declaration for **p**, the parentheses around the ***cmp** are necessary for the compiler to interpret this statement correctly. When the program begins, it assigns **p** the address of **strcmp( )**, the standard string comparison function. Next, it prompts the user for two strings, and then it passes pointers to those strings along with **p** to

check( ), which compares the strings for equality. Inside **check( )**, the expression (*cmp)(a, b) calls **strcmp( )**, which is pointed to by **cmp**, with the arguments **a** and **b**. The parentheses around ***cmp** are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, can also be used. cmp(a, b);

The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer (that is, that **cmp** is a function pointer, not the name of a function). Also, the first style was the form originally specified by C.

Note that you can call **check( )** by using **strcmp( )** directly, as shown here:

check(s1, s2, strcmp);

This eliminates the need for an additional pointer variable, in this case.

You may wonder why anyone would write a program like the one just shown. Obviously, nothing is

gained, and significant confusion is introduced. However, at times it is advantageous to pass functions as parameters or to create an array of functions. For example, when an interpreter is written, the parser (the part that processes expressions) often calls various support functions, such as those that compute mathematical operations (sine, cosine, tangent, etc.), perform I/O, or access system resources. Instead of having a large **switch** statement with all of these functions listed in it, an array of function pointers can be created. In this approach, the proper function is selected by its index.

You can get a better idea of the value of function pointers by studying the expanded version of the previous example, shown next. In this version, **check( )** can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function. When checking for numeric equality, the string ''0123" will compare equal to "123", even though the strings, themselves, differ.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int compvalues(const char *a, const char *b);
```

pointers to those strings along with **p** to **check( )**, which compares the strings for equality. Inside **check( )**, the expression (*cmp)(a, b) calls **strcmp( )**, which is pointed to by **cmp**, with the arguments **a** and **b**. The parentheses around ***cmp** are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, can also be used.

```
cmp(a, b);
```

The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer (that is, that **cmp** is a function pointer, not the name of a function). Also, the first style was the form originally specified by C.

Note that you can call **check( )** by using **strcmp( )** directly, as shown here:

```
check(s1, s2, strcmp);
```

This eliminates the need for an additional pointer variable, in this case. You may wonder why anyone would write a program like the one just shown. Obviously, nothing is gained, and significant confusion is introduced. However, at times it is advantageous to pass
functions as parameters or to create an array of functions. For example, when an interpreter is written, the parser (the part that processes expressions) often calls various support functions, such as those that compute mathematical operations (sine, cosine, tangent, etc.), perform I/O, or access system resources. Instead of having a large **switch** statement with all of these functions listed in it, an array of function pointers can be created. In this approach, the proper function is selected by its index.

You can get a better idea of the value of function pointers by studying the expanded version of the previous example, shown next. In this version, **check( )** can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function. When checking for numeric equality, the string ''0123" will compare equal to "123", even though the strings, themselves, differ.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
```

```
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int compvalues(const char *a, const char *b);
int main(void)
{
char s1[80], s2[80];
printf ("Enter two values or two strings.\n");
gets (s1);
gets(s2);
if(isdigit(*sl)) {
printf(''Testing values for equality.\n");
check(s1, s2, compvalues);
}
else {
printf("Testing strings for equality.\n");
check(s1, s2, strcmp);
}
return 0;
}
void check(char *a, char *b,
int (*cmp)(const char *, const char *))
{
if(!(*cmp)(a, b)) printf("Equal");
else printf("Not Equal");
}
int compvalues(const char *a, const char *b)
{
if(atoi(a)==atoi(b)) return 0;
else return 1;
}
```

DYNAMIC ALLOCATION FUNCTIONS

If there is not enough available memory to satisfy the **malloc( )** request, an allocation failure occurs and **malloc( )** returns a null. The code fragment shown here allocates 1,000 bytes of contiguous memory: char *p;

```
p = malloc(1000); /* get 1000 bytes */
```

After the assignment, **p** points to the first of 1,000 bytes of free memory.

In the preceding example, notice that no type cast is used to assign the return value of **malloc( )** to **p**.

As explained, a **void** * pointer is automatically converted to the type of the pointer on the left side of an assignment. (However, this automatic conversion *does not* occur in C++, and an explicit type cast is needed.)

The next example allocates space for 50 integers. Notice the use of **sizeof** to ensure portability.

```
int *p;
p = malloc(50*sizeof(int));
```

Since the heap is not infinite, whenever you allocate memory, you must check the value returned by

malloc( ) to make sure that it is not null before using the pointer. Using a null pointer will almost certainly crash your program. The proper way to allocate memory and test for a valid pointer is illustrated in this code fragment:

```
p = malloc(100);
if(!p) {
printf(''Out of memory.\n");
exit (1);
}
```

Of course, you can substitute some other sort of error handler in place of the call to **exit( )**. Just make sure that you do not use the pointer **p** if it is null.

The **free( )** function is the opposite of **malloc( )** in that it returns previously allocated memory to the system. Once the memory has been freed, it may be reused by a subsequent call to **malloc( )**. The function **free( )** has this prototype:

void free(void **p*);

Here, *p* is a pointer to memory that was previously allocated using **malloc( )**. It is critical that you *never* call **free( )** with an invalid argument; this will damage the allocation system.

C's dynamic allocation subsystem is used in conjunction with pointers to support a variety of important programming constructs, such as linked lists and binary trees. Several examples of these are included in Part Four. Another important use of dynamic allocation is discussed next: dynamically allocated arrays.

Problems with Pointers

Nothing will get you into more trouble than a wild pointer! Pointers are a mixed blessing. They give you tremendous power, but when a pointer is used incorrectly, or contains the wrong value, it can be a very difficult bug to find.

An erroneous pointer is difficult to find because the pointer, by itself, is not the problem. The trouble starts when you access an object through that pointer. In short, when you attempt to use a bad pointer, you are reading or writing to some unknown piece of memory. If you read from it, you will get a garbage value, which will probably cause your program to malfunction. If you write to it, you might be writing over other pieces of your code or data. In either case, the problem might not show up until later in the execution of your program and may lead you to look for the bug in the wrong place. There may be little or no evidence to suggest that the pointer is the original cause of the problem. Programmers lose sleep over this type of bug time and time again.

Because pointer errors are so troublesome, you should, of course, do your best never to generate one. To help you avoid them, a few of the more common errors are discussed here. The classic example of a pointer error is the *uninitialized pointer*. Consider this program:

```
/* This program is wrong. */
int main(void)
{
int x, *p;
x = 10;
*p = x; /* error, p not initialized */
return 0;
}
```

This program assigns the value 10 to some unknown memory location. Here is why. Since the pointer **p** has never been given a value, it contains an unknown value when the assignment ***p = x** takes place. This causes the value of **x** to be written to some unknown memory location. This type of problem often goes unnoticed when the program is small because the odds are in favor of **p** containing a ''safe" address–one that is not in your code, data area, or operating system. However, as your program grows, the probability increases of **p** pointing to something vital. Eventually, your program stops working. In this simple example, most compilers will issue a warning message stating that you are attempting to use an uninitialized pointer, but the same type of error can occur in more roundabout ways that the compiler can't detect.

A second common error is caused by a simple misunderstanding of how to use a pointer. Consider the following:

```
/* This program is wrong. */
#include <stdio.h>
int main(void)
{
int x, *p;
x = 10;
p = x;
printf("%d", *p);
return 0;
}
```

The call to **printf( )** does not print the value of **x**, which is 10, on the screen. It prints some unknown value because the assignment

p = x; is wrong. That statement assigns the value 10 to the pointer **p**. However, **p** is supposed to contain an address, not a value. To correct the program, write p = &x; As with the earlier error, most compilers will issue at least a warning message when you attempt to
assign **x** to **p**. But as before, this error can manifest itself in a more subtle fashion which the compiler can't detect. Another error that sometimes occurs is caused by incorrect assumptions about the placement of variables in memory. In general, you cannot know where your data will be placed in memory, or whether it will be placed there the same way again, or whether different compilers will treat it in the same way. For these reasons, making any comparisons between pointers that do not point to a common object may yield unexpected results. For example,

```
char s[80], y[80];
char *p1, *p2;
p1 = s;
p2 = y;
if(p1 < p2) . . .
```

is generally an invalid concept. (In very unusual situations, you might use something like this to determine the relative position of the variables. But this would be rare.)
A related error results when you assume that two adjacent arrays may be indexed as one by simply incrementing a pointer across the array boundaries. For example:

```
int first[10], second[10];
int *p, t;
p = first;
for(t=0; t<20; ++t) *p++ = t;
```

This is not a good way to initialize the arrays **first** and **second** with the numbers 0 through 19. Even though it may work on some compilers under certain circumstances, it assumes that both arrays will be placed back to back in memory with **first** first. This may not always be the case.
The next program illustrates a very dangerous type of bug. See if you can find it.

```
/* This program has a bug. */
#include <string.h>
#include <stdio.h>
```

```
int main(void)
{
char *p1;
char s[80];
p1 = s;
do {
gets(s); /* read a string */
/* print the decimal equivalent of each
character */
while(*p1) printf('' %d", *p1++);
} while(strcmp(s, "done"));
return 0;
}
```

This program uses **p1** to print the ASCII values associated with the characters contained in **s**. The problem is that **p1** is assigned the address of **s** only once, outside the loop. The first time through the loop, **p1** points to the first character in **s**. However, the second time through, it continues where it left off because it is not reset to the start of **s**. This next character may be part of the second string, another variable, or a piece of the program! The proper way to write this program is

```
/* This program is now correct. */
#include <string.h>
#include <stdio.h>
int main(void)
{
char *p1;
char s[80];
do {
p1 = s; /* reset p1 to beginning of s */
gets(s); /* read a string */
/* print the decimal equivalent of each
character */
while(*p1) printf(" %d", *p1++);
} while(strcmp(s, "done"));
return 0;
}
```

Here, each time the loop iterates, **p1** is set to the start of the string. In general, you should remember to reinitialize a pointer if it is to be reused. The fact that handling pointers incorrectly can cause tricky bugs is no reason to avoid using them. Just be careful, and make sure that you know where each pointer is pointing before you use it.

Understanding the Scope of a Function

The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. The scopes defined by C were described in Chapter 2. Here we will look more closely at one specific scope: the one defined by a function. Each function is a discrete block of code. Thus, a function defines a block scope. This means that a function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use **goto** to jump into the middle of another function.) The code that constitutes the body of a function is hidden from the rest of the program, and unless it uses global variables, it can neither affect nor be affected by other parts of the program. Stated another way, the code and data defined within one function cannot interact with the code or data defined in another function because the two functions have different scopes. Variables that are defined within a function are local variables. A local variable comes into
existence when the function is entered and is destroyed upon exit. Thus, a local variable cannot hold its value between function calls. The only exception to this rule is when the variable is declared with the **static** storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limit its scope to the function. (See Chapter 2 for additional information on global and local variables.)

The formal parameters to a function also fall within the function's scope. This means that a parameter is known throughout the entire function. A parameter comes into existence when the function is called and is destroyed when the function is exited.

All functions have file scope. Thus, you cannot define a function within a function. This is why C is not technically a block-structured language.

Function Arguments

If a function is to accept arguments, it must declare the parameters that will receive the values of the arguments. As shown in the following function, the parameter declarations occur after the function name.

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
while (*s)
if(*s==c) return 1;
else s++;
return 0;
}
```

The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character **c** is part of the string **s**; otherwise, it returns 0.

Even though parameters perform the special task of receiving the value of the arguments passed to the function, they behave like any other local variable. For example, you can make assignments to a function's formal parameters or use them in an expression.

Call by Value, Call by Reference

In a computer language there are two ways that arguments can be passed to a subroutine. The first is *call by value*. This method copies the *value of* an argument into on the argument.

Call by reference is the second way of passing arguments to a subroutine. In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. With few exceptions, C uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the following program:

```
#include <stdio.h>
int sqr(int x);
int main(void)
{
int t=10;
printf("%d %d", sqr(t), t);
return 0;
}
int sqr(int x)
{
x = x*x;
return(x);
}
```

In this example, the value of the argument to **sqr( )**, 10, is copied into the parameter **x**. When the assignment **x = x*x** takes place, only the local variable **x** is modified. The variable **t**, used to call **sqr ( )**, still has the value 10. Hence, the output is **100 10**. Remember that it is a copy of the value of the argument that is passed into a function. What occurs inside the function has no effect on the variable used in the call.

Creating a Call by Reference

Even though C uses call by value for passing parameters, you can create a call by reference by passing a pointer to an argument, instead of passing the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other argument. Of course, you need to declare the parameters as pointer types. For example, the function **swap( )**, which exchanges the values of the two integer variables pointed to by its arguments, shows how:

```
void swap(int *x, int *y)
{
```

```
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */
}
```

The **swap( )** function is able to exchange the values of the two variables pointed to by **x** and **y** because their addresses (not their values) are passed. Within the function, the contents of the variables are accessed using standard pointer operations, and their values are swapped.

Remember that **swap( )** (or any other function that uses pointer parameters) must be called with the *addresses of the arguments*. The following program shows the correct way to call **swap( )**:

```
#include <stdio.h>
void swap(int *x, int *y);
int main (void)
{
int i, j;
i = 10;
j = 20;
printf("i and j before swapping: %d %d\n", i, j);
swap(&i, &j); /* pass the addresses of i and j */
printf("i and j after swapping: %d %d\n", i, j);
return 0;
}
void swap(int *x, int *y)
{
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */
}
```

The output from this program is shown here:

```
i and j before swapping: 10 20
i and j after swapping: 20 10
```

The return Statement

The mechanics of **return** are described in Chapter 3. As explained, it has two important uses. First, it causes an immediate exit from the function. That is, it causes program execution to return to the calling code. Second, it can be used to return a value. The following sections examine how the **return** statement is applied.

Returning from a Function

A function terminates execution and returns to the caller in two ways. The first occurs when the last statement in the function has executed, and, conceptually, the function's ending curly brace (}) is encountered. (Of course, the curly brace isn't actually present in the object code, but you can think of it in this way.) For example, the **pr_reverse( )** function in this program simply prints the string **I like C** backwards on the screen and then returns.

```
#include <string.h>
#include <stdio.h>
void pr_reverse(char *s);
int main(void)
{
pr_reverse(''I like C");
return 0;
}
void pr_reverse(char *s)
{
register int t;
for(t=strlen(s)-l; t>=0; t--) putchar(s[t]);
}
```

Once the string has been displayed, there is nothing left for **pr_reverse( )** to do, so it returns to the place from which it was called. Actually, not many functions use this default method of terminating their execution. Most functions rely on the **return** statement to stop execution either because a value must be returned or to make a function's code simpler and more efficient. A function may contain several **return** statements. For example, the **find_substr( )** function in the following program returns the starting position of a substring within a string, or it returns –1 if no match is found. It uses two **return** statements to simplify the coding.

```
#include <stdio.h>
int find_substr(char *s1, char *s2);
int main(void)
{
if(find_substr("C is fun", "is") != -1)
printf("Substring is found.");
return 0;
}
/* Return index of first match of s2 in s1. */
int find_substr(char *s1, char *s2)
{
register int t;
char *p, *p2;
for(t=0; s1[t]; t++)
p = &s1[t];
p2 = s2;
while(*p2 && *p2==*p) {
p++;
p2++;
}
if(!*p2) return t; /* 1st return */
}
return -1; /* 2nd return */
}
```

Type Qualifiers

C defines type qualifiers that control how variables may be accessed or modified. C89 defines two of these qualifiers: **const** and **volatile**. (C99 adds a third, called **restrict**, which is described in Part Two.) The type qualifiers must precede the type names that they qualify.

Const Variables of type **const** may not be changed by your program. (A **const** variable can be given an initial value, however.) The compiler is free to place variables of this type into read-only memory (ROM). For example, const int a=10; creates an integer variable called **a** with an initial value of 10 that your program may not modify. However, you can use the variable **a** in other types of expressions. A **const** variable will receive its value either from an explicit initialization or by some hardware-dependent means. The **const** qualifier can be used to prevent the object pointed to by an argument to a function from being modified by that function. That is, when a pointer is passed to a function, that function can modify the actual object pointed to by the pointer. However, if the pointer is specified as **const** in the parameter declaration, the function code won't be able to modify what it points to. For example, the **sp_to_dash( )** function in the following program prints a dash for each space in its string argument. That is, the string ''this is a test" will be printed as "this-is-a-test". The use of **const** in the parameter declaration ensures that the code inside the function cannot modify the object pointed to by the parameter.

```
#include <stdio.h>
void sp_to_dash(const char *str);
int main(void)
{
sp_to_dash("this is a test");
return 0;
}
void sp_to_dash(const char *str)
{
while(*str) {
if(*str== ' ') printf("%c", '-');
else printf("%c", *str);
str++;
}
}
```

If you had written **sp_to_dash( )** in such a way that the string would be modified, it would not compile. For example, if you had coded **sp_to_dash( )** as follows, you would receive a compiletime error:

```
/* This is wrong. */
void sp_to_dash(const char *str)
{
while(*str) {
if(*str==' ' ) *str = '-'; /* can't do this; str is const */
printf("%c", *str);
str++;
}
}
```

Many functions in the standard library use **const** in their parameter declarations. For example, the **strlen( )** function has this prototype: size_t strlen(const char **str*); Specifying *str* as **const** ensures that **strlen( )** will not modify the string pointed to by *str*. In general,

when a standard library function has no need to modify an object pointed to by a calling argument, it is declared as **const**. You can also use **const** to verify that your program does not modify a variable. Remember, a variable of type **const** can be modified by something outside your program. For example, a hardware device may set its value. However, by declaring a variable as **const**, you can prove that any changes to that variable occur because of external events. ***Volatile*** The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. For example, a global variable's address may be passed to the operating system's clock routine and used to hold the system time. In this situation, the contents of the variable are altered without any explicit assignment statements in the program. This is important because most C compilers automatically optimize certain expressions by assuming that a variable's content is unchanging if it does not occur on the left side of an assignment statement; thus, it might not be reexamined each time it is referenced. Also, some compilers change the order of evaluation of an expression during the compilation process. The **volatile** modifier prevents these changes. You can use **const** and **volatile** together. For example, if 0x30 is assumed to be the value of a port that is changed by external conditions only, the following declaration would prevent any possibility of accidental side effects: const volatile char *port = (const volatile char *) 0x30;

Storage Class Specifiers

C supports four storage class specifiers:

1. extern
2. static
3. register
4. auto

These specifiers tell the compiler how to store the subsequent variable. The general form of a variable declaration that uses one is shown here: *storage_specifier type var_name*;

Extern Before examining **extern**, a brief description of C linkage is in order. C defines three categories of linkage: external, internal, and none. In general, functions and global variables have external linkage. This means they are available to all files that constitute a program. File scope objects declared as **static** (described in the next section) have internal linkage. These are known only within the file in which they are declared. Local variables have no linkage and are therefore known only within their own block. The principal use of **extern** is to specify that an object is declared with external linkage elsewhere in the program. To understand why this is important, it is necessary to understand the difference between a declaration and a definition. A *declaration* declares the name and type of an object. A *definition* causes storage to be allocated for the object. The same object may have many

declarations, but there can be *only one* definition. In most cases, variable declarations are also definitions. However, by preceding a variable name with the **extern** specifier, you can declare a variable without defining it. Thus, when you need to refer to a variable that is defined in another

part of your program, you can declare that variable using **extern**. Here is an example that uses **extern**. Notice that the global variables **first** and **last** are declared *after*
main( ).

```
#include <stdio.h>
int main(void)
{
extern int first, last; /* use global vars */
printf("%d %d", first, last);
return 0;
}
/* global definition of first and last */
int first = 10, last = 20;
```

This program outputs **10 20** because the global variables **first** and **last** used by the **printf( )** statement are initialized to these values. Because the **extern** declaration tells the compiler that **first** and **last** are declared elsewhere (in this case, later in the same file), the program can be compiled without error even though **first** and **last** are used prior to their definition.
It is important to understand that the **extern** variable declarations as shown in the preceding program are necessary only because **first** and **last** had not yet been declared prior to their use in **main( )**. Had their declarations occurred prior to **main( )**, there would have been no need for the **extern** statement. Remember, if the compiler finds a variable that has not been declared within the current block, the compiler checks whether it matches any of the variables declared within enclosing blocks. If it does not, the compiler then checks the global variables. If a match is found, the compiler assumes that is the variable being referenced. The **extern** specifier is needed when you want to use a variable that is declared later in the file. As mentioned, **extern** allows you to declare a variable without defining it. However, if you give that variable an initialization, the **extern** declaration becomes a definition. This is important because, as stated earlier, an object can have multiple declarations, but only one definition. An important use of **extern** relates to multiple-file programs. C allows a program to be spread across two or more files, compiled separately, and then linked together. When this is the case, there must be some way of telling all

the files about the global variables required by the program. The best (and most portable) way to do this is to declare all of your global variables in one file and use **extern** declarations in the other, as in Figure 2-1. In File 2, the global variable list was copied from File 1, and the **extern** specifier was added to the declarations. The **extern** specifier tells the compiler that the variable types and names that follow it have been defined elsewhere. In other words, **extern** lets the compiler know what the types and names are for these global variables without **File One File Two**

```
int x, y;
char ch;
int main(void)
{
/* . . . */
}
void func1(void)
{
x = 123;
}
extern int x, y;
extern char ch;
void func22(void)
{
x = y / 10;
}
void func23(void)
{
y = 10;
}
```

Figure 2-1

Using global variables in separately compiled modules actually creating storage for them again. When the linker links the two modules, all references to the external variables are resolved. One last point: In real-world, multiple-file programs, **extern** declarations are normally contained in a header file that is simply included with each source code file. This is both easier and less error prone than manually duplicating **extern** declarations in each file. ***static Variables*** Variables declared as **static** are permanent variables within their own function or file. Unlike global variables, they are not known outside their function or file, but they maintain their values between calls. This feature makes them useful when you write generalized functions and function libraries that other programmers may use. The **static** modifier has different effects upon local variables and global variables.

Assignment Questions

Unit -III

1. List the different storage classes in C and explain each one of them.
2. Explain dynamic memory allocation functions of C with a suitable example.
3. Compare call by value with call by reference and explain using a suitable example.
4. Explain the concept of pointers and arrays.
5. Write a program to find the factorial of a number using functions.
6. Illustrate pointer operators in C.
7. Explain the use of return statement in C with an example.
8. List out the problems with pointers.
9. Discuss the applications of pointer programming.
10. List different categories of functions based on function arguments and return values. Write example program for each category of function.