

CHAPTER 4

POINTER

“The duties of the Pointer were to point out, by calling their names, those in the congregation who should take note of some point made in the sermon.” —H. B. Otis

The pointers are one of the most important and powerful data structure. A pointer is a variable that is used to store an address or location of another variable. We can say that the pointer points to another variable. A pointer declaration consists of a data type, the indirection operator (*) and a variable name. The indirection operator is also called the dereference operator. Do not confuse the indirection operator with the multiplication operator, although they have the same symbol. The indirection operator is a unary operator, whereas multiplication operator is a binary operator.

The pointers are very useful in dynamic memory allocation, and used to modify variable arguments passed to a function.

Pointer Declaration

The general form of pointer declaration is:

```
data-type *pointer-variable-name;
```

The data type specifies the type of data to which the pointer points, that means it can store an address of that data type.

Example:

```
int *p;  
char *q;  
float *r;
```


In the above declaration, p is a pointer to an integer; it can store an address of integer type variables. The q is a pointer to a character, it can store an address of character type variables (it does not mean that q holds the character value rather it means that q contain the address of a character variable). Similarly r is a pointer to a floating-point, it can store an address of floating-point type variables. Remember that address is always an unsigned integer.


Address of Operator

The address-of operator (&) can return an address in memory of its operand. Do not confuse the address-of operator with the bitwise AND operator, although they have the same symbol. The address-of operator is a unary operator, whereas bitwise AND operator is a binary operator.


In the following example, display the addresses of the variables c, a and f along with their values. The addresses of the variables are dependent on compiler and operating system.


KEY FEATURES


 Null Pointer


 Void pointer


 Generic Function


 Dangling Pointer

 Pointer to Pointer

 Array of Pointer

 Pointer to an Array

 Pointer to Function

 Dynamic memory allocation

Example:

```
#include<stdio.h>
main()
{
char c = 'C';
int a = 50;
float f = 3.45;
printf("Address of c = %u, Value of c = %c\n", &c, c);
printf("Address of a = %u, Value of a = %d\n", &a, a);
printf("Address of f = %u, Value of f = %f\n", &f, f);
}
Output:
Address of c = 65529, Value of c = C
Address of a = 65522, Value of a = 50
Address of f = 65518, Value of f = 3.450000
```

Address	Value	Variable name
	:	
65529	67	c
65522	50	a
65518	3.45	f
	:	

Memory

Figure 4.1: Variables in memory

We can store the address of a variable in a pointer variable. It is also possible to store the address of a variable into an unsigned integer variable. That is useless, you cannot use the indirection operator as a prefix of a variable except a pointer variable.

Example:

```
int *p;
int a = 50;
p = &a;
```

Suppose, the variable a is stored at memory location 65522 and size of it four bytes. That means variable a is stored from memory location 65522 to 65525. Since its size four bytes, four memory locations are required to store the value.

In the above example, the memory location of the variable a stored into variable p. The variable p is an integer pointer that means it can store an address of integer type variables. Variable a has a value of 50. Then, after the assignment statement, pointer p will have the value 65522, which is the starting memory location of variable a and we can say pointer p points to the variable a.

Indirection Operator

The indirection operator (*) returns the value of the memory address which is stored in its operand. It

is essentially the opposite of the address of operator. This operator is also known as pointer operator.

Note: Address of operator (&) returns the address of a variable and indirection operator (*) return the value of that address.

Example:

```
int a, b, *p;  
a = 50;  
p = &a;  
b = *p;
```

Suppose, the variable a is at memory location 65522 and a has a value of 50. Now, the memory location of variable a is assigned to the pointer variable p, hence p will have 65522. The assignment statement sets the value of the memory location 65522, which is stored into p. Now b has the value 50 because 50 are stored at location 65522. The indirection operator and address-of operators are the complements of each other.

Example:

```
#include<stdio.h>
main()
{
int a = 50, b;
int *p;
p = &a;
b = *p;
printf("Value of a = %d\n", a);
printf("Address of a = %u\n", &a);
printf("Value of p = %u\n", p);
printf("Value of *p = %d\n", *p);
printf("Value of b = %d\n", b);
printf("Address of b = %u\n", &b);
printf("Address of p = %u\n", &p);
}
```

Output:

```
Value of a = 50
Address of a = 65524
Value of p = 65524
Value of *p = 50
Value of b = 50
Address of b = 65522
Address of p = 65520
```

We can get the value of b in the following manner,

b = *p that means b = *(65524) [since p = 65524]
= 50 [value of address 65524]

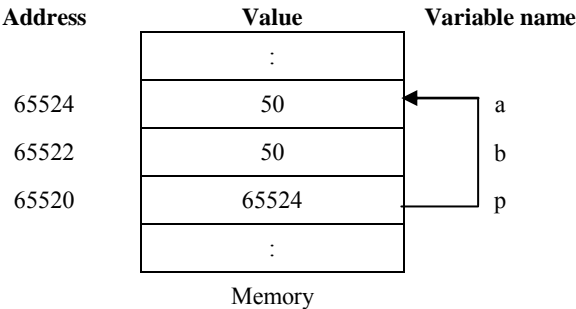


Figure 4.2: Variables in memory

Type casting of Pointers

The pointer variables have their own data type. Unlike basic data types, the pointers do not support implicit type conversion. Therefore, pointer types can be converted to other pointer types using the explicit type casting mechanism.

Example:

```
int a = 10;
float b = 12.3;
int *p;
float *q;
p = &a;
q = &b;
q = p;           /* this is invalid */
p = q;           /* this is invalid */
q = (float*)p;   /* this is valid */
p = (int*)q;     /* this is valid */
```

Note: If the compiler encountered some conversion of a pointer that caused the pointer to point to a different type then the compiler gives an error message like “*Suspicious pointer conversion*”.

Null Pointer

A null pointer is a pointer value that points to no valid location. A null pointer is a constant pointer (often represented by address zero) that is compatible with any pointer. It is not compatible with function pointers. When a pointer is equivalent to NULL it is guaranteed not to point to any variable defined within the program.

A null pointer value is an address that is different from any valid pointer. Assigning the integer constant 0 to a pointer assigns a null pointer value to it. The mnemonic NULL (defined in the standard library header file, `stdio.h`) can be used for legibility.

Example:

```
int *p;
*p = 12;
```

Since `p` is not defined, therefore `p` may contain a garbage value (say, `p = 65550`). Now purposely or accidentally when a value assigning to `*p` then there are chances that modify that memory location

(e.g. 65550) which is not allocated by the program. Therefore, you should use

```
int *p = NULL;
```

Dereferencing a null pointer is meaningless, typically resulting in a run-time error. Therefore, before the use of pointer we should check it with NULL value. All pointers can be successfully tested for equality or inequality to NULL which is logically equivalent to false.

In dynamic memory allocation, when `calloc()` or `malloc()` function fails to allocate memory block then they return NULL. Therefore, after the function call, it requires checking whether the memory block is allocated or not, before the use of the memory.

Example:

```
int *p;
p = malloc(10 * sizeof(int));
if (p==NULL)
{
    printf("Insufficient Memory");
    exit(1);
}
```

The null pointer also indicates the failure of a search operation, such as in the linked-list programs.

Void Pointer

A void pointer is a pointer, which may store the address of any type of variable. That means void pointer is a pointer to anything. The void pointer is also known as a type-less pointer or generic pointer.

Note that a variable of type void cannot be declared. However, the return type of a function may be void.

```
void a;          /* this is invalid */
void *r;         /* valid */
```

The void pointers are used to store the address of any type of variable temporarily. Since the void pointer is a typeless pointer, the compiler has no information that how many bytes of data it will retrieve from the memory starting from stored address. Therefore, the indirection operator cannot be used with a void pointer.

Example:

```
int a = 10;
float b = 12.3;
int *p;
float *q;
void *r;
p = &a;
q = &b;
printf("%d", *p);
r = p;
/* this is invalid */
printf("%d", *r);
```

A void pointer may be assigned to any non-void pointer without explicit type casting operator. Non-void pointer may also be assigned to void pointer without explicit type casting.

Example:

```

r = p;
p = r;
or
r = q;
q = r;

```

where p, q, and r are declared as in the previous example.

The void pointer may be used as a lvalue or as a rvalue. Therefore, by using this concept, void pointer allows violating the basic rule of the type conversion of the pointer.

Example:

```

r = p;
q = r;

```

where p is an integer pointer and q is a floating point pointer. This implies,

```

q = p;

```

Generic Functions

The void pointer is used to write generic functions, which can accept any type of parameter. A group of functions that look the same, except the types of one or more of their arguments. A generic function allows defining a function to replace that group of functions.

Suppose we have a function that can be used to interchange the value of two integer type variables. Now we require another function to interchange the value of two floating-point type variables. For interchange the value of two long double type variables, we need one more function. Therefore, for the different data type, we require to write different function.

Example:

```

/* Interchange the value of two variables */
#include<stdio.h>
void swapi(int *a, int *b);
void swapf(float *a, float *b);
main()
{
    int i = 10, j = 20;
    float x = 12.3, y = 53.4;
    swapi(&i, &j);
    printf("i = %d, j = %d", i, j);
    swapf(&x, &y);
    printf("x = %f, y = %f", i, j);
}
/* Function to interchange two integer type variables*/
void swapi(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

```

```

}
/* Function to interchange two floating-point type variables*/
void swapf(float *a, float *b)
{
float temp;
temp = *a;
*a = *b;
*b = temp;
}

```

Instead of a group of similar type functions, we can use a generic function. A generic function permits to define a function to replace that group of functions.

Example:

```

/*Interchange the value of two variable using generic function */
#include<stdio.h>
void swap(void *a, void *b, int n);
main()
{
int i = 10, j = 20;
float x = 12.5, y = 53.5;
swap(&i, &j, sizeof(int));
printf("i = %d, j = %d\n", i, j);
swap(&x, &y, sizeof(float));
printf("x = %f, y = %f\n", x, y);
}
/* Generic function to interchange two variables */
void swap(void *a, void *b, int n)
{
char *p, *q, temp;
p = (char*)a;
q = (char*)b;
while(n>0)
{
temp = *p;
*p = *q;
*q = temp;
p++;
q++;
n--;
}
}
Output:
i = 20, j = 10
x = 53.500000, y = 12.500000

```

Note that a character is always a byte and a character pointer works as a byte pointer. The sizeof operator calculates the size of the data type in bytes. The generic function sequentially interchanges

each byte of the two variables, since every type of variable can be treated as sequence of character.

Dangling Pointer

Dangling pointers are pointers that do not point to a valid variable of the proper type. Dangling pointers are created when memory is deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. When the system reallocates the previously freed memory to another process and the original program dereferences the dangling pointer, then it may produce an unpredictable result, as the memory may now contain completely different data.

Example:

```
{
    int *cp;
    {
        int c;
        cp = &c;
    }
/* c is in out of scope, cp is now a dangling pointer */
}
```

A solution to the above is to assign NULL to cp immediately before the inner block is exited. Another frequent source of dangling pointers is a combination of malloc () and free () library function calls: a pointer becomes dangling when the block of memory it points to is freed. As with the previous example, one way to avoid this is to make sure to reset the pointer to null after freeing its reference.

Example:

```
#include <stdlib.h>
{
    int *cp = malloc (100);
    free ( cp );    /* cp now becomes a dangling pointer */
    cp = NULL;      /* cp is no longer dangling */
}
```

Allowable Operations with Pointer

There are only a few numbers of operations of pointer are allowed. The allowable operations of pointer as follows:

- Increment and decrement operations with pointer variables.
- Subtraction of two pointer variables.
- Addition and subtraction of integer value with pointers.
- Relational operations between two pointer variables.
- Assignment Operation.

Note: Addition of two pointer variables of any type is not accepted.

Arithmetic Operation with Pointer

Addition and subtraction of integer value with pointer variable are permitted. However, other arithmetic operations, including multiplication, division are not permitted with pointers. Non-integer values such as floats or double value addition or subtraction with pointer variables are not accepted.

The format for adding or subtracting an integer to a pointer is:

Pointer-variable + integer-value

Pointer-variable - integer-value

One pointer may be subtracted from another pointer of the same type and the result will be an integer, not a pointer.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i, *p, *q;
p = a + 1;
q = a + 3;
printf("%d", q - p);
Output:
2
```

Increment and Decrement operations with Pointer

Increment operation can be used with pointer variable that is similar to add one with pointer variable. Decrement operation can also be used with pointer variable that is similar to subtract one from pointer variable. The formats for incrementing or decrementing a pointer are:

Pointer-variable++

Pointer-variable--

There is three possible mix pointer increment and indirection are as follows:

i) *p++

ii) *++p

iii) ++*p

Where p is a pointer variable.

i) *p++ returns the content at the location being pointed by p and then increment the pointer by one. The pointer p will point the next element. The decrement operator can be used with pointers, in the same manner, to move to the previous element. Note that the compiler may give some warning or error message if next memory location is not allocated by the program statically or dynamically.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
p = a;
printf("%d ", *p++);
printf("%d ", p - a);
printf("%d ", *p);
Output:
10 1 20
```

It should be noted that increment (++) and decrement (--) operators have a higher precedence than the precedence of indirection operator (*).

ii) *++p increments the pointer by one and then it returns the content at the location being pointed by p. The pointer p will point the next element. The decrement operator can be used with pointers, in the same manner, to move to the previous element. Note that the compiler may give some warning or

error message if the next memory location is not allocated by the program statically or dynamically.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
p = a;
printf("%d ", *++p);
printf("%d ", p - a);
printf("%d ", *p);
```

Output:

20 1 20

ii) ++*p increments the value at the location being pointed by p. The pointer p will be points same element. The decrement operator can be used with pointers in the same manner to decrement the value at the location being pointed.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
p = a;
printf("%d ", ++*p);
printf("%d ", p - a);
printf("%d ", *p);
```

Output:

11 0 11

Relational Operation with Pointer

A pointer variable can be compared with another pointer variable of the same type using different relational operators. Table 4.1 describes allowable relational operations with pointer.

Table 4.1: Relational operators with pointer

Operators	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

A pointer variable can also be compared with a NULL pointer.

The Table 4.2 shows pointer notations and array notations. The notations are based on the following code:

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr;
ptr = arr;
```

Table 4.2: Pointer notation and array notation

Pointer Notation	Array Notation	Results
<code>ptr</code>	<code>arr, &arr[0]</code>	Address of the first element of the array.
<code>ptr++</code>	<code>&arr[x++]</code>	Move the pointer to the next element of the array.
<code>ptr + 1</code>	<code>&arr[1]</code>	Address of the second element of the array.
<code>*ptr</code>	<code>arr[0]</code>	Value of the first element of the array.
<code>*(ptr + 1)</code>	<code>arr[1]</code>	Value of the second element of the array.
<code>*(ptr) + 1</code>	<code>arr[0] + 1</code>	Add 1 and the first element of the array together.
<code>*ptr++</code>	<code>arr[x++]</code>	Move to the next element in the array after the value is used.
<code>*ptr--</code>	<code>arr[x--]</code>	Move to the previous element in the array after the value is used.
<code>++ ptr</code>	<code>arr[++x]</code>	Move to the next element in the array before using the value.
<code>-- ptr</code>	<code>arr[--x]</code>	Move to the previous element in the array before using the value.
<code>(* ptr)++</code>	<code>arr[0]++</code>	Increment the first element of the array by 1.
<code>(* ptr)--</code>	<code>arr[0]--</code>	Decrement the first element of the array by 1.
<code>*(ptr + 2)++</code>	<code>arr[2]++</code>	Increment the third element of the array by 1.
<code>*(ptr + 2)--</code>	<code>arr[2]--</code>	Decrement the third element of the array by 1.

Implicit Scaling in Pointer Addition

When several data items of the same type are placed consecutively in the memory, then a unit increment or decrement to a pointer to any of the data items, always gives the address of the next or previous items. This works independently of the data item.

Example:

```
int *p;
float *q;
```

Now, when we write `p + i` where `i` is an integer number, then the system evaluates as

`p + i * sizeof(int)`.

Similarly, `q + i` systematically evaluate as `q + i * sizeof(float)`.

Therefore, we always get the address of i^{th} element independently of the data item.

Pointer to Pointer

A pointer is a variable that is used to store an address or location of another variable. That variable may also be a pointer. The pointer to pointer is a special type of pointer that is used to store an address of another pointer variable.

Example:

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

```
int a = 50, b, c;
int *p;
int **q;
p = &a;
b = *p;
printf("Value of a = %d\n", a);
printf("Address of a = %u\n", &a);
printf("Value of p = %u\n", p);
printf("Value of *p = %d\n", *p);
printf("Value of b = %d\n", b);
q = &p;
c = **q;
printf("Value of p = %u\n", p);
printf("Address of p = %u\n", &p);
printf("Value of q = %u\n", q);
printf("Value of **q = %u\n", **q);
printf("Value of c = %d\n", c);
printf("Address of q = %u\n", &q);
}
```

Output:

```
Value of a = 50
Address of a = 65524
Value of p = 65524
Value of *p = 50
Value of b = 50
Value of p = 65524
Address of p = 65518
Value of q = 65518
Value of **q = 50
Value of c = 50
Address of q = 65516
```

	Value	Variable name
	:	
65524	50	a
65518	65524	p
65516	65518	q
	:	
Memory		

Figure 4.3: Variables in memory

Pointers and Arrays

In the C language, pointers and arrays are closely related. The name of an array is also the address of the first element of the array and points to the location in memory of the first element in the array.

Arrays are nothing but pointers. Hence, there is no index overflow or underflow checking. Arrays are also called as a static pointer. Since the name of an array is a constant pointer.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i;
a[5] = 5;           /* overflow */
a[-5] = -5;         /* underflow */
```

No index overflow or underflow error message will be given by the compiler and index overflow or underflow checking does not take place at runtime.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
a = p;              /* this is invalid */
```

Here, the pointer variable `p` is assigned to the base address of the array. This is not accepted because the base address of the array is a constant and a pointer is the variable.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i, *p;
p = a;
for(i=0; i<5;i++)
    printf("%d", p[i]);
```

The base address of the array is assigned to the pointer variable `p`. The pointer variable `p` can be used just like as it is an array name and the `printf` statement outputs the value of the each element of the array.

An alternative way doing the same thing with the following statement that uses the address of the first element of the array is assigned to the pointer variable `p`.

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

An array element `a[i]` can be written as `*(a+i)` in pointer representation and `a+i` signify the address of that element.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i;
for(i=0; i<5;i++)
    printf("%d", *(a+i));
```

A two-dimensional array can be thought as a one-dimensional array and that may be represented by a pointer.

Example:

```
int a[3][4] = {{11, 12, 13, 14},{21, 22, 23, 24},{31, 32, 33, 34}};
```

```

int *p;
int i, j;
p = a[0];
for(i=0; i<12; i++)
    printf("%5d", p[i]);

```

The base address of the array `a[0]` is assigned to the pointer variable `p`. The pointer variable `p` can be used just like as it is a one-dimensional array and `printf` statement outputs the value of the each element of the array. Figure 4.4 shows how two-dimensional array stores in memory.

4000	4002	4004	4006	4008	4010	4012	4014	4016	4018	4020	4022
11	12	13	14	21	22	23	24	31	32	33	34

Figure 4.4: Two-dimensional array stores in memory, such as one-dimensional array

Suppose a two-dimensional array is declared as

```
int a[3][4];
```

An array element `a[i][j]` of the two dimensional array can be accessed through a pointer by following different syntaxes:

```

a[i][j] = (*(a+i)+j)
a[i][j] = *(a[i]+j)
a[i][j] = (*(a+i))[j]
a[i][j] = *((*a)+(i*4+j))

```

Both addition operations are following the rules of pointer arithmetic, but pointers to different types are involved. The inner arithmetic operation involves with a pointer to an array of size 4. That is why we need to specify the number of columns of the two-dimensional array in function definition when it is used as a formal argument. The outer arithmetic operation involves with a pointer to an integer.

Example:

```

#include<stdio.h>
main()
{
    int a[5]={3, 5, 6, 8};
    printf("%d %d ", a[3], 3[a]);
}
Output:
8 8

```

Do not get surprised! It is a correct syntax:

`3[a]` equals to `*(3+a)`

Array variable always returns the base address, `[]` does sum with array index (i.e. `*(base + index)`) and addition (+) is commutative. Therefore, `*(base + index)` is equal to `*(index + base)`.

Since `3[a]` has represented `*(3+a)` and `*(3+a)` is equal to `*(a+3)`. Therefore, `a[3]` is same as `3[a]` since `a[3]` represents `*(a+3)`.

But on the other hand `[a]3` or `[3]a` is not correct syntax and will result into syntax error, since `(a + 3)*` and `(3 + a)*` are not valid expressions.

Array of Pointers

An array of pointers is nothing more than an array of elements that contain the addresses of values in memory. The format for creating an array of pointers is similar to the format for creating any other array:

```
data_type *array_name[size];
```

The array type should match the data type it points to; the indirection operator precedes the name of the array; and the number of elements (pointers) is enclosed in the braces. To assign an address to an element in an array of pointers, use the address operator, as in the following example:

Example:

```
#include<stdio.h>
main()
{
int x,y,z;
int *a[3];
int **p;
x = 10;
y = 20;
z = 30;
a[0] = &x;
a[1] = &y;
a[2] = &z;
printf("x = %d\n", *a[0]);
printf("y = %d\n", *a[1]);
printf("z = %d\n", *a[2]);
p = a;
printf("x = %d\n", *p[0]);
printf("y = %d\n", *p[1]);
printf("z = %d\n", *p[2]);
}
Output:
x = 10
y = 20
z = 30
x = 10
y = 20
z = 30
```

The address of each integer variable x, y and z is assigned to an element in the array of pointers. The format `*a[0]` indirectly refers to the value of x by using the address stored in `a[0]`.

The base address of the array is assigned to the pointer variable p. The pointer variable p can be used just like as it is an array name and the printf statement outputs the value of the each element of the array.

Pointer to an Array

A two-dimensional array name is a composite pointer that means it is a pointer to arrays, each of the same size as the number of columns. A one-dimensional array can be represented by an elementary pointer variable and a two-dimensional array can be represented by a composite pointer variable that is

a pointer to an array.

The format for creating a pointer to an array:

```
data_type (*pointer_name)[size];
```

where the `data_type` should match the data type it points to; the size within bracket equals to the number of columns and `pointer_name` is the name of the pointer.

The objective of declaring a pointer to an array is to ensure that the unit increment of the corresponding pointer always takes from beginning of one row to the beginning of next row, independent of the data type. Remember that pointer to an array is different from the array of pointer.

Example:

```
int a[3][4] = {{11, 12, 13, 14},
               {21, 22, 23, 24},
               {31, 32, 33, 34}};

int (*p)[4];      /* pointer to an array of size 4 */
int i, j;
p = a;
for(i=0;i<3;i++)
{
    for(j=0;j<4;j++)
        printf("%5d", p[i][j]);
    printf("\n");
}
```

Here, the pointer to an array of size 4 is declared that should be equal to the column size of the two-dimensional array. The base address of the array is assigned to the pointer variable `p`. The pointer variable `p` can be used just like as it is an array name and the `printf` statement outputs the value of the each element of the array.

Pointer to Function

Pointers can point to integer, character, array, pointer as well as pointers can also point to the C function. The functions are loaded into computer memory before they are invoked by the program. Therefore, they have also addresses, from where they are loaded into the memory. When a function address is known then a pointer can point to it. The pointer can provide another way to invoke the function. Note that, the function name itself is an address of the function.

The format for creating a pointer to a function:

```
data_type (*pointer_name)();
```

where `data_type` same as the return type of the function and `pointer_name` is the name of the pointer that points to a function.

Example:

```
#include<stdio.h>
int show();
main()
{
    int (*p)();
    p = show;
    printf("Address of Function show is %u\n", show);
}
```



```

/* Function show is called using pointer */
(*p) ();
}
int show()
{
printf("Hello India!");
return 0;
}

```

Output:

```

Address of Function show is 4198701
Hello India!

```

Pointers can point to user-defined functions as well as they can point to library functions.

The uses of the pointer to the function are as follows:

- To write memory resident programs.
- To write viruses (i.e. Worms, Trojan horses, etc.) and vaccines (i.e. anti-virus) to remove viruses.

Passing Addresses to Function

A function can pass a value, as well as it can pass addresses to the function i.e. can pass a reference. The addresses of actual arguments in the calling function are copied to formal arguments of the called function. The addresses of the actual arguments are copied to formal arguments as a value, not a reference. In C language functions are called by value and C does not support reference data types. That means the called function uses the values of its arguments as temporary variables rather than the originals. When the temporary variables are modified in the called function, then it has no effect on the actual arguments in the calling function.

It is possible to create a function that can modify a variable in the calling function. Generally, a function can return only one value to the calling function. However, a function can return more than one value at a time to the calling function by passing addresses. The calling function passes the addresses of the variables and the called function must declare the arguments to be a pointer and access the variable indirectly through it.

Example:

```

#include<stdio.h>
void swap(int *a, int *b);
main()
{
int i = 10, j = 20;
printf("Before calling function i = %d, j = %d\n", i, j);
swap(&i, &j);
printf("After calling function i = %d, j = %d\n", i, j);
}
void swap(int *a, int *b)
{
int temp;
temp = *a;

```

```
*a = *b;
*b = temp;
}
```

Output:

Before calling function i = 10, j = 20

After calling function i = 20, j = 10

Function Returning Pointer

The functions can return an integer, a floating point or any other data type; as well as it can also return a pointer. To create a function returning a pointer should explicitly mention in the function prototype and in the function definition.

The format of function returning pointer:

data-type *function-name(argument-list)

Example:

```
#include<stdio.h>
#include<stdlib.h>
char *xstrcat(char*,char*);
main()
{
    char *dest, *source = "India!", *target = "Hello ";
    dest=xstrcat(target, source);
    printf("source = %s\n",source);
    printf("target = %s\n",target);
    printf("dest = %s\n",dest);
}
char *xstrcat(char* t,char* s)
{
    int i;
    char *temp;
    temp=(char*)malloc(25);
    for(i=0;* (t+i)!='\0';i++)
        *(temp+i)=*(t+i);
    while(*s)
    {
        *(temp+i)=*s;
        s++;
        i++;
    }
    return(temp);
}
```

Output:

source = India!

target = Hello

dest = Hello India!

Dynamic Memory Allocation

When memory has been needed by a program, it has been set aside by declaring the desired type of variables. For variables declared in any function, space in memory is set aside to hold the value assigned to the variable. This memory is reserved until the function finishes.

Another problem may arise when declared an array in a program. The user may provide a less number of elements than the number of elements of the declared array, and then the rest of the memory space will be wasted. This leads to the inefficient use of memory. As well as the user may also enter a number of elements more than the number of elements of the declared array, then the user may get unexpected result or user may face a runtime error or system may be crashed.

This may not always be the optimal way to allocate memory. Fortunately, you can instead write your programs and obtain memory as they are running. With dynamic memory allocation, memory is not reserved or set aside at the start of the program; rather, it is allocated on an as-needed basis.

When a program is compiled, many of the memory locations needed for the program to hold variable and constant data can be determined in advance. As the compiler works through the main part of the program and each of the other functions, it can figure out what memory will be needed when the program runs. The memory can be divided into four regions as shown in the figure.

When the program is loaded, it can request the needed memory from the operating system before the program actually begins to run. The operating systems reserve the needed memory locations by stacking one variable on top of another in memory, in a tight, neat block. Because of the way this process works, this part of the memory is known as the stack. Memory reserved within the stack cannot be freed up until the program quits running.

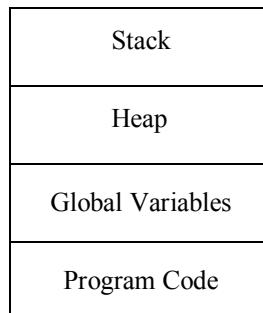


Figure 4.5: Four regions of memory

Some programs need to use a large block of memory to hold data, but they only need those blocks for a short period of time. Rather than use all that memory the entire time the program is running, such programs can temporarily allocate storage locations from another portion of memory, known as the heap. When the program is done using a particular block of heap memory, it simply tells the operating system that it is done, and the system returns that memory to the heap, where it can be freed out to other needy programs. For best utilization of memory, clearly having a heap is a good idea.

A compiled C program creates and uses four logically distinct regions of memory. The first region is the memory that actually holds the program's executable code. The next region is a memory where global variables are stored. The remaining two regions are the stack and the heap. The stack is used for a great many things while your program executes. It holds the return addresses of function calls, arguments to functions, and local variables. It will also save the current state of the CPU. The heap is a region of free memory that your program can use via C's dynamic memory allocation functions.

Although the exact physical layout of each of the four regions of memory differs among CPU types and C implementations, the diagram in Figure shows conceptually how your C programs appear in memory. The important functions involved in dynamic memory allocation are `malloc()`, `calloc()`, `realloc()` and `free()`.

Table 4.3: Functions in dynamic memory allocation

Functions	Description
<code>malloc()</code>	The <code>malloc()</code> function allocates a block of size bytes from the memory heap.
<code>calloc</code>	<code>calloc()</code> allocates a block of size bytes from the memory heap and block is initialized to zero.
<code>realloc()</code>	<code>realloc()</code> function reallocates the memory block that means it attempts to reduce or increase the previously allocated block.
<code>free()</code>	<code>free()</code> function is used to de-allocate a memory block allocated by a previous call to <code>calloc()</code> , <code>malloc()</code> , or <code>realloc()</code> .

The malloc Function

The `malloc()` function is a library function that uses to allocate a block of memory (size in bytes) from the memory heap. This function allows a program to allocate memory dynamically (i.e. in run time) when it is needed and in the exact amounts. The contents of the block are left unchanged.

The general form of the `malloc` function is

```
void *malloc(size)
```

The `malloc()` function returns a void pointer to the newly allocated block of memory. When there is not enough space exists for the new block or if the size argument is given zero then `malloc` function returns `NULL`.

Example:

```
int *p;
p = (int*) malloc(10 * sizeof(int));
```

In the example, the `malloc()` function, allocate a memory block for 10 integer elements at runtime. After the memory allocation, it returns a pointer to the memory block. It is possible to guess how much memory is required. However, because the size of an integer varies from system to system, therefore problems can arise when run the code to another platform. The `sizeof` operator is used to avoid problems. The `sizeof` operator is used to evaluate the size of each integer element in bytes. The multiplication operator calculates the total size of the memory block. The explicit type conversion is optional.

The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ a heap memory allocation.

The calloc Function

The `calloc()` function is used to allocate a block of memory (size in bytes) from the memory heap. This function allows a program to allocate memory dynamically (i.e. at run time) when it is needed and in the exact amounts. The block is cleared to zero

The general form of the `calloc()` function is

```
void *calloc(nitems, size)
```

Where `nitems` is the number of item and `size` represents the size of each item in bytes. The

calloc() function allocates a block of memory of size nitems * size.

Example:

```
int *p;
p = (int*) calloc(10, sizeof(int));
```

In the example, the calloc() function allocates a memory block for 10 integer elements at runtime. After the memory allocation, it returns a pointer to the memory block and the block is initialized to zero. The sizeof operator is used to evaluate the size of each integer element in bytes. The explicit type conversion is optional.

The calloc() function returns a void pointer to the newly allocated block of memory. When there is not enough space exists for the new block or if the argument size or nitem is zero then the calloc function returns NULL.

The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ a heap memory allocation.

The realloc Function

The realloc() function reallocates the memory block that means it attempts to reduce or increase the previously allocated block.

The general form of the calloc() function is

```
void *realloc(void *p , size)
```

Here, p is a pointer to block that is previously allocated and size is representing the size of the memory block.

If size is zero, the memory block is freed and NULL is returned. The block argument points to a memory block previously obtained by calling malloc, calloc() or realloc(). If the block is a NULL pointer, the realloc () function works just like malloc. The realloc() function adjusts the size of the allocated block to size, copying the contents to a new location if necessary.

The realloc() function returns the address of the reallocated block, which can be different than the address of the original block. When the block cannot be reallocated then the realloc() function returns NULL. If the value of size is zero, the memory block is freed and realloc() returns NULL.

The free Function

The free() function is used to deallocate a memory block allocated by a previous call to calloc(), malloc(), or realloc(). The general form of the calloc() function is

```
void free(void *p);
```

Here, p is a pointer to block that is previously allocated. It returns nothing.

Example:

```
int *p;
p = malloc(10 * sizeof(int));
free(p);
```

The free() function releases the memory block, which is previously allocated by malloc() function.

Example:

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

```

main()
{
    char *s;
    /* allocate memory for string */
    s = (char *) malloc(10);
    if (s == NULL)
    {
        printf("Not enough memory to allocate memory");
        exit(1); /* terminate program if out of memory */
    }
    /* copy "Hello" into string */
    strcpy(s, "Hello");
    /* display string */
    printf("String is %s", s);
    /* free memory */
    free(s);
}

```

Creating One-dimensional Array

Sometimes we can use a pointer variable instead of an array. Suppose, a is declared as a one-dimensional array of integer and p is declared as a pointer to integer.

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

we can write instead of the array

```
int *p;
```

but p is not automatically allocated a block of a memory, therefore before the use of the pointer variable in the place of a array we need to allocate a block of memory for it.

The one-dimensional array can be created by using malloc () or calloc () function.

Example:

```

#include<stdio.h>
#include<stdlib.h>
main()
{
    int *p, n, i;
    printf("Enter the number of elements");
    scanf("%d", &n);
    p = (int*)malloc(n * sizeof(int));
    if(p==NULL)
    {
        printf("Insufficient Memory");
        exit(1);
    }
    printf("Enter the elements of the array");
    for(i=0;i<n;i++)
        scanf("%d", p+i);
    printf("Elements of the array");
}

```

```
for(i=0;i<n;i++)
    printf("%5d",*(p+i));
}
```

Output:

```
Enter the number of elements 5
Enter the elements of the array 3 5 6 7 9
Elements of the array      3      5      6      7      9
```

Creating Two-dimensional Array

There are two methods of creating two-dimensional arrays dynamically.

In the first method, the number of columns of the two-dimensional array has been known before compile time, but the number of rows will be known only at the execution time. The two-dimensional array is allocated dynamically which is represented by using a pointer to array having the same size of a number of columns of the two-dimensional array.

Example:

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int (*p)[4], m, i, j;
    printf("Enter number of rows: ");
    scanf("%d", &m);
    p = (int (*)[4])malloc(m*4*sizeof(int));
    if(p==NULL)
    {
        printf("Insufficient Memory");
        exit(1);
    }
    printf("Enter the elements of the array\n");
    for(i=0;i<m;i++)
        for(j=0;j<4;j++)
            scanf("%d", &p[i][j]);
    printf("Elements of the array\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<4;j++)
            printf("%5d",p[i][j]);
        printf("\n");
    }
}
```

Output:

```
Enter numner of rows: 3
Enter the elements of the array
11 12 13 14
```

```

21 22 23 24
31 32 33 34
Elements of the array
  11  12  13  14
  21  22  23  24
  31  32  33  34

```

Note that p may now be used as an array of size 4.

In the second method, the numbers of rows, as well as a number of columns both, are unknown at compile time they will be known at the execution time. The following steps are used to create a two-dimensional array dynamically:

- i) At first, an array of pointers is to be created dynamically. This array of pointers will have the same size as the number of rows of the two-dimensional array. The i^{th} array element should point to the beginning of the i^{th} row.
- ii) Allocate memory space for the entire array dynamically. The total space may be contiguous or may not be so. However, the minimum condition is that all the elements of each row must be allocated space contiguously.
- iii) The value of each element of the pointer array that created in the first step is to be properly initialized.

Contiguous Allocation of Two-dimensional Array

In the contiguous allocation of memory space for the two-dimensional array, the whole array is allocated dynamically. The total memory space of two-dimensional array should be contiguous. An array of pointers is also to be created dynamically. This array of pointers will have the same size as the number of rows of the two-dimensional array and the i^{th} array element should point to the beginning of the i^{th} row.

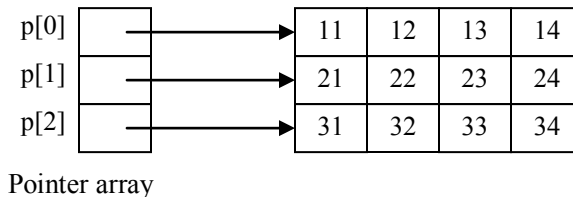


Figure 4.6: Contiguous allocation of entire 2D array

Note that the array of pointers can be represented by pointer to pointer.

Example:

```

#include<stdio.h>
#include<stdlib.h>
main()
{
int **p, *q, m, n, i, j;
printf("Enter number of rows");
scanf("%d", &m);
printf("Enter number of columns");

```



```

scanf("%d", &n);
p =(int**)malloc(m * sizeof(int*));
q =(int*)malloc(m * n * sizeof(int));
if(p==NULL || q==NULL)
{
    printf("Insufficient Memory");
    exit(1);
}
for(i=0; i<m; i++)
{
    p[i] = q;
    q += n;
}
printf("Enter the elements of the array\n");
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        scanf("%d", &p[i][j]);
printf("Elements of the array\n");
for(i=0; i<m; i++)
{
    for(j=0; j<n; j++)
        printf("%5d",p[i][j]);
    printf("\n");
}
}

```

Output:

```

Enter number of rows3
Enter number of columns3
Enter the elements of the array
11 12 13
21 22 23
31 32 33
Elements of the array
  11   12   13
  21   22   23
  31   32   33

```

Non-contiguous Allocation of Two-dimensional Array

Non-contiguous allocation of two-dimensional array is used when we need to create a large two-dimensional array (for example 1000 x 1000) and/or contiguous memory may not be available. For non-contiguous allocation of memory, space for each row is to be allocated separately as the need arises. Therefore, the total space is not being contiguous. However, all the elements of each row must be allocated space contiguously.

An array of pointers is created dynamically. This array of pointers will have the same size as the number of rows of the two dimensional array and the i^{th} array element should point to the beginning of the i^{th} row.

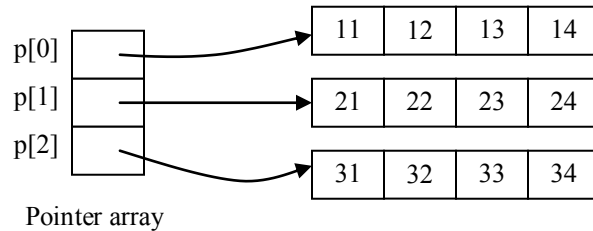


Figure 4.7: Non-contiguous allocation of 2D array, but each row must be allocated space contiguously

Example:

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int **p, m, n, i, j;
    printf("Enter number of rows");
    scanf("%d", &m);
    printf("Enter number of columns");
    scanf("%d", &n);
    p = (int**) malloc(m * sizeof(int*));
    for(i=0; i<m; i++)
        p[i] = (int*)malloc(n * sizeof(int));
    printf("Enter the elements of the array");
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            scanf("%d", &p[i][j]);
    printf("Elements of the array");
    for(i=0; i<m; i++){
        for(j=0; j<n; j++)
            printf("%5d", p[i][j]);
        printf("\n");
    }
}
```

Output:

```
Enter number of rows3
Enter number of columns3
Enter the elements of the array
11 12 13
21 22 23
31 32 33
Elements of the array
    11    12    13
    21    22    23
    31    32    33
```

However, the drawbacks of dynamic memory allocation for the two-dimensional array are to be followed:

- Extra memory space is required for storage of pointer array.
- Access to the individual array element is obtained only after the first level of indirect addressing.

Pointers, Arrays and Strings

Sometimes we can use a pointer variable instead of an array. Suppose, a is declared as a one-dimensional array of integer and p is declared as a pointer to an integer.

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

we can write instead of the array

```
int *p;
```

Since p is not defined so p may contain a garbage value and p is not automatically allocated a block of a memory. Therefore, before the use of the pointer variable in the place of an array we need to allocate a memory for it.

Example:

```
p = (int *)malloc(100*sizeof(int));
```

This statement allocates a memory block for 100 integer values.

Example:

```
char *s;
scanf("%s", s);           /* invalid */
or
char *s;
s = "Hello"               /* invalid */
```

We should require allocating a block of memory before the use of pointer variable. Otherwise, it may display a runtime error like: ~~Segmentation fault~~.

Example:

```
char *s;
s = (char*)malloc(10);
scanf("%s", s);
```

However in the case of array, memory block is allocated at compile time.

```
char t[10];
scanf("%s", t);           /* valid */
```

but we cannot assign a value to a string variable

```
char t[10];
t = "Hello";
```

we should use the library function strcpy() to do this

Example:

```
char t[10];
strcpy(t, "Hello");
```

Program: How to find size of int data type without using sizeof operator.

```
#include<stdio.h>
main()
{
    int *ptr = 0;
    ptr++;
    printf("Size of int data type:  %d",ptr);
}
```

Summary

- A pointer is a variable that can point to another variable.
- That variable may be an integer type, character type or floating-point type.
- A pointer can also point to another pointer variable then the pointer is known as pointer-to-pointer.
- The void pointer is a pointer, which may store the address of any type of variable.
- Sometimes we need to allocate memory space when a program is running. This type of memory allocation is called dynamic memory allocation.
- The malloc() and calloc() functions are used to allocate memory space at runtime.
- The realloc() function is required to modify the previously allocated memory space.
- The free() function is used to release the memory space that allocated previously call of malloc(), calloc() or realloc() function.

Exercises

1. What is a pointer?
2. What is the purpose of indirection operator? How can the indirection operator be used to access a multidimensional array element?
3. What is pointer to pointer?
4. What is a void pointer? What is a generic function?
5. What is a dangling pointer?
6. How do you get access to an element in an array by using a pointer? Explain with suitable example.
7. How pointer to function works?
8. How can a function return a pointer to its calling function? Explain with a suitable example.
9. What is dynamic memory allocation? What is the advantage of dynamic memory allocation over static memory allocation?
10. What are the differences between malloc() and calloc() function?
11. Write a program using pointers to compute the sum of all elements stored in an array.
12. Write a program using pointers to determine the length of a character string.
13. Write a function using pointers to exchange the values stored in two locations in the memory.
14. Write a program to generate an array of N elements dynamically and sort then in ascending order.
15. Write a program of matrix multiplication by using dynamic memory allocation.
16. What is the output of the following C program?


```
(i)      main()
        {
```

```

        const int x = 5;
        int *prt;
        ptr = &x;
        *prt = 10;
        printf("%d", x);
    }
(ii)  main()
    {
        int a=2, *f1, *f2;
        f1 = f2 = &a;
        *f1+=*f2+=a+=2.5;
        printf("%d %d %d", a, *f1, *f2);
    }
(iii) main()
    {
        register int i =5;
        printf("Address of a=%u", &i);
        printf("Value of a=%d", i);
    }
(iv)  main()
    {
        int a=10;
        void *j;
        j=&a;
        j++;
        printf("%u", j);
    }
(v)   main()
    {
        char *p="Hello";
        printf("%c", *p);
    }
(vi)  main()
    {
        int i=320;
        char *ptr=(char *)&i;
        printf("%d", *ptr);
    }
(vii) main()
    {
        static char str[]="Limericks";
        char *s;
        s=&str[6]-6;
        while(*s)
            printf("%c", *s++);
    }

```
