POINTERS

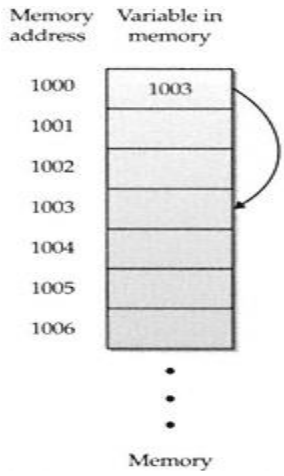
The correct understanding and use of pointers is crucial to successful C programming.
There are several reasons for this:

First, pointers provide the means by which functions can modify their calling arguments.
Second, pointers support dynamic allocation.
Third, pointers can improve the efficiency of certain routines, and
Finally, pointers provide support for dynamic data structures, such as binary trees and linked lists.

Pointers are one of the strongest but also one of the most dangerous features in C.
For example, a pointer containing an invalid value can cause your program to crash.

Define a Pointer

A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory.
For example, if one variable contains the address of another variable, the first variable is said to *point to* the second.
Figure illustrates this situation.



One variable points to another

Pointer Variable Declaration

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.
- Technically, any type of pointer can point anywhere in memory.
- All pointer operations are done relative to the pointer's base type.

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. (That is, an **int *** pointer always ''thinks" that it points to an **int** object, no matter what that piece of memory actually contains.)
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.

Pointer Variable Initialization

Pointer Initialization is the process of assigning address of a variable to **pointer** variable.

Pointer variable contains address of variable of same data type.
In C language **address operator** & is used to determine the address of a variable.
The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ;      //pointer declaration
ptr = &a ;      //pointer initialization
or,
int *ptr = &a ;     //initialization and declaration together
Pointer variable always points to same type of data.
float a;
int *ptr;
ptr = &a;    //ERROR, type mismatch
```

Example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
{
int a;      // Step 1
int *ptr;    // Step 2
a = 10;     // Step 3
ptr = &a;    // Step 4
return(0);
}
```

Explanation of Above Program :

- Pointer should not be used before initialization.
- "ptr" is pointer variable used to store the address of the variable.
- Stores address of the variable **'a'** .
- Now "ptr" will contain the address of the variable "a" .

Note :
Pointers are always initialized before using it in the program

Example : Initializing Integer Pointer

```
#include<stdio.h>
int main()
{
int a = 10;
int *ptr;
ptr = &a;
printf("\nValue of ptr : %u",ptr);
return(0);
}
```

Output :

```
Value of ptr : 4001
```

The Pointer Operators
There are two pointer operators:

* and **&**.

- The **&** is a unary operator that returns the memory address of its operand. (A unary operator only requires one operand.)

  For example,

  m = &count; &-- " address of"

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** ."
- Assume that the variable **count** uses memory location 2000 to store its value and **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; * -- "at address."
  - 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**.
  - 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. 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

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.

<u>Pointer Conversions</u>

- 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.

- 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 ,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.
- 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 */
         printf(''The (incorrect) value of x is: %f", y); /* statement won't output 100.1. */
        return 0;
}
```

- 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.
- 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.
- 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.
- We 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.)

Pointer Arithmetic

There are only two arithmetic operations that you can use on pointers:

addition and subtraction.

To understand what occurs in pointer arithmetic, let **p1** be an integer pointer with a current value of 2000.
Also, assume **ints** are 2 bytes long.
After the expression

```
p1++;
```

p1 contains 2002, not 2001.

The reason for this is that each time **p1** is incremented, it will point to the next integer. The same is true of decrements.
For example, assuming that **p1** has the value 2000, the expression p1--; causes **p1** to have the value 1998.
Generalizing from the preceding example, the following rules govern pointer arithmetic.

- Each time a pointer is incremented, it points to the memory location of the next element of its base type.

- Each time it is decremented, it points to the location of the previous element.
- When applied to **char** pointers, this will appear as ''normal" arithmetic because a **char** object is always 1 byte long no matter what the environment.
- All other pointers will increase or decrease by the length of the data type they point to.
- This approach ensures that a pointer is always pointing to an appropriate element of its base type.

Below figure illustrates this concept.
We may add or subtract integers to or from pointers.
The expression p1 = p1 + 12; makes **p1** point to the 12th element of **p1**'s type beyond the one it currently points to.
Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed:
You can subtract one pointer from another in order to find the number of objects of their base type that separate the two.
All other arithmetic operations are prohibited, such as

- We cannot multiply or divide pointers;
- We cannot add two pointers;
- We cannot apply the bitwise operators to them; and
- We cannot add or subtract type **float** or **double** to or from pointers.

```
char *ch = (char *) 3000;
int *i = (int *) 3000;
```

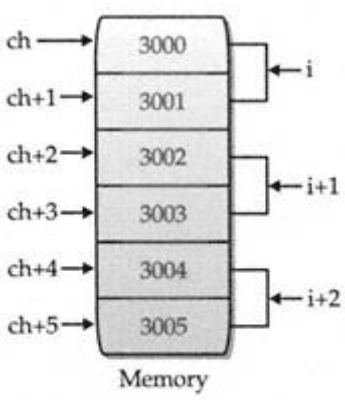


Figure 5-2
All pointer arithmetic is relative to its base type (assume 2-byte integers)

Pointer Comparisons
You can compare two pointers in a relational expression. For instance, given two pointers **p** and **q**, the following statement is perfectly valid:

```
if(p < q) printf("p points to lower memory than q\n");
```

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**, make it as 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**.

- The array name without an index returns the starting address of the array, which is the address of the first element.
- C provides two methods of accessing array elements: pointer arithmetic and array indexing.
- 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++);
        }
```

Arrays of Pointers

- Pointers can be arrayed like any other data type.
- The declaration for an **int** pointer array of size 10 is int *x[10];
- To assign the address of an integer variable called **var** to the third element of the pointer array, write x[2] = &var;
- To find the value of **var**, write *x[2]
- If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays: Simply call the function with the array name without any subscripts.

For example, a function that can receive array **x** looks like this:

```
void display_array(int *q[])
        {
                int t;
                for(t=0; t<10; t++)
                printf(''%d '', *q[t]);
        }
```

- Remember, **q** is not a pointer to integers, but rather a pointer to an array of pointers to integers.
- Therefore you need to declare the parameter **q** as an array of integer pointers, as just shown.
- You cannot declare **q** simply as an integer pointer because that is not what it is.
- Pointer arrays are often used to hold pointers to strings.

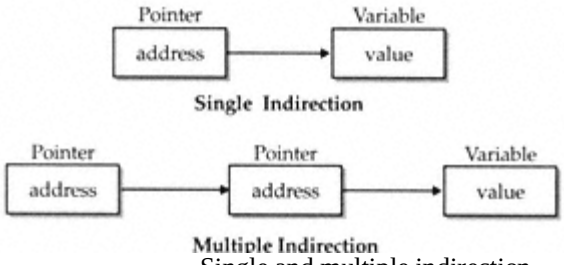
Multiple Indirection

- When a pointer point to another pointer that points to the target value , then that pointer is called *multiple indirection*, or *pointers to pointers*.
- Below figure shows the concept of multiple indirection.
- 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. This can be done 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;

the **newbalance** is not a pointer to a floating-point number but rather a pointer to a **float** pointer.



Single and multiple indirection

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;
        }
```

Here, **p** is declared as a pointer to an integer and **q** as a pointer to a pointer to an integer. The call to **printf( )** prints the number **10** on the screen.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

type (***pointer-name**)(*parameter*);

Example :

```
int (*sum)();   //legal declaraction of pointer to function
int *sum();   //This is not a declaraction of pointer to function
A function pointer can point to a specific function when it is assigned the name of the function.
int sum(int, int);
int (*s)(int, int);
s = sum;
```

s is a pointer to a function **sum**. Now **sum** can be called using function pointer **s** with the list of parameter.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>
#include <conio.h>
int sum(int x, int y)
{
return x+y;
}
int main( )
{
        int (*fp)(int, int);
        fp = sum;
        int s = fp(10, 15);
        printf("Sum is %d",s);
        getch();
        return 0;
}
```

Output : 25

Dynamic Memory Allocation in C

The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into a executable form.
The size of array you have declared initially can be sometimes insufficient and sometimes more than required.
Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.
Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr=(cast-type*)malloc(byte-size)
```

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size.
If the space is insufficient, allocation fails and returns NULL pointer.

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

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation".
The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

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

```
{
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
if(ptr==NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
scanf("%d",ptr+i);
sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
 ptr=(int*)calloc(n,sizeof(int));
if(ptr==NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
scanf("%d",ptr+i);
sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient.
Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, *ptr* is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
int *ptr,i,n1,n2;
printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
printf("%u\t",ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=realloc(ptr,n2);
for(i=0;i<n2;++i)
printf("%u\t",ptr+i);
return 0;
}
```

Problems with pointers

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Basically, pointer bugs are difficult to debug. Its programmer's responsibility to use pointers effectively and correctly.