

Unit - I

Data types in C Language

- ✓ Data types in C are used to specify what type of data the data elements are going to hold during program execution.
- ✓ C language has some predefined set of data types to handle various kinds of data in our program. They are categorized based on different storage capacities.

C language supports 2 different types of data types:

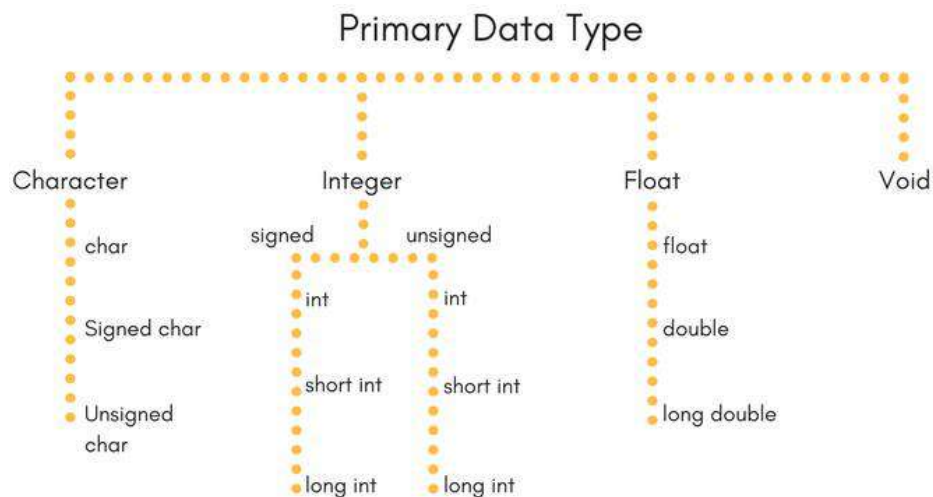
Primary data types:

These are fundamental data types in C namely integer (int), floating point (float), character (char) and void.

Derived data types:

Derived data types are nothing but primary datatypes but are grouped together like array, stucture, union and pointer.

- ✓ Data type determines the type of data a variable will hold.
- ✓ If a variable x is declared as int. it means x can hold only integer values.
- ✓ Every variable which is used in the program must be defined as what data-type it is.



VOID TYPE

- ✓ void type means no value and no operations.
- ✓ This is usually used to specify the type of functions which returns nothing.
- ✓ The more use of this type can be seen in more advanced topics of C language, like functions, pointers etc.

INTEGER TYPE:

- ✓ Integers are used to store whole numbers.
- ✓ An integer type is a number without a fraction part.
- ✓ C supports 3-different types of integers and they can be signed and unsigned.
- ✓ C defines these data types so that they can be organized from smallest to the largest.
`sizeof(short)<sizeof(int)<sizeof(long)`

- ✓ An unsigned data type has the same memory requirements as a signed data type.
- ✓ In the case of a signed data type the leftmost bit is reserved for the sign and in an unsigned data type all of the bits are used to represent the numerical value. Thus, an unsigned data type can be approximately twice as large as signed (negative values are not permitted).
- ✓ For example, if signed int can vary from -32,768 to +32,767 (which is typical for a 2-byte int), then an unsigned int will be allowed to vary from 0 to 65,535.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32,767
unsigned int	2	0 to 65,535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

CHARACTER TYPE:

Character types are used to store characters value.

In C a character is any value that can be represented in the computer's alphabet (known as **character set of C**)

In C a character is 1-byte long to store characters. A byte is 8-bits long, with 8-bits there are 256 different possible values for character.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

FLOATING POINT TYPE:

- ✓ In C floating points are used to store real values.
- ✓ A real value consists of an integral and a fractional part.
- ✓ C language supports three different sizes of real types: float, double and long double.
- ✓ These real types can be organized from smallest to largest. sizeof(float) < sizeof(double) < sizeof(long double).

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

TYPE CONVERSION AND CASTING:

- ✓ In C type conversion is required whenever we have mixed types in an expression.
Example: multiplying *float* with *int*.
- ✓ To perform this type of expression evaluations one type must be converted to the other type.
- ✓ In C, there are two types of type conversions: implicit conversion and explicit conversion. Explicit type conversion can also be known as *type casting*.

Implicit Type Conversion:

When the types of the different operands in an expression are differ, C automatically converts the complete expression type to the maximum available type in the expression. This type of conversion is known as *implicit type conversion*.

There are two things: consider the following code segment.

```
int i=10;
float f=20;
short s;
s=i*f;
```

The First thing in the above code example: we have multiplicative expression in the right side of assignment operator, where integer and float values were multiplied. As it is a mixed type expression the final result of the evaluation always promoted to the highest type in the expression. As a result the final value of the expression will be in float.

The second thing: when assigning the value to the left variable, based on the type difference either promotion or demotion will occur.

Promotion: When the left hand side variable's type is higher than the right hand side expression's type then C automatically promotes the right side expression's type to match left side variable's type. This is known as promotion. Generally because of promotion no problems will arise in programming i.e. there will not be any data loss.

Demotion: When the left hand side variable's type is lower than the right hand side expression's type then C automatically demotes the right hand side expression's type to match left side variable's type. This is known as demotion.

Demotion may or may not create problems. If the size of the left side variable can accommodate the value of the expression, there will be no problem. But if the size of the left side variable is smaller than the value of the expression then there will be surprising results like: when a float value is assigned to an integer the fractional part is dropped (truncated), and when an integer value larger than the range of short is assigned to a short variable the process will consider the values from circular range assignment.

Explicit type conversion/Casting

- ✓ Rather than let the compiler implicitly convert the data type, we can convert the data from type to another using explicit type conversion.
- ✓ Explicit type conversion uses the unary cast operator.

- ✓ To cast the data from one type to another type, specify the new type in parentheses before the value to be converted.
- ✓ One of the important uses of casting is to ensure that the result of a divide operation between two integers must be a real number.

Example: consider the following code segment.

<pre>int a=10,b=3; float f; f=a/b; printf("%f",f);</pre>	<pre>int a=10,b=3; float f; f=(float)a/b; printf("%f",f);</pre>
----------------------------------------------------------	-----------------------------------------------------------------

PRECEDENCE AND ASSOCIATIVITY

- ✓ Precedence is used to determine the order in which different operators in a complex expression are evaluated.
- ✓ Associativity is used to determine the order in which operators with the same precedence are evaluated in a complex expression.
- ✓ Precedence is applied before the associativity to determine the order in which expressions are evaluated. Associativity is then applied, if necessary.
- ✓ Consider the operator precedence and associativity table from table.

Example: 1) $2+3*4 \rightarrow 14$

2) $-b++ \rightarrow -(b++) \rightarrow$ if $b=5$ then $-(b++) = -5$

3) $c += (a > 0 \ \&\& \ a \leq 10) ? ++a : a/b;$ if $a=1, b=2 \ \& \ c=3$ then The result is $c=5$

ARRAYS:

- ✓ If we have to read, process and print 10 integers, then we need 10 integers in memory for the duration of the program.
- ✓ We can declare and define 10 variables with a different name.
- ✓ To read these 10 variables from the keyboard we need 10 read statements, and also to print these 10 variables onto the output screen we need 10 output (printf) statements. This is something redundancy in programming code.
- ✓ The approach may look simpler and acceptable to some extent for 10 variables, but if the variable count increases rapidly nowhere the above concept is acceptable.
- ✓ To process large amount of data, C uses a powerful data structure called array.
- ✓ An array is a collection of elements of the same data type.
- ✓ Array elements can be initialized, accessed and manipulated using the indexing.
Example: `score[10]` Here score will contain 10-elements
- ✓ Array index always start at '0' and end at total-number of elements minus one.
- ✓ In our example the index for score will start at 0 and end at 9.

Using Arrays in C

C provides two different types of arrays

- One-Dimensional arrays.
- Two-Dimensional arrays.

In One-dimensional arrays the data are organized linearly in only one direction.

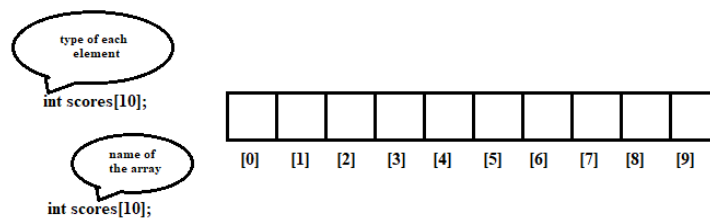
In Two-Dimensional arrays the data are organized in rows and columns, like a matrix.

Array declaration and definition

- ✓ Like every other object in C, an array must be declared, defined and initialized before it can be used in the program.
- ✓ Array declaration tells the compiler the name of the array.
- ✓ Array definition specifies the size or number of elements in the array. In a fixed length array, the size of the array is a constant and must have a value at the compile time.

Example: `type arrayname[array_size];`

- ✓ The declaration and definition format for a variable length is same as like a fixed length array except the size is a variable. The array size will be determined when the program is executed. Once the size is determined it cannot be changed.



Accessing elements in arrays:

- ✓ To access the individual elements in an array the array index can be used. The index must be an integral value or can be an expression that evaluates to an integral value.
- ✓ To process all the elements in the array a loop can be used. Consider the following code segment:

```
for(i=0;i<10;i++)  
    printf("%d", scores[i]);
```

- ✓ The arrays name is the symbolic reference for the address to the first byte of the array. The index represents an offset from the beginning of the array to the element being referenced.

Storing values in Arrays:

- Declaration and definition only reserves space for the elements in the array.
- To store values in the array they must be initialized.
- To initialize values in the array elements they can be read from the keyboard or they can be assigned with individual values.

Initialization:

- ✓ Array elements in a fixed-length array can be initialized when they are declared. For variable-length arrays they cannot be initialized when they are defined.
- ✓ To initialize the array elements with a set of values, they must be enclosed in braces and separated by commas.
- ✓ It is a compile error to specify more values than elements in the array.

- ✓ The initialization can be done in the following ways:

- (a) Basic initialization: `int scores[5]={3,7,12,24,45};`
- (b) Initialization without size: `int scores[]={3,7,12,24,45};`
- (c) Partial initialization: `int scores[5]={3,7,0,0,0};`
- (d) Initialization to all zeros: `int scores[5]={0};`

- ✚ The first example array the score array will have 5-elements and they contain specific set of values.
- ✚ In the second example, when the array is completely initializing then the size attribute can be omitted.
- ✚ In the third example, if the array is initialized with partial values rest of the elements will be initialized to zero.
- ✚ In the fourth example can be used to initialize all the elements to zeros.

Inputting values:

- ✓ An array can be initialized from the keyboard. The values can be read from the keyboard and initialized to array.
- ✓ The most appropriate loop to use with arrays is the *for* loop.

```
for(i=0;i<9;i++)
    scanf("%d",&scores[i]);
```

Assigning values:

- ✓ Array elements can be assigned individually by using a set of values, with the help of assignment operator.
Example: `scores[4]=10;`
- ✓ Assigning one array to another array is not possible even if they match fully in type and size. To do so we have to copy all the elements individually from one array to another.

```
for(i=0;i<10;i++)
    scores[i]=temp[i];
```

Printing values:

- ✓ To print the contents of the array, a normal *for* loop can be used.
Example: `for(i=0;i<10;i++)`

```
    printf("%d ",scores[i]);
```

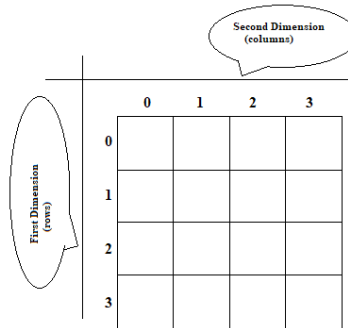
Index Range checking:

- ✓ The C-language does not check the boundary of an array.
- ✓ As a programmer it is our job to ensure all references to indexed elements are valid and within the range.
- ✓ If an array is used with an invalid index, the results will be unpredictable.

```
Example:    for(i=1;i<=10;i++) // indexing going out of range
            printf("%d ",scores[i]);
```

TWO-DIMENSIONAL ARRAYS

- ✓ In One-dimensional arrays the data are organized linearly in only one direction.
- ✓ Many applications require data to be stored in more than one dimension.
- ✓ Matrices require an array that consists of rows and columns as shown in the following figure, which is generally called as two-dimensional array.



- ✓ In C-language a two dimensional array will be considered as an array of arrays. That is a two dimensional array is an array of one-dimensional arrays.

Declaration:

- ✓ Two-dimensional arrays like One-dimensional arrays must be declared and defined before being used.
- ✓ Declaration tells the compiler the name of the array; definition specifies the type and size of each dimension.
- ✓ The size of a two array must be a constant and must have a value at compile time.

Example: `int scores[5][4];`

Initialization:

- ✓ The definition of a Two-Dimensional array only reserves the memory for the elements in the array.
- ✓ No values will be stored in the locations. The locations will generally contain unpredictable values or garbage values without initialization.
- ✓ Initialization of array can be done when the array is defined.
- ✓ The values for the array must be enclosed in braces.
- ✓ Nested braces must be used to show the exact number of rows and columns.
- ✓ In a Two-Dimensional array, if the array is completely initialized with values, only the first dimension can be omitted. The second dimension must need to be specified.

Example: `int scores[3][2]={2,3,5,4,6,9};`

- ✓ The whole array can be initialized to zeros.

Example: `int scores[5][4]={0};`

Inputting Values:

- ✓ Another way to initialize the values is, we can read them from the keyboard.
- ✓ In Two-Dimensional array, it usually requires nested *for* loops.
- ✓ The first loop, the outer loop controls the rows from zero to the maximum number and the inner loop controls the columns from zero to the maximum number.

Example: `for(i=0;i<2;i++)
for(j=0;j<2;j++) scanf("%d",&scores[i][j]);`

Outputting values:

- ✓ The values inside a Two-Dimensional array can be printed using two nested loops.
- ✓ The first loop, the outer loop controls the rows from zero to the maximum number and the inner loop controls the columns from zero to the maximum number.
- ✓ To print the matrix in a table format, a newline is printed at the end of each row.

Example: for(i=0;i<2;i++)
 {
 for(j=0;j<2;j++)
 printf(“ %d”,scores[i][j]);
 printf(“\n”);
 }

Accessing values:

- ✓ Individual elements can be initialized using the assignment operator.
scores[2][0]=23;
scores[2][2]=scores[1][1]+12;

STRINGS

- ✓ A string is a series of characters treated as a single unit.
- ✓ String is a variable length piece of data. Strings are divided into two major categories.
 - Fixed length strings.
 - Variable length strings.

Fixed length strings:

- ✓ In implementation of a fixed-length string, the important thing is to make decide the size of the variable. If we make it too small, it sometimes can't store all of the data. If we make it too big, we waste the memory.
- ✓ Another problem is how to differentiate between data and non-data. A solution is to use special characters like spaces to indicate non data. In this case the space can't be used as a character in the string.

Variable-Length strings:

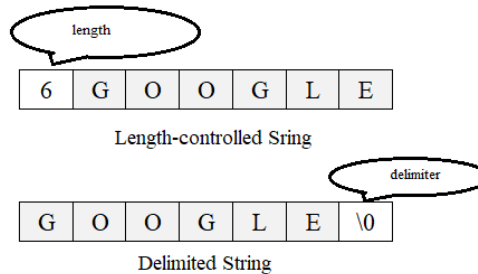
- ✓ A much preferred solution is to create a structure that can expand and contract according to the size of the value we want to store.
- ✓ Example: To store a person name with only three characters the structure should contract and provide three characters and to store a person name with 30-characters the structure should expand and provide 30-characters.
- ✓ Here also there is a problem: as we are using a variable length structure in computers memory, there must be a way to indicate the end of the string.
- ✓ Two common techniques are used to indicate the end of the string.
 - Length-Controlled Strings
 - Delimited Strings

Length-Controlled Strings:

- ✓ In this type of strings a count is used as the first character in the string that specifies the number of characters in the string.
- ✓ This count then be used by string manipulation functions to determine the length of the string.

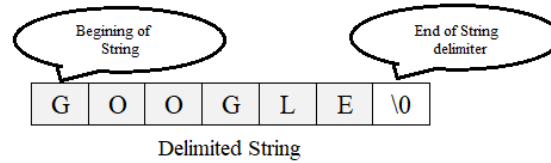
Delimited Strings:

- ✓ In this type of strings the end is specified by a special character known as delimiter, hence the name delimited strings.
- ✓ This concept is similar to using a full stop (.) at the end of a sentence in English. Where each sentence is of variable length and a special symbol full stop is used to indicate the end of the sentence.
- ✓ The disadvantage of using a delimiter is that it eliminates one character from being used for data in the string.
- ✓ The most common delimiter is a null character (`\0`).
- ✓ All C strings are of variable length and delimited once.

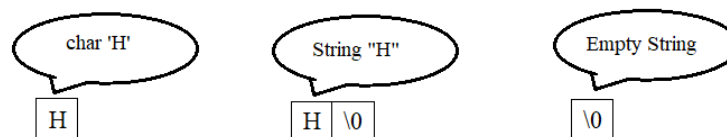


C-Strings

- ✓ A C string is a variable length array of characters that is delimited by the null character.
- ✓ A string is stored in an array of characters and delimited by a null character (`\0`).



- ✓ As a string is stored in an array, the name of the string is a pointer to the beginning of the string.
- ✓ There is a big difference between how a character is stored in memory and a one-character string is stored. The character requires only one memory location, but the one-character string requires two memory locations one for actual character and one for delimiter.
- ✓ To store an empty string in memory also requires one memory location to store the delimiter.



String Constants:

- ✓ A string literal or a string constant is a sequence of characters enclosed in double quotes.
- ✓ When string constants are used in a program, C automatically creates an array of characters, initializes to a null delimited string, and stores it.
Example: "`C is a programming language`"
 "`Hello World`"
- ✓ A string literal is stored in memory like any other object. It has an address, and can be referred by using a pointer. Here the string literal as it is a sequence of characters, itself a pointer constant to the first element.
- ✓ The string itself can be used to refer the individual characters by using index.
Example: "`hello`"[1] → e

Declaring and defining Strings:

- ✓ C has no string type. To use strings in C programming they must be declared and defined as character arrays.
- ✓ When defining the array to store a string, we must provide enough space for both actual data and delimiter.
Example: `char str[9];`
- ✓ In this example it is possible to store a string of eight characters long as the last character must be a delimiter.
- ✓ A character pointer can also be used to hold strings of variable length in a program, because the name of the string or the string itself is a reference to the starting memory location.
Example: `char* pStr;`

Initializing Strings:

- ✓ Initialization of strings is much like initialization any other object in C. We use assignment operator to initialize the string when they are defined.
Example: `char str[9]="Good day";`
- ✓ As declaration, definition and initialization all are happening at a same point of time, the size attribute can be ignored.
Example: `char str[]="Good day";`
- ✓ Strings can be initialized by using a set of characters where the last character is going to be the delimiter. This method is not used, because it is very tedious to code.
Example: `char str[9]={ 'G','o','o','d',' ','D','a','y','\0' };`
- ✓ The most common way for defining and initializing a string in programming is using a pointer. This method first creates a string literal into the memory and assigns its address to the character pointer. Later that pointer can be used to refer the string.
Example: `char* pStr="Good Day";`

String Input/Output functions

Functions `gets()` and `puts()` are two string functions to take string input from the user and display it respectively

```
#include<stdio.h>

int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);      //Function to read string from user.
    printf("Name: ");
    puts(name);      //Function to display string.
    return 0;
}
```

Strings and the Assignment Operator

- ✓ The string is an array and the name of the string is a pointer constant.
- ✓ As a pointer constant it should always be used as an rvalue but not as lvalue.
Example: `char str1[6]="Hello";`

```
char str2[6];  
str1=str2; //compile error
```

String Manipulation Functions:

C provides different predefined set of string functions which are helpful in manipulating the strings.

String Length

- ✓ The string length function (strlen) returns the length of a string that is the number of characters in the string excluding the null character.
- ✓ If the string is empty then the string length function returns zero.
- ✓ Syntax: `int strlen(char* string);`

//Program for printing the length of the given string using strlen(); function

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
void main(void)  
{  
    char* str="Hello World";  
    int len;  
    clrscr();  
    len=strlen(str);  
    printf("The length of the string is:%d",len);  
    getch();  
}
```

Output:

The length of the string is:11

String Copy

- ✓ The String copy functions strcpy(); copies the contents from one string including the null character to another string.
- ✓ There are two string copy functions:
 - I. Basic string copy strcpy();
 - II. String copy length controlled strncpy();

I. Basic string copy strcpy();

- ✓ The basic string copy function strcpy(); copies the contents of the from one string *fromstr* including the null character to another string *tostr*.
Syntax: `char* strcpy(char* tostr, char* fromstr);`
- ✓ If *fromstr* is longer than the *tostr* then, the data in memory after *tostr* is destroyed.
- ✓ The destination string should be large enough to hold the source string.
- ✓ If *fromstr* is longer than *tostr*, the data in memory after *tostr* is destroyed.
- ✓ The function returns the address of *tostr*.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char source[ ] = "Hyderabad" ;
    char target[20]= "" ;
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
    strcpy ( target, source ) ;
    printf ( "\ntarget string after strcpy( ) = %s", target ) ;
}
```

Output:

```
source string = Hyderabad
target string = 
target string after strcpy( ) = Hyderabad
```

II. String copy length controlled strncpy();

- ✓ String copy length controlled strncpy(); function copies the content of one string to another, but it sets a maximum number of characters that are to be copied.
- ✓ This function contains a parameter that specifies the maximum number of characters that can be copied at a time.
Syntax: char* strncpy(char* tostr, char* fromstr, int size);
- ✓ If the actual size of the *fromstr* is equal or greater than size parameter, then *size* number of characters are copied.
- ✓ If the *fromstr* size is smaller than the *size* parameter, the entire string is copied and then null characters are inserted into the *tostr* until the *size* parameter is satisfied.
- ✓ If the *fromstr* is longer than the *size* the copy stops after size bytes have been copied. In this case the destination variable *tostr* may not be a valid string; that is it may not have a delimiter.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char source[ ] = "Hyderabad" ;
    char target[20]= "" ;
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
    strncpy ( target, source,3 ) ;
    target[4]='\0';
    printf ( "\ntarget string after strcpy( ) = %s", target ) ;
}
```

Output:

```
source string = Hyderabad
target string = 
target string after strcpy( ) = Hyd
```

String Compare

C has two string compare functions:

- I. Basic string compare strcmp();
- II. String compare length controlled strncmp();

I. Basic string compare strcmp();

- ✓ The strcmp(); function compares the two string until an unequal character is found or until the end of the string is reached.
- ✓ This function returns an integer to indicate the result of the comparison.
Syntax: int strcmp(char* str1, char* str2);
- ✓ If both strings are equal the function returns zero. If length of str1 < str2, it returns < 0 value. If length of str1 > str2, it returns > 0 value.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str1[ ] = "Hyderabad" ;
    char str2[ ] = "Hydarabad" ;
    int i, j, k ;
    i = strcmp ( str1, "Hyderabad" ) ;
    j = strcmp ( str1, str2 ) ;
    k = strcmp ( str1, "I" ) ;
    printf ( "\n%d %d %d", i, j, k ) ;
}
```

Output

```
0  4  -1
```

II. String comparison length controlled strncmp();

- ✓ The strncmp(); function compares the two strings until unequal character is found in a specified number of characters have been tested, or until the end of the string is reached.
- ✓ This function returns an integer to indicate the result of the comparison.
Syntax: int strncmp(char* str1, char* str2, int size);
- ✓ If both strings are equal the function returns zero. If length of str1 < str2, it returns < 0 value. If length of str1 > str2, it returns > 0 value.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str1[ ] = "Hyderabad" ;
    char str2[ ] = "Hydarabad" ;
    int i, j, k;
    i = strncmp ( str1, str2, 3 ) ;
    j = strncmp ( str1, str2, 4 ) ;
    k = strncmp ( str1, str2, 10 ) ;
    printf ( "\n%d %d %d", i, j, k ) ;
}
```

Output

```
0  4  4
```

String Concatenation

- ✓ The string concatenation function appends one string to the end of another string.
- ✓ The size of the destination string should be large enough to hold the resulting string. If it is not the data at the end of the destination string will be destroyed.
- ✓ C has two string concatenation functions:
 - I. Basic string concatenation `strcat()`;
 - II. String concatenation length controlled `strncat()`;

I. Basic string concatenation `strcat()`;

- ✓ The function copies `str2` to the end of `str1`, beginning at the delimiter. That is the delimiter is replaced with the first character of the `str2`. The delimiter from `str1` is copied to the resulting string.

Syntax: `char* strcat(char* str1, char* str2);`

- ✓ The resulting string length is the sum of the length of `str1` plus the length of `str2`.
- ✓ The function returns the address pointers to the resulting string.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str1[ ] = "Hyderabad" ;
    char str2[ ] = "ISL" ;
    strcat ( str1, str2 ) ;
    printf ( "%s", str1 ) ;
}
```

Output

HyderabadISL

II. String concatenation length controlled `strncat()`;

- ✓ The `strncat()` function copies only the specified number of characters from source string `str1` to destination string `str2`. A null character is appended at the end.
Syntax: `char* strncat(char* str1, char* str2, int size);`
- ✓ If the length of `str2` is less than `size`, then the function will work in same as basic string copy.
- ✓ If the length of `str2` is greater than `size`, then only the number of characters specified by `size` are copied, and a null character is appended at the end.
- ✓ If the value of `size` is zero or less than zero, then no characters are copied.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str1[ ] = "Hyderabad" ;
    char str2[ ] = "ISL" ;
    strncat ( str1, str2, 2 ) ;
    printf ( "%s", str1 ) ;
}
```

Output

HyderabadIS

Character in String

- ✓ These set of functions are helpful to find the location of a character in a string.
- ✓ In C there are two string functions to search for a character in a string.
 - I. String character strchr();
 - II. String rear character strrchr();

I. String character strchr();

- ✓ The strchr(); function searches for the first occurrence of a character from the beginning of the string.
- ✓ If the character is located the function returns a pointer to it.
- ✓ If the character is not in the string, the function returns a null pointer.
- ✓ Syntax: char* strchr(char* str, char ch);

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str[ ] = "Hyderabad";
    char* strtmp;
    strtmp = strchr ( str, 'a' ) ;
    printf ( "%s", strtmp) ;
}
```

Output

abad

II. String rear character strrchr();

- ✓ The strrchr(); function searches for the first occurrence of a character from the rear end of the string.
- ✓ If the character is located the function returns a pointer to it.
- ✓ If the character is not in the string, the function returns a null pointer.
- ✓ Syntax: char* strrchr(char* str, char ch);

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str[ ] = "Hyderabad";
    char* strtmp;
    strtmp = strrchr ( str, 'a' ) ;
    printf ( "%s", strtmp) ;
}
```

Output

ad

Search for a substring

- ✓ Searching for a sub-string in a string can be done only from the beginning of the string.
- ✓ The function strstr(); is used to locate a substring in a string from the beginning of the string.
Syntax: char* strstr(char* string, char* sub_string);
- ✓ If sub_string exists in string then a pointer pointing to the first character of the sub_string is returned.
- ✓ If sub_string does not exist in string then a null pointer is returned.

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char str[ ] = "Hyderabad";
    char* strtmp;
    strtmp = strstr ( str, "era" ) ;
    printf ( "%s", strtmp) ;
}
```

Output

erabad

Unit – III, Basic Algorithms

- ✓ **Algorithm:** In programming, algorithm is a set of well-defined instructions in sequence to solve the problem.
- ✓ Algorithms are universal. The algorithm you use in C programming language is also the same algorithm you use in every other language.
- ✓ Generally, the algorithms are written in natural languages, irrespective to the computer language used to solve the problem.

Qualities of a good algorithm:

1. Input and output should be defined precisely.
2. Each step in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

SORTING

- ✓ One of the most common applications in computer science is sorting.
- ✓ Sorting is the process through which data are arranged according to their values.
- ✓ If the data are not arranged in an order, it will take hours trying to find a single piece of information.
- ✓ Example: Finding some one's telephone number in a telephone directory if it is not ordered in name sequence is a difficult job.
- ✓ Sorting is a combination of two different operations: compare and swap. In the comparison operation two different element values of the array are compared. After comparison based on the requirement the values are swapped. Swapping is interchanging the values in elements of the array.

```
for(i=0;i<9;i++)  
    if(a[i]>a[i+1])  
    {  
        temp=a[i];  
        a[i]=a[i+1];  
        a[i+1]=temp;  
    }
```

- ✓ There are different sorting mechanisms: Selection sort, Bubble sort, Merge sort, Heap sort ... etc.

SEARCHING

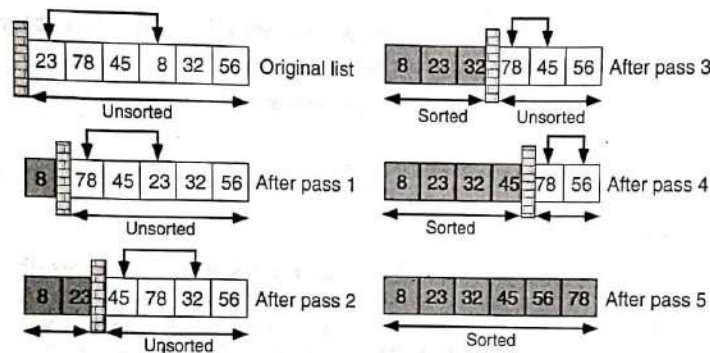
- ✓ Another common operation in computer science is searching.
- ✓ Searching is a process used to find the location of a target among a list of objects.
- ✓ In an array searching means that given a value, finding the location (index) of the first element in the array that contains that value.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
4	21	36	14	46	62	91	8	7	81

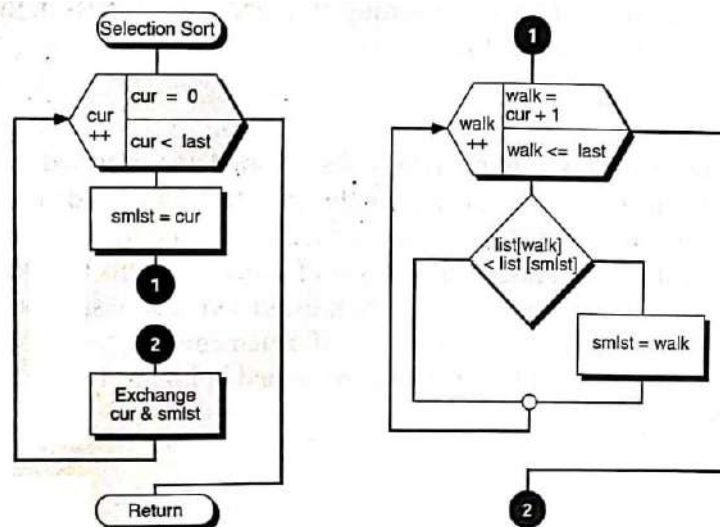
- ✓ There are two basic searching techniques that can be applied to an array of elements.
- ✓ The Linear/Sequential search is used whenever the list is not ordered, small in size and not searched too often.
- ✓ The Binary search is used, if the list is ordered, large in size and searched too often.

SELECTION SORT:

- ✓ In the selection sort, the list is divided into two sub-lists, sorted and unsorted, which are divided by an imaginary wall.
- ✓ The smallest element from the unsorted sub-list is found and swapped with the element at the beginning of the unsorted sub-list.
- ✓ After each selection and swapping, the imaginary wall between the two sub-lists moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted elements.
- ✓ Each time an element is moved from the unsorted sub-list to the sorted sub-list, we call it a sort pass is completed.
- ✓ To sort a list of n-elements, n-1 sort passes are needed.



Selection Sort Example



Design for Selection Sort

```
// C program for implementation of selection sort
```

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

```
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

```
void selectionSort(int list[], int last)
{
    int cur, walk, smlst;

    // One by one move boundary of unsorted subarray
    for (cur = 0; cur < last; cur++)
    {
        // Find the minimum element in unsorted array
        smlst = cur;
        for (walk = cur+1; walk <= last; walk++)
            if (list[walk] < list[smlst])
                smlst = walk;

        // Swap the found minimum element with the first element
        swap(&list[smlst], &list[cur]);
    }
}
```

```
/* Function to print the list */
void printArray(int list[], int last)
{
    int i;
    for (i=0; i < last; i++)
        printf("%d ", list[i]);
    printf("\n");
}
```

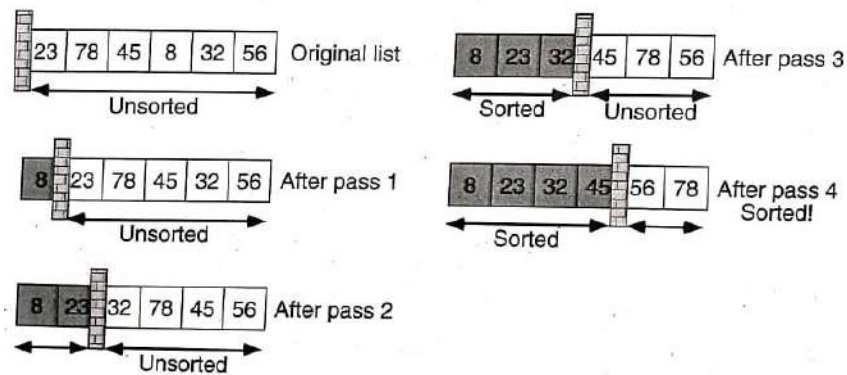
```
// Driver program to test above functions
int main()
{
    int list[] = { 64, 25, 12, 22, 11 };
    int last = sizeof(list)/sizeof(list[0]);
    selectionSort(list, last-1);
    printf("Sorted array: \n");
    printArray(list, last);
    return 0;
}
```

Output:

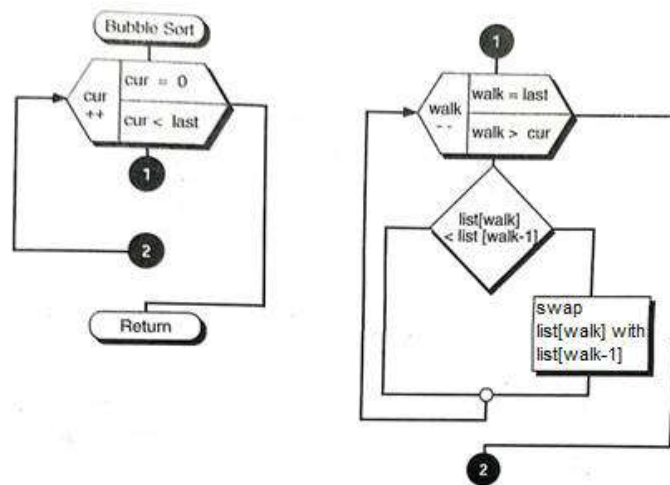
Sorted array:
11 12 22 25 64

BUBBLE SORT:

- ✓ In the bubble sort, the list is divided into two sub-lists, sorted and unsorted.
- ✓ The smallest element is bubbled from unsorted sub-list and moved to the sorted sub-list.
- ✓ After moving the smallest element to the sorted sub-list, the imaginary wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted elements.
- ✓ Moving one element each time from unsorted sub-list to the sorted sub-list completes one sort pass.
- ✓ Given a list of n-elements, the bubble sort takes n-1 sort passes to sort the data.
- ✓ Remember the bubble sort concept starts from the end of the list.



Bubble Sort Example



Bubble Sort Design

// C program for implementation of Bubble sort

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

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

```
// A function to implement bubble sort
void bubbleSort(int list[], int last)
```

```

{
int cur, walk,temp;
for (cur = 0; cur < last; cur++)

    // Last i elements are already in place
    for (walk = last; walk > cur; walk--)
        if (list[walk] < list[walk-1])
            swap(&list[walk-1], &list[walk]);
}

/* Function to print an array */
void printArray(int list[], int last)
{
    int i;
    for (i=0; i < last; i++)
        printf("%d ", list[i]);
}

// Driver program to test above functions
int main()
{
    int list[] = {64, 34, 25, 12, 22, 11, 90};
    int last = sizeof(list)/sizeof(list[0]);
    bubbleSort(list, last-1);
    printf("Sorted array: \n");
    printArray(list, last);
    return 0;
}

```

Output:

Sorted array:

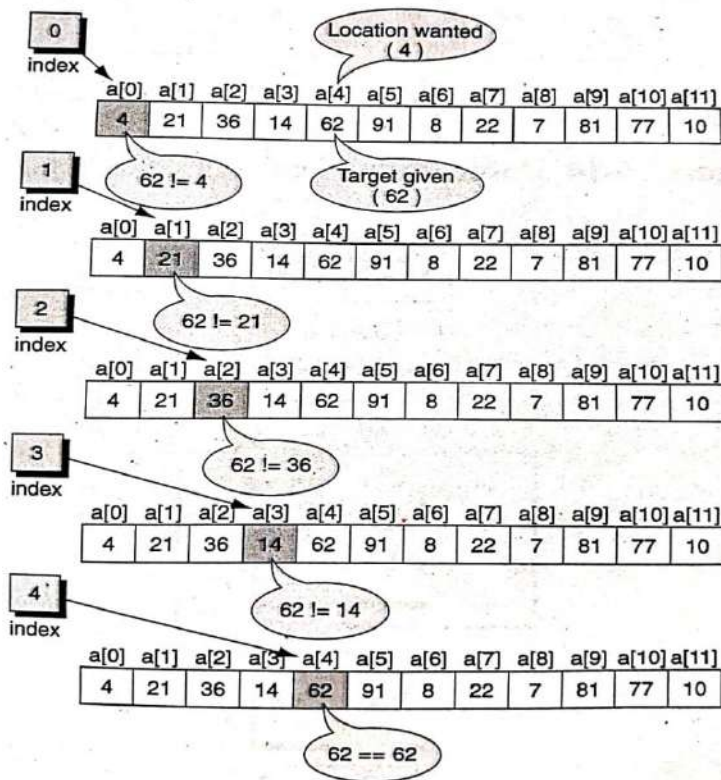
11 12 22 25 34 64 90

LINEAR/SEQUENTIAL SEARCH

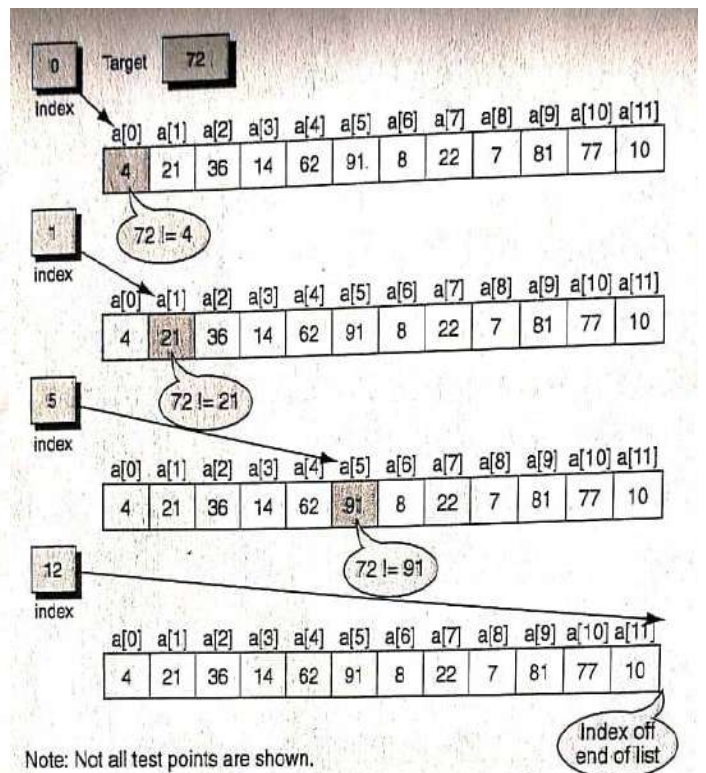
- ✓ In sequential search, the searching for the target will start from the beginning of the list, and will continue until the target is found or until the end of the list is reached (no target found case).

Example:

- ✓ The following figure lists the steps to find 62. The process will first compare the value with data at index 0, then 1, 2, and 3 before finding the element at 5th location (index 4).
- ✓ If the target value is not in the list, then the comparison operation will reach the end of the list, making sure that the target is not in the list. The figure also lists the steps for finding the value 72, which is not there in the list.

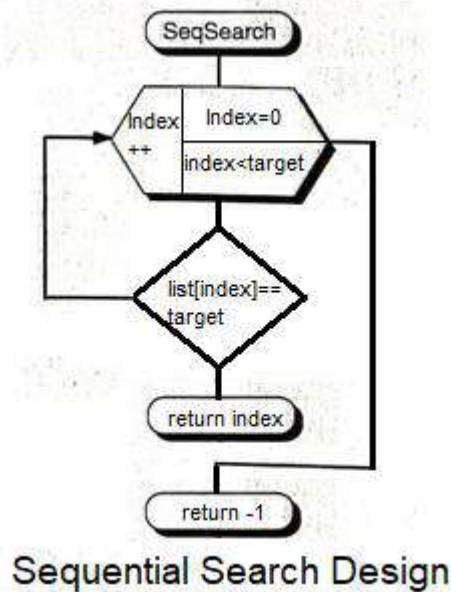


Locating Data in Unordered List



Note: Not all test points are shown.

Unsuccessful Search in Unordered List



Sequential Search Design

// C-Program for Linear Search.

```

#include <stdio.h>
int seqsearch(int list[], int last, int target)
{
    int index;
    for (index = 0; index < last; index++)
        if (list[index] == target)

```

```

        return index;
    return -1;
}

int main(void)
{
    int list[] = { 2, 3, 4, 10, 40 };
    int target = 10;
    int last = sizeof(list) / sizeof(list[0]);
    int result = seqsearch(list, last, target);
    if(result== -1)
        printf("Element is not present in array");
    else
        printf("Element is present at index %d", result);
    return 0;
}

```

Output:

Element is present at index 3

BINARY SEARCH

- ✓ The sequential search algorithm is very slow.
- ✓ If we have an array of 1 million elements, we must do 1 million comparisons in the worst case.
- ✓ If the array is not sorted, this is the only solution.
- ✓ If the array is sorted, we can use a more efficient algorithm called binary search.
- ✓ We have to use binary search whenever the list contains more than 50 elements.

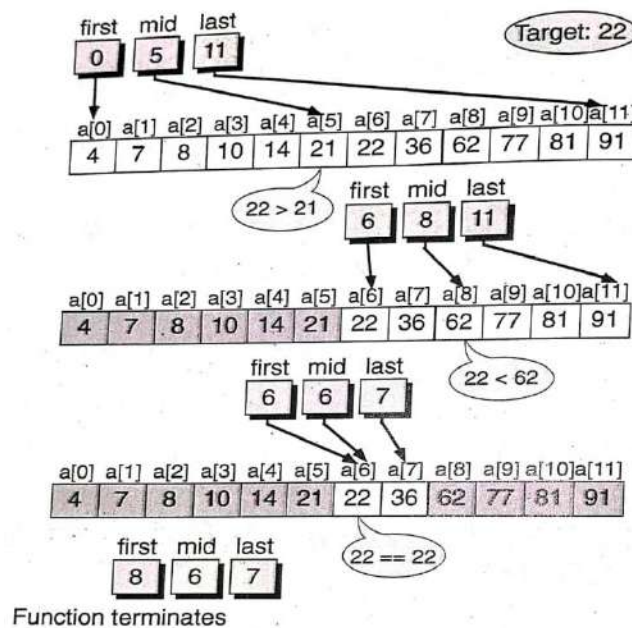
How it works:

- ✓ The binary search starts by comparing the data in the element in the middle of the list. This determines if the target is in the first half or second half of the list.
- ✓ If the target is in the first half, no need to search the second half and if the target is in the second half, then no need to search the first half.
- ✓ Either way half of the list will be eliminated from consideration.
- ✓ The same process will be repeated until the target is found or it'll be confirmed that the target is not in the list.
- ✓ The two important cases in binary searching process are:
 - The target found case.
 - The target not found case.

To find the middle of the list, we need three variables, one to identify the beginning of the list (*first*), one to identify the middle of the list (*mid*), and one to identify the end of the list (*last*).

Target Found:

- ✓ The following figure shows finding 22 in a sorted list. In our case *first* will be 0 and *last* will be 11 and *mid* can be calculated as follows.
- ✓ $mid = (first + last) / 2;$

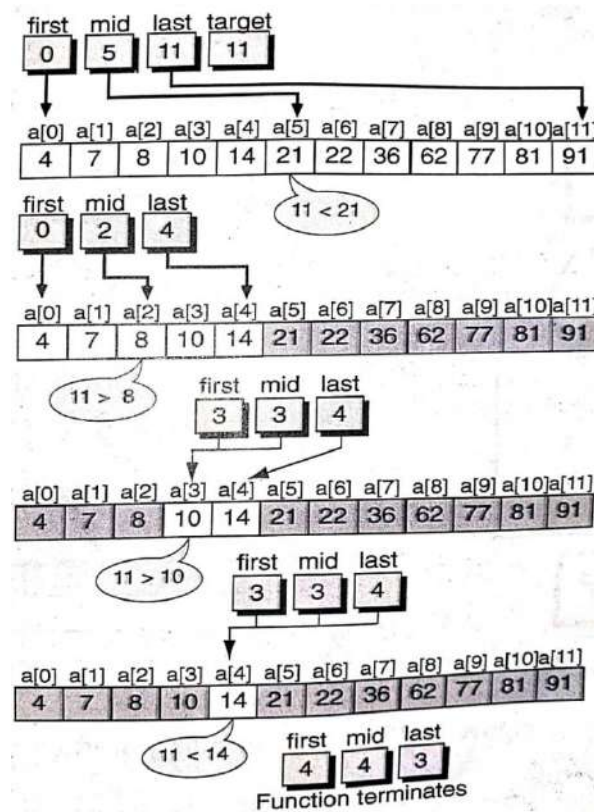


Binary Search Example

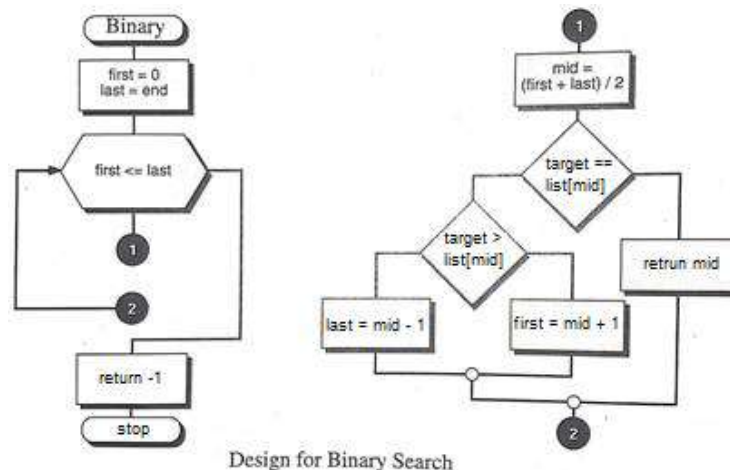
- ✓ Since the index *mid* is an integer, the result of will be an integral value.
- ✓ $mid = (0 + 11) / 2 = 11 / 2 = 5$
- ✓ At index location 5, the target is greater than the list value ($22 > 21$). Therefore the list location from 0 through 5 can be eliminated.
- ✓ Now in the step the *first* will be made as: $first = mid + 1$ i.e. $first = 5 + 1 = 6$. And further calculates the *mid* by using same formulae: $mid = (first + mid) / 2$ i.e. $mid = (6 + 11) / 2 = 17 / 2 = 8$.
- ✓ Again the target will be tested with value at *mid*, this time the target is less than the list value ($22 < 62$).
- ✓ This eliminates locations 8 through 11.
- ✓ This time last will be adjusted as: $last = mid - 1$ i.e. $last = 8 - 1 = 7$ and $mid = (first + last) / 2 = (6 + 7) / 2 = 6$.
- ✓ Now at *mid* location the value matches with the target. This will stop the search.

Target Not Found:

- ✓ The target not found in the binary search is something when we searched all the possibilities and didn't find the target in the list.
- ✓ This is done in the binary search when *first* becomes greater than *last*.
- ✓ Thus only two conditions terminate the binary search algorithm: Either the target is found or first becomes larger than *last*.



Unsuccessful Binary Search Example



Design for Binary Search

//Binary Search algorithm

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

```
int binarySearch(int list[], int first, int last, int target)
```

```
{
```

```
    while (first <= last) {
```

```
        int mid = first + (last - first) / 2;
```

```
        if (target == list[mid])
```

```
            return mid;
```

```
        if (target > list[mid])
```

```
            first = mid + 1;
```

```
        else
```

```
            last = mid - 1;
```

```
    }
```

```

        return -1;
    }

int main(void)
{
    int list[] = { 2, 3, 4, 10, 40 };
    int last = sizeof(list) / sizeof(list[0]);
    int target = 10;
    int result = binarySearch(list, 0, last - 1, target);
    if(result == -1)
        printf("Element is not present in array");
    else
        printf("Element is present at index: %d",result);
    return 0;
}

```

Output:

Element is present at index: 3

FUNCTIONS:

- The programs we have presented so far have been very simple. They solved problems that could be understood without too much effort.
- As the program size grows and it is common practice to divide the complex program into smaller elementary parts.
- The planning for large programs is simpler. First we must understand the problem as a whole; then break it into simpler and smaller understandable parts. We call each of these parts of a program a module and subdividing a problem into manageable parts **top-down design**.
- The technique used to pass data to a function is known as **parameter passing**.

Functions in C:

- In C, the idea of top-down design is done using functions. A C program is made of one or more functions, but one and only one of which must be named *main*.
- The execution of the program always starts and ends with main, but it can call other functions to do some of the job.
- A function in main including main is an independent module that will be called to do a specific task. A called function receives control from calling function.
- When the called function completes its task, it returns control back to the calling function. It may or may not return a value to the caller.
- The function main is called by the operating system; main in turn calls other functions. When main is complete, control return to the operating system.
- In general, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.

Advantages of using functions in C:

- Problems can be divided into understandable and manageable steps.
- Functions provide a way to reuse code that can be used in more than one place in the program.
- Functions are used to protect data. The local data described in a function is available only to the function and only while the function is executing. Local data in a function cannot be seen or changed by a function outside of its scope.

User-Defined Functions:

- ✓ Like every other object in C, functions must be declared, defined and called when needed.
- ✓ The function declaration needs to be done before the function call, mentions the name of the function, the return type and type and order of the formal parameters. The declaration uses only the header of the function and ended with a semicolon.
- ✓ The function definition can be coded after the function call contains the code needed to complete the task.
- ✓ The function call comes within a function which calls it to get the portion of the job to be done.
- ✓ A function name is used three times: for declaration, in a call and for definition.
- ✓ Example programs:

```
// Simple greetings program using functions
```

```
#include <stdio.h>
void greetings(void);
void main()
{
    greetings();
}

void greetings()
{
    printf("Hello World\n");
}
```

Output:
Hello World

Basic Function Design:

- ✓ The classification of the basic function designs is done by their return values and their parameter list.
- ✓ Functions either return a value or don't. Functions that don't return a value are known as *void* functions.
- ✓ Based on requirement parameter list is also optional. Either they can have parameters or don't.
- ✓ Combining the return values and parameter lists functions can be classified into four basic designs:
 1. void functions without parameters
 2. void functions with parameters
 3. non void functions without parameters
 4. non void functions with parameters

1. void functions without parameters:

- ✓ A void function can be written without any parameters.

- ✓ As the void function does not have a return value, it can be used only as a statement. It cannot be used in an expression.

```
result = greeting(); // error. void function
```

```
// void functions without parameters
```

```
#include <stdio.h>
void add(void);
void main()
{
    add();
}

void add(void)
{
    int a=10,b=20,sum;
    sum=a+b;
    printf("Sum of %d,%d is:%d",a,b,sum);
}
```

Output:

```
Sum of 10,20 is:30
```

2. void function with parameters

- ✓ A void function can have parameters.
- ✓ Parameters are helpful in passing the values from the calling function to the called function. Passing parameters is a technique of sending the values to the called function.
- ✓ As the void function does not have a return value, it can be used only as a statement. It cannot be used in an expression.

```
// void functions with parameters
```

```
#include <stdio.h>
void add(int,int);
void main(void)
{
    int a=10,b=20;
    add(a,b);
}

void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf("Sum of %d,%d is:%d",x,y,sum);
}
```

Output:

```
Sum of 10,20 is:30
```

3. Non void functions without parameters

- ✓ This type of functions returns a value but don't have any parameters.
- ✓ The most common use of this design reads data from the keyboard or file and returns to the calling function.
- ✓ As this function returns a specific type of value, the function call must be initialized with variable of that specific type.

// Non - void functions without parameters

```
#include <stdio.h>
int add(void);
void main(void)
{
    int sum;
    sum=add();
    printf("Sum of a,b is:%d",sum);
}

int add()
{
    int a=10,b=20,sum;
    sum=a+b;
    return sum;
}
```

Output:

Sum of a,b is:30

//Programs for getValue() function

```
#include <stdio.h>
int getValue(void);
void main(void)
{
    int val1,val2;
    val1=getValue();
    val2=getValue();
    printf("Val1:%d, val2:%d",val1,val2);
}

int getValue(void)
{
    int val;
    printf("Enter Value:");
    scanf("%d",&val);
    return val;
}
```

Output:

Enter Value:24

Enter Value:36

Val1:24, val2:36

4. Non-void functions with parameter

- ✓ This type of functions passes parameters and returns a value.
- ✓ As this function returns a specific type of value, the function call must be initialized with variable of that specific type.

//Non-void function with parameters

```
#include <stdio.h>
int add(int,int);
void main(void)
{
    int a=10,b=20,sum;
    sum=add(a,b);
    printf("Sum of a,b is:%d",sum);
}

int add(int x,int y)
{
    int res;
    res=x+y;
    return res;
}
```

Output:

Sum of a,b is:30

User Defined Functions:

- Like every other object in C, functions must be declared, defined and initialized (calling of the function).
- The functions defined by the user who is writing the program, but not by the developers of the language are known as user defined functions.
- To use functions in a C-Program one has to do, the declaration, definition and initialization of the function.
- The function declaration has to be done before function call, which will give the complete picture of the function.
- And definition will succeed function call, which actually contains the statements (body) of the function.

Function Declaration:

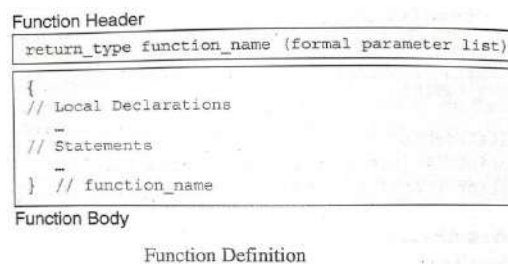
- Function declaration consists of only a function header. It contains no code (body of the function).
- Function declaration consists of three parts: the return type, the function name and the formal parameter list.
- Function declaration will be terminated by a semicolon.
- The return type is the type of value the function is going to return after performing its job. If there is no return type the type will be void type.
- In declaration the parameter list will indicate only the valid type and order of the parameters; the variable identifier can be omitted here in this case.
- Generally the declarations are placed in the global declaration area of the program. That is before the main function.

Function call:

- The function call is postfix expression.
- The operand in the function call is the function name and the operator is the parenthesis set, which contains the actual parameters.
- The actual parameters hold the actual values that are to be sent to the called function. They must match with the function definition's formal parameter list in both type and order. While specifying actual parameters variable identifiers are required.
- If the function call is having multiple parameters to be passed, they must be separated by commas.
- Functions can return values. If a functions return type is void it, then it cannot be used as part of an expression, it has to be called as a stand-alone.
- But if a function is having a return value, then either its call must be initialized with a suitable type variable or it has to be used in an expression. If the function is called like a stand-alone then the return value will be discarded.

Function Definition:

- The function definition contains the code for a function.
- It is made up of two parts: the function header and the function body.



function header

- A function header consists of three parts: the return type, the function name, and the formal parameter list.
- A semicolon is not required at the end of the function definition header.
- The return-type should come before the function name; return-type specifies the type of value a function will return. If the function has nothing to return, the return type will be void.
- The formal parameter list must match with the type and order of the actual parameters. Here while specifying the formal parameters the variable identifier is must, which will hold the values passed by actual parameters.

function body

- The function body contains the local declarations and the function statements.
- The body starts with local definitions that specify the variables needed by the function. After local definition, the function statements, terminating with a return statement.
- If a functions return type is void, it can be written without a return statement.

Formal Parameter List:

- In the definition of a function, the parameters are contained in the formal parameter list.
- This list declares and defines the variables that will contain the data received by the function.

- If the function has no parameters; if it is not receive any data from the calling function, then the parameter list can be void.

Local Variables:

A local variable is a variable that is defined inside a function body and used without having any role in the communication between functions.

//Example program for functions

```
#include<stdio.h>
int add(int,int);
int getValue();
void putValue(int);
void main(void)
{
    int a,b,c;
    a=getvalue();
    b=getvalue();
    c=add(a,b);
    putvalue(c);
}

int add(int x,int y)
{
    int z;
    z=x+y;
    return z;
}

int getvalue()
{
    int temp;
    printf("Enter value:");
    scanf("%d",&temp);
    return temp;
}

void putvalue(int res)
{
    printf("Result: %d",res);
}
```

Output:

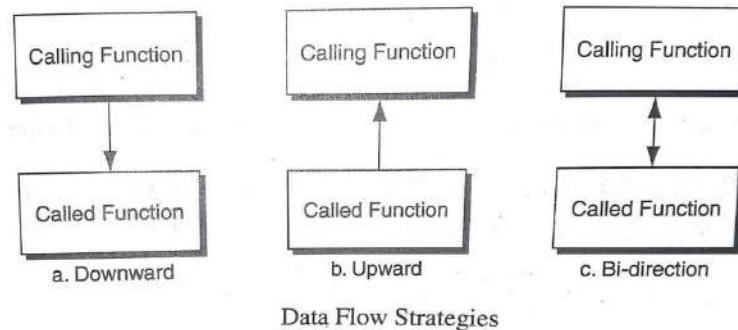
Enter value: 24

Enter value: 32

Result: 56

INTER-FUNCTION COMMUNICATION:

- ✓ Inter function communication is a concept through which calling function and called function will communicate with each other, and exchange the data.
- ✓ The data flow between the called and calling function can be divided into three strategies:
 - I. Downward flow
 - II. Upward flow
 - III. Bi-directional flow



I. Downward flow:

- ✓ In downward communication, the calling function sends data to the called function.
- ✓ No data flows in the opposite directions.
- ✓ The called function may change the values passed, but the original values in the calling function remains untouched.
- ✓ The pass by value or call by value mechanism is a perfect solution for downward flow.
- ✓ A variable is declared and defined in the called function for each value to be received from the calling function.
- ✓ Downward communication is one-way communication. The calling function can send data to the called function, but the called function cannot send any data to the calling function.

//downward communication

```
#include<stdio.h>
void downFun(int,int);
void main(void)
{
    int a=5;
    downFun(a,15);
    printf("Value of a is:%d",a);
}

void downFun(int x,int y)
{
    x=x+y;
}
```

Output:

Value of a is: 5

II. Upward Communication:

- ✓ Upward communication occurs when the called function sends data back to the calling function without receiving any data from it.
- ✓ C provides only one way for upward flow, the return statement.
- ✓ Using return statement only one piece of data item can be returned.
- ✓ To send multiple values back to the calling function, we have to use references of variables. References are memory locations which carries the data from called function to the calling function.

//upward communication

```
#include<stdio.h>
void upFun(int*,int*);
void main(void)
{
    int a,b;
    upFun(&a,&b);
    printf("Value of a, b are:%d,%d",a,b);
}

void upFun(int* aptr,int* bptr)
{
    *aptr = 24;
    *bptr = 32;
}
```

Output:

Value of a, b are: 24,32

III. Bi-direction Communication:

- ✓ Bi-directional communication occurs when the calling function sends data down to the called function. After processing the called function sends data up to the calling.
- ✓ The strategy described for upward communication can also be used for bi-direction communication, but with a little modification.
- ✓ The only change is that the indirect reference must be used in both sides of the assignment statement.
- ✓ With this change the variable in the called function first is accessed for retrieving data using address variable in the right hand side.
- ✓ The same parameter is used again to store a value in the left-hand side.
- ✓ Note: both upward and bidirectional communications are examples for pass by reference or call by reference mechanisms.

//Bi-directional communication:

```
#include<stdio.h>
void biFun(int*,int*);
void main(void)
{
    int a=2,b=3;
    biFun(&a,&b);
    printf("Value of a, b are:%d,%d",a,b);
}
```

```
void biFun(int* aptr,int* bptr)
{
    *aptr = *aptr+24;
    *bptr = *bptr+32;
}
```

Value of a,b are: 26, 35

STANDARD FUNCTIONS (LIBRARY FUNCTIONS):

- ✓ C provides a rich collection of standard functions whose definitions have been written and are ready to be used in our programs.
- ✓ To use these functions, we must include their function declarations into our program.
- ✓ The function declarations are generally found in header files.
- ✓ Instead of adding each declaration separately, a simple statement with header file can be included on top of the program.

Math functions

- ✓ Several library functions are available for mathematical calculations.
- ✓ Most of them are in either math header file (*math.h*) or standard library (*stdlib.h*).

Absolute value functions:

- ✓ These functions will return the absolute value of a number.
- ✓ An absolute value is the positive value of the value regardless of its sign.
- ✓ There are 3-integer and 3-floating point functions.
- ✓ The integer functions are abs, labs, and llabs.
- ✓ For abs the parameter must be an int and it returns an int. For labs the parameter must be a long int, and it returns a long int. For llabs the parameter must be a long long int, and it returns a long long int.
- ✓ *General Syntax:*
 - int abs(int);
 - long labs(long);
 - long long llabs(long long);
- ✓ The real functions are fabs, fabsf and fabsl. For fabs the parameter is a double, and it returns a double. For fabsf the parameter is a float and returns a float. For fabsl the parameter is a long double and returns a long double.
- ✓ *General Syntax:*
 - double fabs(double);
 - float fabsf(float);
 - long double fabsl(long double);

// Absolute value functions

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

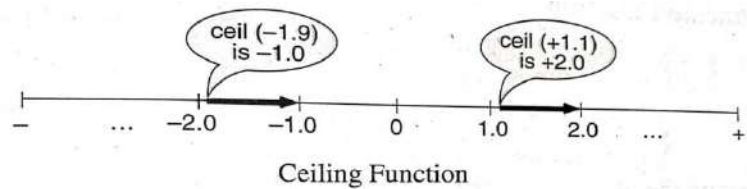
```
{
    float val=-2.674;
    float res;
    res = fabsf(val);
    printf("Absolute value of %f is %f", val, res);
}
```

Output:

Absolute value of -2.674000 is 2.674000

Ceiling Functions

- ✓ A ceiling is the smallest integral value greater than or equal to a number.
Example: Ceiling of 3.00001 is 4.
- ✓ If the complete numbers are considered as continues range from minus infinity to plus infinity, this function moves the number right towards an integral value.



- ✓ The declarations for ceiling function are:
 - double ceil (double);
 - float ceilf(float);
 - long double ceilll(long double);
- ✓ Example: ceil (-1.9) returns -1.0
 ceil (1.1) returns 2.0

// Ceiling Functions

```
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val = 2.674;
    float res;
    res = ceilf(val);
    printf("Ceiling value of %f is %f", val, res);
    val = -2.674;
    res = ceilf(val);
    printf("\nCeiling value of %f is %f", val, res);
}
```

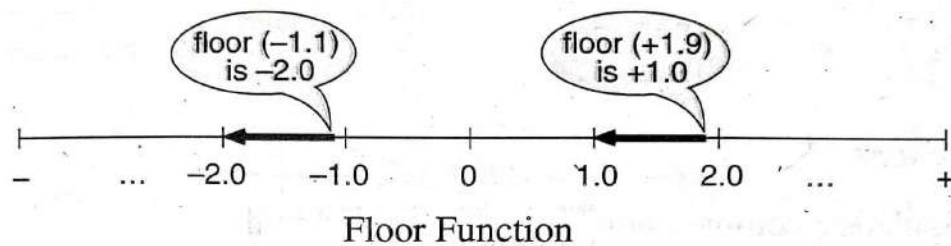
Output:

Ceiling value of 2.674000 is 3.000000

Ceiling value of -2.674000 is -2.000000

Floor Functions

- ✓ A floor is the largest integral value that is equal to or less than a number.
- ✓ Example: Floor of 3.99999 is 3.0.
- ✓ On number series, this function moves the number left towards an integral value.



- ✓ The declarations for floor function are:
 - double floor (double);
 - float floorf (float);
 - long double floorl (long double);
- ✓ Example:
 - floor (-1.1) returns -2.0
 - floor (1.9) returns 1.0

// Floor Functions

```
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val = 2.674;
    float res;
    res = floorf(val);
    printf("Floor value of %f is %f", val, res);
    val = -2.674;
    res = floorf(val);
    printf("\nFloor value of %f is %f", val, res);
}
```

Output:

Floor value of 2.674000 is 2.000000
Floor value of -2.674000 is -3.000000

Truncate Functions

- ✓ The truncate functions return the integral value in the direction of 0.
- ✓ They are same as floor functions for positive numbers and same as ceiling function for negative numbers.
- ✓ Their function declarations are:
 - double trunc (double);
 - float truncf (float);
 - long double trunc (long double);
- ✓ Example:
 - trunc (-1.1) returns -1.0
 - trunc (1.9) returns 1.0

// Truncate Functions

```
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val = 2.674;
    float res;
    res = truncf(val);
    printf("Truncate value of %f is %f", val, res);
    val = -2.674;
    res = truncf(val);
    printf("\nTruncate value of %f is %f", val, res);
}
```

Output:

Truncate value of 2.674000 is 2.000000
Truncate value of -2.674000 is -2.000000

Round Functions

- ✓ The round functions return the nearest integral value.
- ✓ Their function declarations are:
 - double round (double);
 - float roundf(float);
 - long double roundl(long double);
- ✓ Example: round (-1.1) returns -1.0
 round (1.9) returns 2.0

// Round Functions

```
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val = 2.674;
    float res;
    res = roundf(val);
    printf("Round value of %f is %f", val, res);
    val = -2.674;
    res = roundf(val);
    printf("\nRound value of %f is %f", val, res);
}
```

Output:

Round value of 2.674000 is 3.000000
Round value of -2.674000 is -3.000000

Power Functions:

- ✓ The power (pow) function return the value of the x raised to the power of y.
- ✓ An error occurs if the base (x) is negative and the exponent (y) is not an interger, or if the base is zero and the exponent is not positive.
- ✓ The power function declarations are:

- double pow (double, double);
- float powf (float, float);
- long double powlf (long double, long double);

✓ Example: pow (3.0, 4.0) returns 81.0

Square Root Functions:

- ✓ The square root functions return the non-negative square root of a number.
- ✓ An error occurs if the number is negative.
- ✓ The square root function declarations are:
 - double sqrt (double);
 - float sqrtf (float);
 - long double sqrtlf(long double);

✓ Example: sqrt (25) returns 5.0

// Power and Square root functions

```
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val1 = 3;
    float val2 = 4;
    float res1, res2;
    res1 = powf(val1,val2);
    printf("%f power of %f is %f", val1, val2, res1);
    res2 = sqrtf(res1);
    printf("\nSquare root of %f is %f", res1, res2);
}
```

Output:

3.000000 power of 4.000000 is 81.000000
 Square root of 81.000000 is 9.000000

PASSING ARRAY TO FUNCTIONS

- ✓ Arrays can also be passed as parameters to functions when required.
- ✓ There are two possibilities in which arrays can be passed:
 - a) Passing individual elements
 - b) Passing the whole array

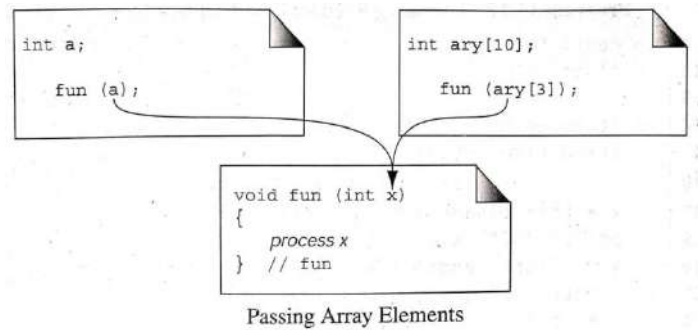
Passing Individual elements

Individual elements can be either by their values or by their addresses.

Passing data values

- ✓ Individual data values of arrays can be passed using their index along with array name.

- ✓ As long as the type is matching, the called function is unaware whether the value is coming from an array or a normal variable.



// Passing data values from an array

```
#include<stdio.h>
void arrAccess(int);
void main(void)
{
    int a[5]={2,4,3,6,9};
    arrAccess(a[3]);
    printf("Array element after function call: %d",a[3]);
}

void arrAccess(int temp)
{
    printf("Parameter received from calling function: %d",temp);
    temp = 36;
}
```

Output:

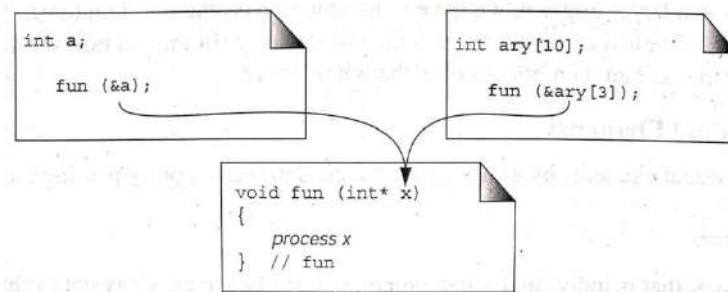
Parameter received from calling function: 6
Array element after function call: 6

Passing Addresses

- ✓ The address of the elements of the array can also be passed to the function.
- ✓ The individual element addresses can be passed just like any other variable address.
- ✓ To pass an array element's address, the address operator can be prefixed to the element's indexed reference.
- ✓ Example: The address of 4th element in the array can be passed in following way: `&arr[3];`

Passing an address of an array element requires two changes in the called function.

- ✓ First, it must declare that it is receiving an address.
- ✓ Second, it must use the indirection operator (*) to refer to the elements value.



Passing the Address of an Array Element

// Passing the address of an array element

```
#include<stdio.h>
void arrAccess(int*);
void main(void)
{
    int a[5]={2,4,3,6,9};
    arrAccess(&a[3]);
    printf("Array element after function call: %d",a[3]);
}

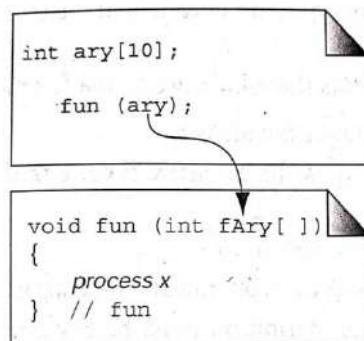
void arrAccess(int* temp)
{
    printf("Parameter received from calling function: %d",*temp);
    *temp = 36;
}
```

Output:

Parameter received from calling function: 6
Array element after function call: 36

Passing the whole array

- ✓ In C, the name of an array holds the address of the first element in the array.
- ✓ Using array name and index values the complete array elements can be accessed.
- ✓ Passing the array name instead of a single element, allows the called function to refer to the same array back in the calling function.



Passing the Whole Array

//Passing whole array to called function

```
#include<stdio.h>
void arrAccess(int[]);
void main(void)
{
    int a[5]={2,4,3,6,9},i;
    arrAccess(a);
    printf("\nAfter function call Elements in Array:");
    for(i=0;i<5;i++)
        printf("%d ", a[i]);
}

void arrAccess(int arrTemp[])
{
    int i;
    printf("Elements in Array:");
    for(i=0;i<5;i++)
        printf("%d ", arrTemp[i]);
    for(i=0;i<5;i++)
        arrTemp[i] = i;
}
```

Output:

Elements in Array:2 4 3 6 9

After function call Elements in Array:0 1 2 3 4

Unit - IV

RECURSION

- ✓ Programmers use two approaches for writing repetitive algorithms: One approach uses loops and the other uses recursion.
- ✓ Recursion is a repetitive process in which a function calls itself.
- ✓ Some older languages do not support recursion. One major language that does not support recursion is COBOL.
- ✓ To study the concept of recursion let us consider the example of finding factorial of a given number.
- ✓ In our study we'll see both iterative definition and recursive definition for finding the factorial.

Iterative Definition

- ✓ The factorial of a number is the product of the integral values from 1 to the given number.
- ✓ The iterative definition can be shown as:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) \dots 3 * 2 * 1 & \text{if } n > 0 \end{cases}$$

Iterative Factorial Definition

- ✓ An iterative function can be defined which will take only the iteration counter to calculate the factorial but not the function call.

Example: $\text{factorial}(4) = 4 * 3 * 2 * 1 = 24$

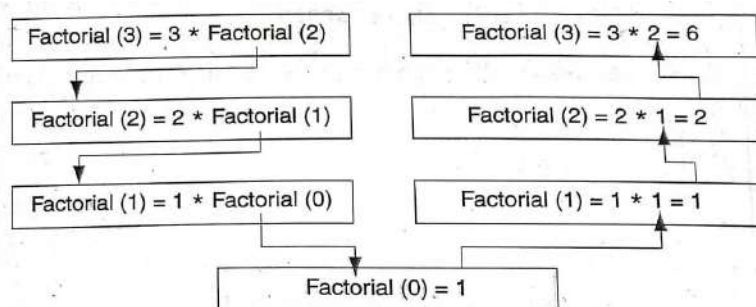
Recursive Definition

- ✓ A function is defined recursively whenever the function itself appears within the definition itself.
- ✓ The recursive definition of factorial function can be shown as:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

Recursive Factorial Definition

- ✓ The decomposition of factorial (3), using the above formula is shown in the following formula.



Factorial (3) Recursively

- ✓ Observe it carefully; the recursive solution for a problem involves a two-way journey: the decomposition will be from top to bottom, and then we solve it from bottom to the top.
- ✓ Every recursive call must either solve part of the problem or reduce the size of the problem.
- ✓ The recursive calculation looks more difficult when using pen and paper, but it is often much easier and more elegant solution when a computer is used.
- ✓ Every recursive solution to the problem will have a base-case and general-case. A base-case will provide the solution to the problem and all other cases will be known as general-cases.
- ✓ In the factorial example the base-case is factorial of (0) and all other cases are general-cases.

Iterative solution to factorial problem

```
//Iterative solution to factorial problem
```

```
#include<stdio.h>
int factorial(int);
void main(void)
{
    int n,res;
    printf("Enter any number:");
    scanf("%d",&n);
    res=factorial(n);
    printf("Factorial of %d is: %d", n, res);
}

int factorial(int num)
{
    int i, fact = 1;
    for(i=1;i<=num;i++)
        fact=fact*i;
    return fact;
}
```

Output:

```
Enter any number:5
Factorial of 5 is: 120
```

Recursive solution to factorial problem

```
//Recursive solution to factorial problem
```

```
#include<stdio.h>
int factorial(int);
void main(void)
{
    int n,res;
    printf("Enter any number:");
    scanf("%d",&n);
    res=factorial(n);
    printf("Factorial of %d is: %d", n, res);
}

int factorial(int num)
{
```

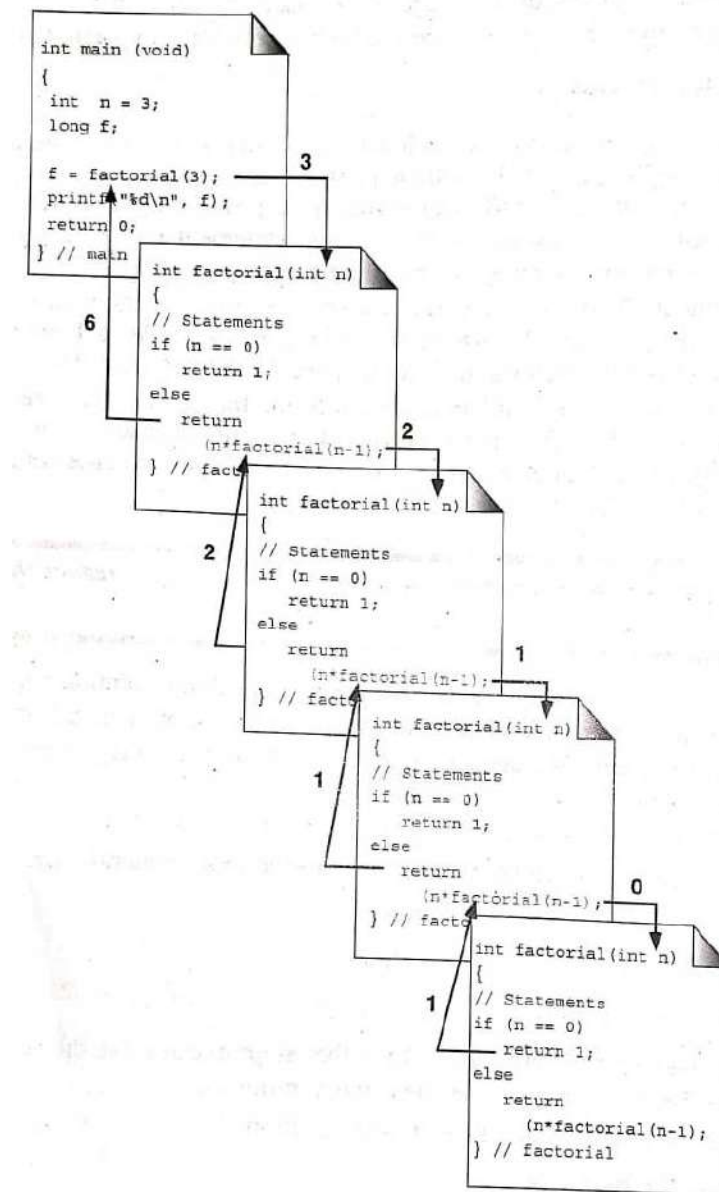
```

if(num == 0)
    return 1;
else
    return (num * factorial(num-1));
}

```

Output:

Enter any number:5
 Factorial of 5 is: 120



Calling a Recursive Function

Fibonacci Numbers

- ✓ Finding the Fibonacci numbers is another example for recursion.
- ✓ Fibonacci numbers are a series in which each number is the sum of the previous two numbers. This number series is named after an Italian mathematician Leonardo Fibonacci of thirteenth century.
 Example: First few numbers in Fibonacci series are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

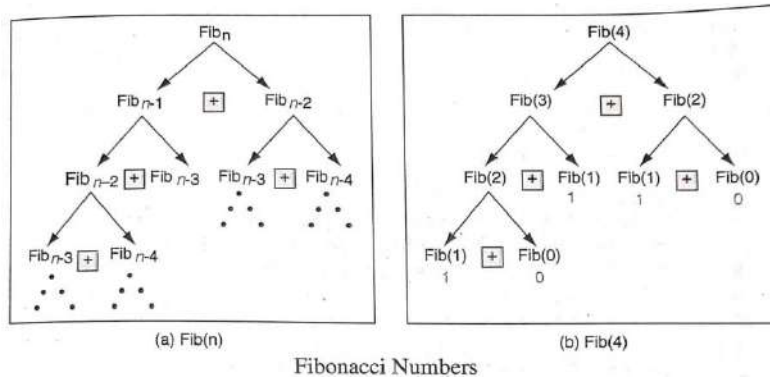
- ✓ To start the series always first two numbers will be 0 and 1.
- ✓ The generalized statements for Fibonacci series are as follows:

Given: $\text{fibonacci}_0 = 0$;

$\text{fibonacci}_1 = 1$;

Then: $\text{fibonacci}_n = \text{fibonacci}_{n-1} + \text{fibonacci}_{n-2}$

- ✓ To calculate the fibonacci4 the steps are shown in the following figure:



Iterative solution to factorial problem

//Iterative solution to Fibonacci Series Problem

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

    printf("Enter the number of terms: ");
    scanf("%d", &n);
    fibonacci(n);
}

void fibonacci(int n)
{
    int i, term1 = 0, term2 = 1, nextTerm;
    for (i = 1; i <= n; ++i)
    {
        printf("%d, ", term1);
        nextTerm = term1 + term2;
        term1 = term2;
        term2 = nextTerm;
    }
}
```

Output:

Enter the number of terms: 6

0, 1, 1, 2, 3, 5,

Recursive solution to factorial problem

//Recursive solution to factorial problem

```
#include <stdio.h>
int fibonacci(int);
void main(void)
{
    int nterms, fib = 0, i;
    printf("Enter the number of terms:");
    scanf("%d", &nterms);
    printf("Fibonacci series terms are:\n");
    for (i = 0; i < nterms; i++)
    {
        printf("%d, ", fibonacci(fib));
        fib++;
    }
}

int fibonacci(int n)
{
    if(n == 0 || n == 1)
        return n;
    return (fibonacci(n - 1) + fibonacci(n - 2));
}
```

Output:

Enter the number of terms:6
Fibonacci series terms are:
0, 1, 1, 2, 3, 5,

Finding GCD (Greatest Common Divisor):

- ✓ The HCF (Highest Common Factor) or GCD (Greatest Common Divisor) of two integers is the largest integer that can exactly divide both numbers (without a remainder).

Iterative solution to GCD problem

//Iterative solution to GCD problem

```
#include <stdio.h>
void main(void)
{
    int n1, n2, i, gcd;

    printf("Enter two integers: ");
    scanf("%d %d", &n1, &n2);

    for(i=1; i <= n1 && i <= n2; i++)
    {
        // Checks if i is factor of both integers
        if(n1%i==0 && n2%i==0)
            gcd = i;
    }
}
```

```
printf("G.C.D of %d and %d is %d", n1, n2, gcd);  
}
```

Output:

Enter two integers: 5 6
G.C.D of 5 and 6 is 1

Recursive solution to GCD problem**// Recursive solution to GCD problem**

```
#include <stdio.h>  
int gcd(int, int);  
void main(void)  
{  
    int n1, n2;  
    printf("Enter two integers: ");  
    scanf("%d %d", &n1, &n2);  
  
    printf("G.C.D of %d and %d is %d.", n1, n2, gcd(n1,n2));  
    return 0;  
}  
  
int gcd(int n1, int n2)  
{  
    if (n2 != 0)  
        return gcd(n2, n1%n2);  
    else  
        return n1;  
}
```

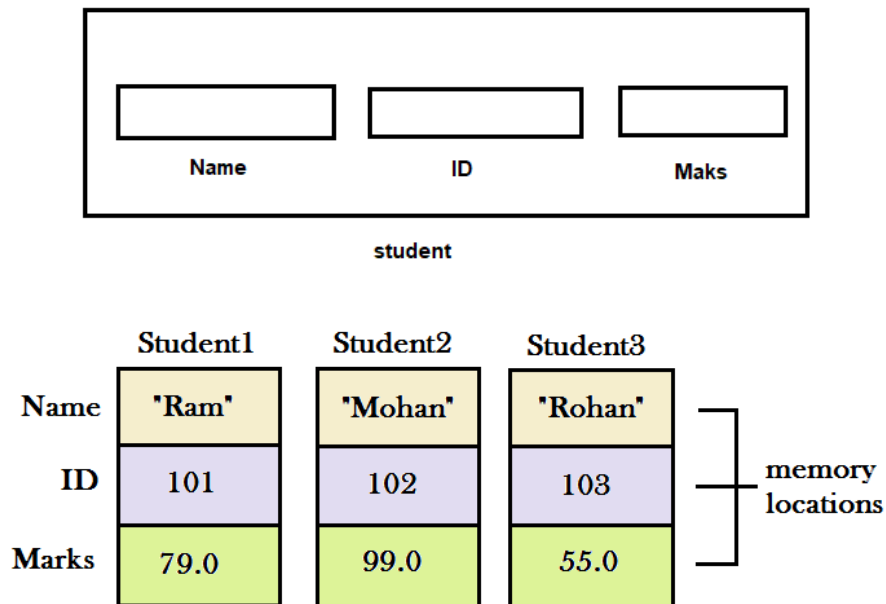
Output:

Enter two integers: 6 42
G.C.D of 6 and 42 is 6.

STRUCTURE

- ✓ A structure is a collection of heterogeneous data elements.
- ✓ A structure in C is a collection of items of different types.
- ✓ A structure is a collection of logically related elements, possibly of different types.

Example:



- ✓ Each element in a structure is called a field. A field is the smallest element of named data that has meaning. It has many of the characteristics of the variables. It has a type, an identifier and it also exists in the memory. It can be initialized a value, and later can be used for manipulations. The only difference between the variable and a structure field is that, structure field is a part of the structure.
- ✓ The difference between an array and a structure is that, all elements in an array must be of same type, but the elements in a structure can be of same or different types.
- ✓ A structure is a convenient way in which data are grouped into one single name. It provides flexibility and an increased control over accessing the data when it needed. Using structures data can be accessed as a single grouped object or individual elements can be accessed.

Structure Type Declaration:

- ✓ Like every other element in C, a structure must be declared and defined.
- ✓ In C we have two ways to declare a structure:
 1. Tagged Structures
 2. Type-defined structures

1. Tagged Structures:

- ✓ The first way to define a structure is using tagged structure.
- ✓ A tagged structure can be used to define variables, parameters to functions and return types of functions.
- ✓ A tagged structure starts with the keyword struct, followed by the TAG. The TAG is the identifier for the structure; it is used for accessing the fields of the structure.
- ✓ The format is:

Syntax:	Example:
<pre>Struct TAG { Filed list; };</pre>	<pre>Struct STUDENT { char name[20]; char ID[10]; int marks; };</pre>
<p>Note: Notice that the structure is concluded with a semicolon, no variables are defined so no associated storage. It is just a template for the structure.</p>	

2. Type-defined structures (using typedef):

It is a more powerful way of declaring a structure. By using the keyword *typedef* the structure will be declared.

The format is:

Syntax:	Example:
<pre>typedef struct { Filed list; }TYPE;</pre>	<pre>typedef struct { char name[20]; char ID[10]; int marks; }STUDENT;</pre>

- ✓ The type-defined structure differs from the tagged declarations in two ways:
 - First, the keyword, type-def, is added in the beginning of the declaration.
 - Second, an identifier is required at the end of the block. The identifier will be the structure type name.

Structure Variable declaration:

- ✓ After a structure has been declared, a variable of structure type can created using it.
- ✓ The structure type can be declared in the global area of the program, to make it visible to all functions.
- ✓ The structure variables are declared in the local declarations.

Tagged Structure	Type defined structure
<pre>// Global Type declarations struct STUDENT { char name[20]; char ID[20]; int marks; }; //Local declarations struct STUDENT aStudent;</pre>	<pre>// Global Type declarations typedef struct { char name[20]; char ID[20]; int marks; }STUDENT; //Local declarations STUDENT aStudent;</pre>

//Demonstration of Tagged Structure

```
#include<stdio.h>
struct NODE
{
    int data;
    char ch;
};
void main(void)
{
    struct NODE node;
    node.data = 20;
    node.ch = 'c';
    printf("Data = %d",node.data);
    printf("\nCharacter = %c",node.ch);
}
```

Output:

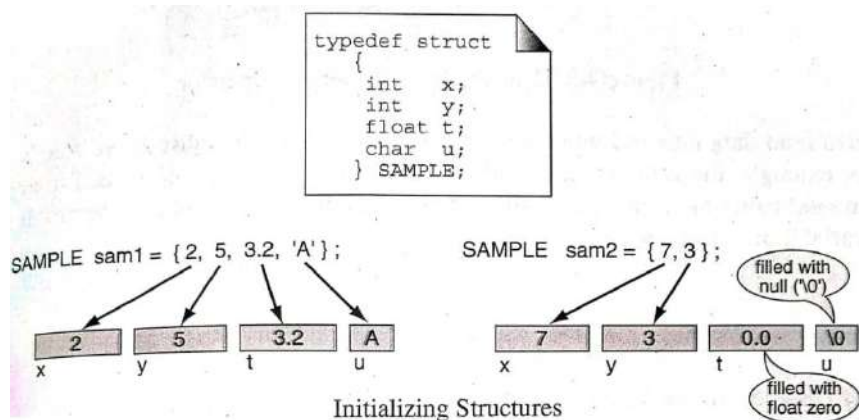
Data = 20

Character = c

Initialization

- ✓ The rules for structure initialization are similar to the rules for array initialization.
- ✓ The initializing values are enclosed in braces and separated by commas.
- ✓ They must match the corresponding types in the structure definition.

Examples:



Accessing Structures:

- ✓ The elements in a structure can be accessed individually or the complete structure can be assigned as a whole.

Referencing Individual Fields:

- ✓ Structure fields can be accessed and manipulated using operators just like any other normal variables.
- ✓ To refer to a field in a structure the structure and the field both have to be referred.
- ✓ C uses a hierarchical naming convention that first uses the structure variable identifier and then the field identifier.

- ✓ The structure variable identifier is separated from the field identifier by a dot.
- ✓ The dot is known as the direct selection operator.

Example1:	Example2:
aStudent.ID aStudent.name aStudent.marks	SAMPLE sam1={2,5,3.2,'A'}; scanf("%d %d %f %c", &sam1.x, &sam1.y, &sam1.t, &sam1.u); printf("%d %d %f %c", sam1.x, sam1.y, sam1.t, sam1.u);

//Example program on structures

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
    float t;
    char u;
}SAMPLE;

void main(void)
{
    //declaring, defining and initializing the structure variable sam1
    SAMPLE sam1 = {5,4,3.2,'A'};
    SAMPLE sam2;
    printf("sam1 contents:\n");
    printf("%d %d %f %c", sam1.x,sam1.y,sam1.t,sam1.u);
    printf("\nEnter contents for sam2:\n");
    scanf("%d %d %f %c",&sam2.x,&sam2.y,&sam2.t,&sam2.u);
    //Modifying sam1
    sam1.x=45;
    sam1.y=32;
    sam1.t=5.4;
    sam1.u='d';
    printf("\nsam1 contents after modification:");
    printf("%d %d %f %c", sam1.x,sam1.y,sam1.t,sam1.u);
    printf("\nsam2 contents after modification:");
    printf("%d %d %f %c", sam2.x,sam2.y,sam2.t,sam2.u);
}
```

Output:

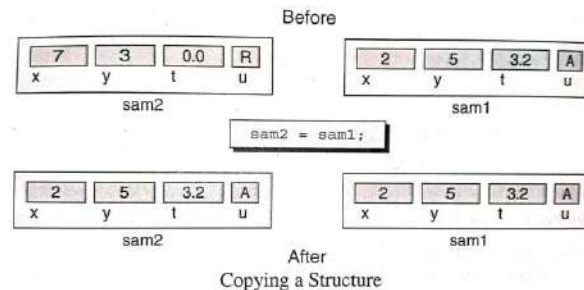
sam1 contents:
5 4 3.200000 A

Enter contents for sam2:
3 5 2.45 s

sam1 contents after modification:45 32 5.400000 d
sam2 contents after modification:3 5 2.450000 s

Operations on Structures

- ✓ Only one operation is allowed on the structure that is the assignment of a structure to another structure.
- ✓ A structure can only be copied to another structure of the same type using the assignment operator.
- ✓ When one structure needs to be copied to another structure, instead of assigning the individual members to another structure's members, the whole structure can be initialized to another structure.



//Assignment operation between structure variables

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
    float t;
    char u;
}SAMPLE;

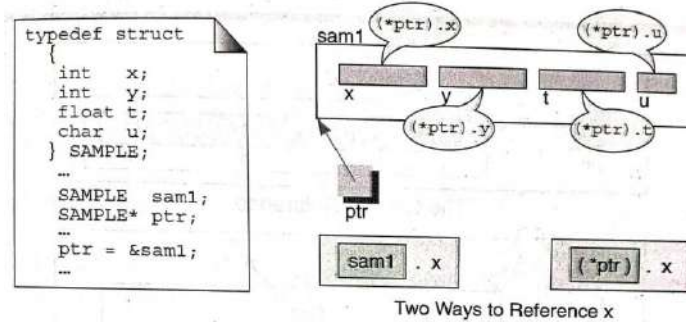
void main(void)
{
    //declaring, defining and initializing the structure variable sam1
    SAMPLE sam1 = {5,4,3.2,'A'};
    SAMPLE sam2;
    sam2 = sam1;
    printf("sam1 contents:\n");
    printf("%d %d %f %c", sam1.x,sam1.y,sam1.t,sam1.u);
    printf("\nsam2 contents:\n");
    printf("%d %d %f %c", sam2.x,sam2.y,sam2.t,sam2.u);
}
```

Output:

```
sam1 contents:
5 4 3.200000 A
sam2 contents:
5 4 3.200000 A
```

Pointer to structure:

- ✓ Structures can also be accessed through pointers.
- ✓ Using pointers is one of the most common methods used to reference structures.
- ✓ The `SAMPLE` structure can be used with pointers in the following manner:



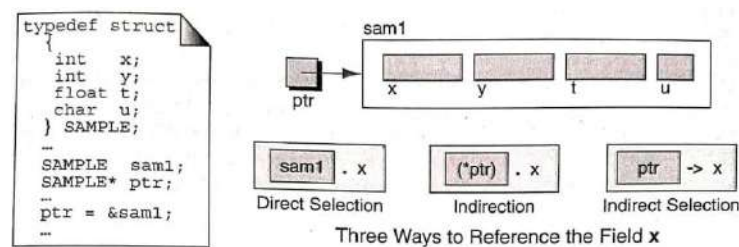
Pointers to Structures

```
SAMPLE* ptr;
Ptr = &sam1;
*ptr // refers to the whole structure
(*ptr).x (*ptr).y (*ptr).t (*ptr).u //Individual field reference
```

- ✓ The parenthesis in the above example are absolutely required, without the parenthesis the compiler will consider `*ptr.x` as `*(ptr.x)`. C applies the dot (.) operator first and then the asterisk operator next.
- ✓ The reason the parenthesis are needed is that the precedence of the direct selection operator (.) is higher than the indirection operator (*).
- ✓ Here `*ptr.x` will be interpreted as `*(ptr.x)` which is wrong. `*(ptr.x)` means a completely different undefined structure called `ptr` that contains a member `x`, which must be a pointer. Since this is not true, a compile time error will be generated.

Indirect selection operator:

- ✓ To avoid the above confusion (to eliminate the problem with pointers to structures) a special operator known as indirect selection operator is used.
- ✓ The symbol used for indirect selection operator is an arrow formed by minus sign and the greater than symbol (->).
- ✓ The indirection operator is placed between the pointer identifier of structure and the member to be referenced.



Indirect Selection Operator

//Example program for indirect selection operator

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
    float t;
    char u;
```

```
}SAMPLE;
```

```
void main(void)
```

```
{
```

```
//declaring, defining and initializing the structure variable sam1
```

```
SAMPLE sam1 = {5,4,3.2,'A'};
```

```
SAMPLE* strPtr;
```

```
strPtr = &sam1;
```

```
printf("Accessing sam1 contents through structure variable and direct selection operator (dot(.)):\n");
```

```
printf("%d %d %f %c", sam1.x,sam1.y,sam1.t,sam1.u);
```

```
printf("\nAccessing sam1 contents through pointer to structure and direct selection operator (dot(.)):\n");
```

```
printf("%d %d %f %c", (*strPtr).x,(*strPtr).y,(*strPtr).t,(*strPtr).u);
```

```
printf("\nAccessing sam1 contents through pointer to structure and indirect selection operator (->):\n");
```

```
printf("%d %d %f %c", strPtr->x,strPtr->y,strPtr->t,strPtr->u);
```

```
}
```

Output:

Accessing sam1 contents through structure variable and direct selection operator (dot(.)):

5 4 3.200000 A

Accessing sam1 contents through pointer to structure and direct selection operator (dot(.)):

5 4 3.200000 A

Accessing sam1 contents through pointer to structure and indirect selection operator (->):

5 4 3.200000 A

COMPLEX STRUCTURES

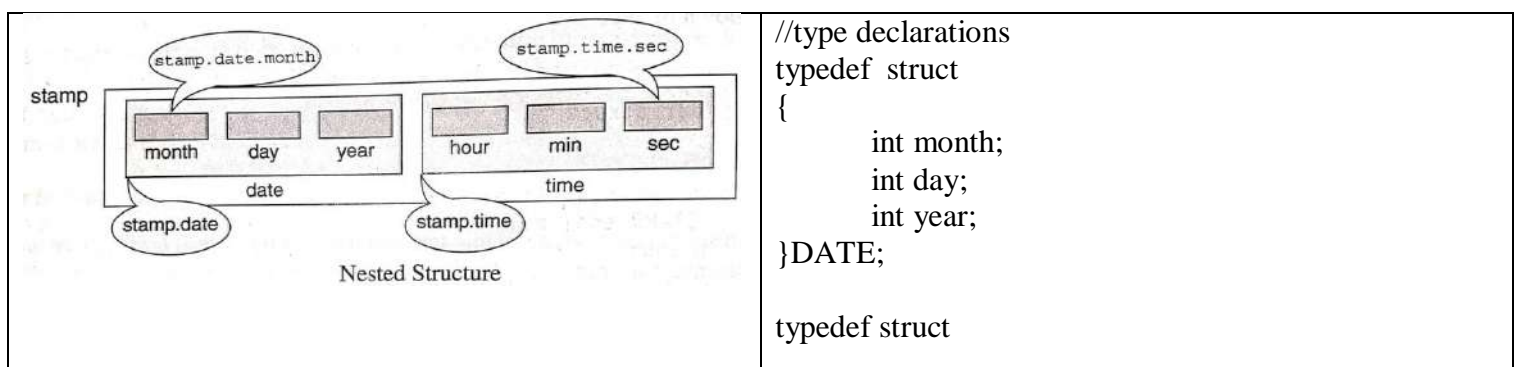
- ✓ Structures within structures (nested structure), arrays within structure, and arrays of structures are some of the complex forms of structures.

- i. Nested Structures
- ii. Structures containing Arrays
- iii. Structures containing pointers
- iv. Array of structures

i. Nested Structures:

- A structure can have another structure as a member.
- When a structure includes another structure it is nested structure.

Example:



```
// A more complex form
typedef struct
{
    STAMP startTime;
    STAMP endTime;
}JOB;

//variable declaration
JOB job;
```

```
{
    int hour;
    int min;
    int sec;
}TIME;

typedef struct
{
    DATE date;
    TIME time;
}STAMP;

//variable declarations
STAMP stamp;
```

Referencing Nested structures:

- ✓ To access fields of nested structures the direct referencing operator dot(.) is used from highest level to inner component level.

```
stamp
stamp.date
stamp.date.month
stamp.date.day
stamp.date.year
```

```
stamp
stamp.time
stamp.time.hour
stamp.time.min
stamp.time.sec
```

```
job.startTime.time.hour
job.endTime.time.hour
```

Nested Structure Initialization:

```
STAMP stamp = {{05, 10, 2019}, {23, 45, 01}};
```

```
// Nested Structure Program
```

```
#include<stdio.h>
typedef struct
{
    char* month;
    int day;
    int Year;
}DATE;

typedef struct
{
    int hour;
    int min;
    int sec;
}TIME;
```



```

typedef struct
{
    DATE date;
    TIME time;
}TIMESTAMP;

void main(void)
{
    TIMESTAMP tstamp = {{05, 10, 2019}, {23, 45, 01}};
    printf("Event Date (mm:dd:yyyy): %d / %d / %d", tstamp.date.month, tstamp.date.day, tstamp.date.Year);
    printf("\nEvent Time (hh:mm:ss): %d : %d : %d", tstamp.time.hour, tstamp.time.min, tstamp.time.sec);
}

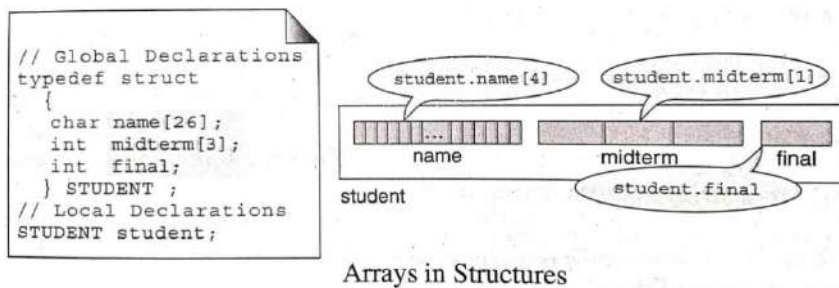
```

Output:

Event Date (mm:dd:yyyy): 5 / 10 / 2019

Event Time (hh:mm:ss): 23 : 45 : 1

ii. Structures containing arrays



```
STUDENT student = {"John", {20,18,17},80};
```

// Arrays within structures example

```

#include<stdio.h>
typedef struct
{
    char name[20];
    int midterm[3];
    int final;
}STUDENT;

void main(void)
{
    STUDENT student = {"John", {18, 17, 15}};
    student.final = student.midterm[0]+student.midterm[1]+student.midterm[2];
    printf("Student Name: %s",student.name);
    printf("\nStudent midterm Marks: %d %d %d", student.midterm[0], student.midterm[1], student.midterm[2]);
    printf("\nStudent Final Marks: %d", student.final);
}

```

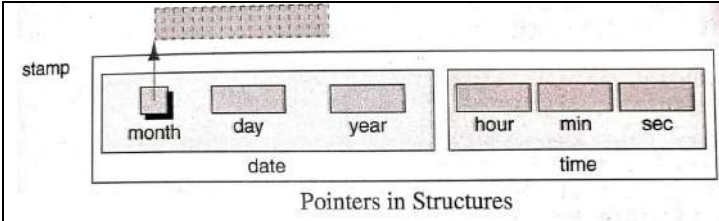
Output:

Student Name: John

Student midterm Marks: 18 17 15

Student Final Marks: 50

iii. Structures containing pointers:

 <p>Pointers in Structures</p>	<pre>typedef struct { char* month; int day; int Year; }DATE; typedef struct { DATE date; }STAMP; //variable declarations STAMP tstamp; //Initialization stamp.date.month = may;</pre>
----------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Structures containing pointers

```
#include<stdio.h>
typedef struct
{
    char* month;
    int day;
    int year;
}DATE;

typedef struct
{
    DATE date;
}TIMESTAMP;

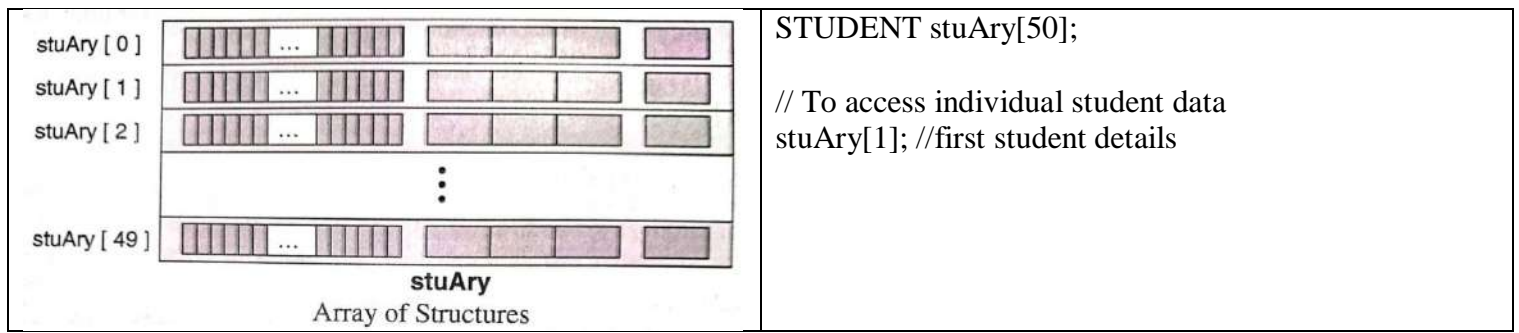
void main(void)
{
    TIMESTAMP tStamp = {"May", 9, 2019};
    printf("Event DATE: %dth %s - %d", tStamp.date.day, tStamp.date.month, tStamp.date.year);
}
```

Output:

Event DATE: 9th May - 2019

iv. Array of Structures:

- ✓ In many situations array of structures can be created.
- ✓ Example: To manage a group of students, an array of student's structure can be created.
- ✓ By putting the data into an array it will become more flexible and easy to work with the data, to calculate sum and average for instance.
- ✓ To create an array of 50 students the following example can be used:



//Array of structures Program

```
#include<stdio.h>
typedef struct
{
    char    name[20];
    int     rollno;
    int     midmarks[3];
}STUDENT;

void main(void)
{
    STUDENT std[5];
    int i;

    printf("Enter information of students:\n");

    // storing information
    for(i=0; i<5; i++)
    {
        std[i].rollno = i+1;

        printf("\nFor roll number: %d,\n",std[i].rollno);

        printf("Enter name: ");
        scanf("%s",std[i].name);

        printf("Enter 3 Mid Marks: ");
        scanf("%d %d %d",&std[i].midmarks[0], &std[i].midmarks[1], &std[i].midmarks[2]);

        printf("\n");
    }

    printf("Displaying Information:\n\n");
    // displaying information
    for(i=0; i<5; i++)
    {
        printf("\nRoll number: %d\n",std[i].rollno);
        printf("Name: ");
        puts(std[i].name);
        printf("Marks: %d %d %d",std[i].midmarks[0], std[i].midmarks[1], std[i].midmarks[2]);
        printf("\n");
    }
}
```

```
}  
}
```

Output:

Enter information of students:

For roll number: 1

Enter name: John

Enter 3 Mid Marks: 12 13 15

For roll number: 2

Enter name: Asif

Enter 3 Mid Marks: 15 17 19

Displaying Information:

Roll number: 1

Name: John

Marks: 12 13 15

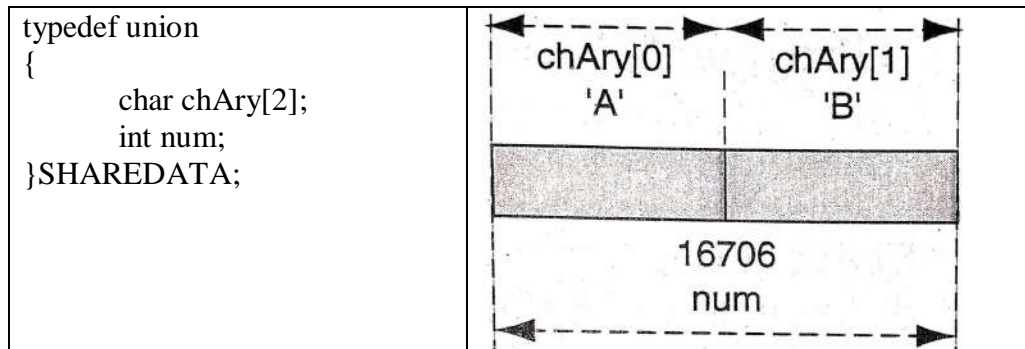
Roll number: 2

Name: Asif

Marks: 15 17 19

UNIONS

- ✓ A union is similar to a structure, as it is also a collection of elements of different types.
- ✓ A union is a constructs that allows memory to be shared by different types of data. In simple terms the integer memory locations can be used to store two characters. The union basically works on the same principle.
- ✓ The union follows same format and syntax as structure. The only difference between declaration of structure and union is that, the keyword union is used when declaring a union.



Referencing Unions

- ✓ The rules for referencing a union are similar to those of a structure.
- ✓ To reference individual fields within a union, the direct selection operator (dot) can be used.

```
SHAREDATA shareData;
shareData.num
shareData.chAry[0]
shareData.chAry[1]
```

- ✓ When a union is being referenced through a pointer, the indirect selection operator (→) can be used.

```
SHAREDATA* ptrShareData;
ptrShareData→num;
ptrShareData→chAry[0];
ptrShareData→chAry[1];
```

Initializing a Union:

- ✓ Only the first type declared in the union can be initialized when the union variable is defined. The other types can be initialized by assigning values or reading values into the union.
- ✓ When initializing a union, the value must be enclosed in a set of braces, even if there is only one value.

//Demonstration of Unions

```
#include<stdio.h>
typedef union
{
    int num;
    char chAry[2];
}SHAREDATA;
```

```
void main(void)
{
    SHAREDATA shareData;
    shareData.num = 16706;
    printf("Integer Value: %d", shareData.num);
    shareData.chAry[0] = 'X';
    shareData.chAry[1] = 'Y';
    printf("\nValues from character array: %c %c",shareData.chAry[0], shareData.chAry[1]);
    printf("\nInteger Value: %d", shareData.num);
}
```

Output:

Integer Value: 16706

Values from character array: X Y

Integer Value: 22872

POINTERS:

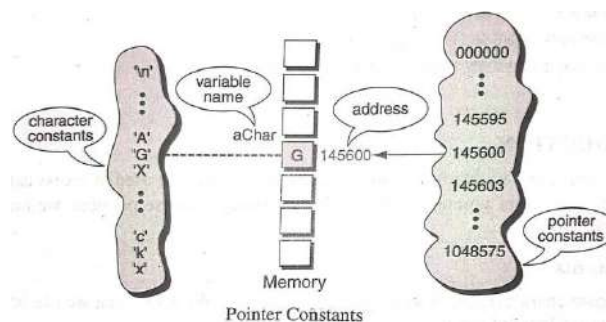
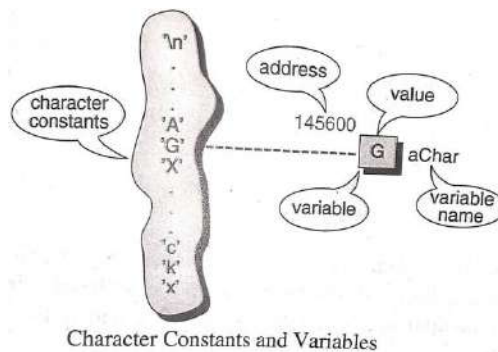
- ✓ Every computer has addressable memory locations.
- ✓ The data manipulations so far we did with the help of variable names (identifiers). We assigned an identifier to the data location and manipulated the content with the help of it.
- ✓ Pointers have many uses in C:
 - They are one of an efficient way of accessing the data for manipulations.
 - They also provide efficient techniques for manipulating data in arrays.
 - They are used in functions as pass by address parameter.
 - They are the basis for dynamic allocation of memory.

Introduction:

- ✓ A pointer is a constant or variable that contains an address that can be used to access data.
- ✓ Pointers are built on the basis of pointer constants.

Pointer constant:

- ✓ A pointer constant is an address value that specifically represents a memory location in the computers main memory.
- ✓ Like every other constant in computer world a character constant cannot be changed.
- ✓ Every time a program runs all variables in the program will get different addresses, this is because in modern operating systems can put a program in memory wherever it feels convenient.
- ✓ Pointer constants are drawn from the set of addresses of a computer. They exist during the run time of a computer. We cannot change them but we can only use them.



- ✓ Being existed in the computer a pointer constant can be retrieved and saved inside a variable.
- ✓ The address operator (&) extracts the address of a variable. Syntax: `&variable_name`
- ✓ To print the address of a variable the conversion code `%p` can be used.

```
//Printing the address of varriables
```

```
#include<stdio.h>
void main(void)
{
    char a, b;
    printf("%p %p", &a, &b);
}
```

Output:

142300

142301

Pointer Variables:

- ✓ A pointer constant can be saved in a pointer variable, and later can be used to access the value.
- ✓ The type of a pointer must be specific to the type of the variable to which it is going to point. That is an integer type of variables address can be saved into an integer type of pointer.
- ✓ To store the address of a variable we need a special type of variable that is Pointer variable.
- ✓ A single variables address can be saved into multiple pointers, so that we'll have more number of pointers pointing to the same location, and it is possible in C.
- ✓ If a Pointer variable not pointing to anywhere, that is if we want to make sure that a pointer is not pointing to any address, C provides a special null pointer (NULL) in the standard Input/Output stdio.h header file for this purpose.

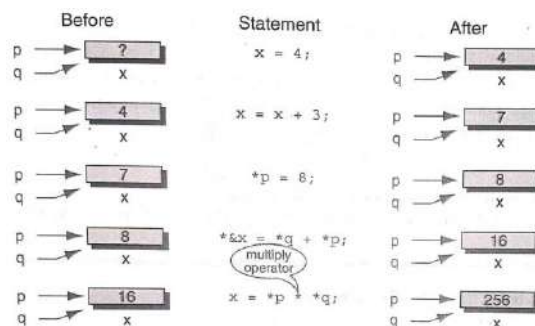
Accessing Variables through pointers:

- ✓ The indirection operator (*) can be used to dereference the pointer's address.
- ✓ The indirection operator is a unary operator whose operand must be a pointer value. The result is, a pointed variables value that can be used for inspection or manipulation.
- ✓ To access the variables a through pointer p, simply code *p.

To add 1 to the variable a's content any of the following ways can be used

a++	a=a+1	*p=*p+1	(*p)++
-----	-------	---------	--------

- ✓ The indirection and address operators are the inverse of each other, and when combined in an expression they cancel each other. $*\&x \leftarrow \text{SAME} \rightarrow x$

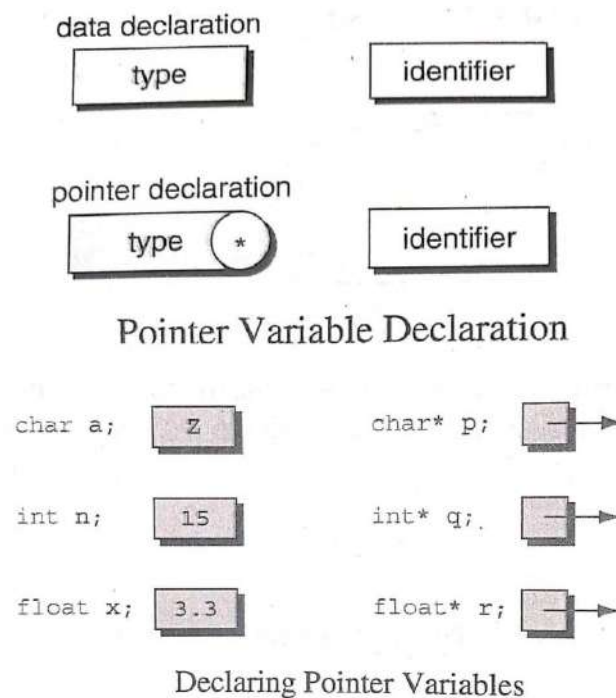


Accessing Variables Through Pointers

Pointer Declaration and Definition:

- ✓ To declare a pointer variable the asterisk symbol (*) can be used along with type specification.

- ✓ The following figure shows the declaration and definition of different pointer variables and their corresponding data variable declarations.



//Demonstrate use of pointers

```
#include<stdio.h>
void main(void)
{
    int a;
    int* ptr;
    a = 4;
    ptr = &a;
    printf("%d %p\n", a, &a);
    printf("%d %p", *ptr, ptr);
}
```

Output:

```
4 0x7ffd93455594
4 0x7ffd93455594
```

Declaration Vs Redirection:

- ✓ The asterisk operator can be used in two different contexts: for declaration and for redirection.
- ✓ When asterisk is used for declaration, it is associated with a type. Example: `int* ptr;` will declare an integer pointer.
- ✓ When asterisk is used for redirection, it is associated with a pointer variable to get the data value of the variable whose address is stored in that pointer. Example: `sum = *aptr + *bptr;`

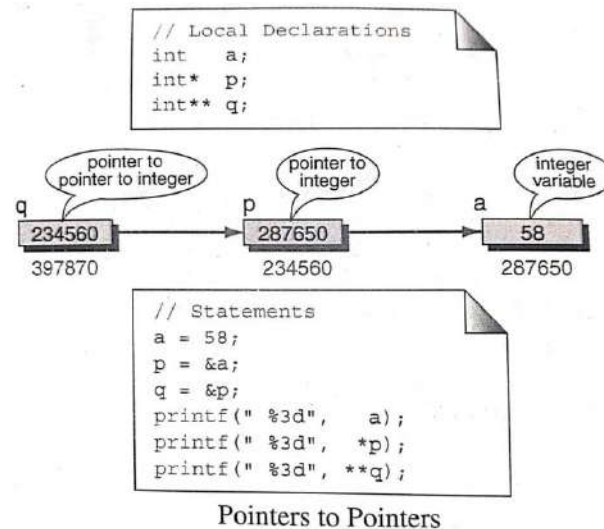
Initialization of Pointer variables:

- ✓ Like every other object in C, pointers is also must be initialized, before they can be used in the program.
- ✓ As variables in our program without initialization will have an unknown garbage value in them. The same is true for pointers.

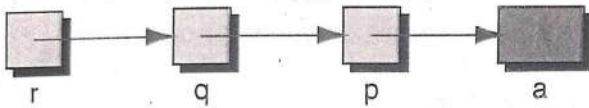
- ✓ When program starts all uninitialized pointers will hold some unknown address which are pointing to some unknown values.

Pointers to Pointers

- ✓ In C, it is possible to use pointers that points to other pointers. That is, we can have a pointer pointing to a pointer to an integer.
- ✓ The following figure demonstrates this two level indirection:
- ✓ Each level of pointer indirection requires a separate indirection operator when it is dereferenced.



//demonstration on Using Pointers to Pointers



Using Pointers to Pointers

```
#include<stdio.h>
void main(void)
{
    int a;
    int* p;
    int** q;
    int*** r;
    p = &a;
    q = &p;
    r = &q;
    printf("Enter a value:");
    scanf("%d",&a);
    printf("a's value is:%d",a);

    printf("\nEnter a value:");
    scanf("%d",p);
    printf("a's value: %d %d", a, *p);

    printf("\nEnter a value:");
    scanf("%d",*q);
```

```
printf("a's value: %d %d %d", a, *p, **q);

printf("\nEnter a value:");
scanf("%d",&r);
printf("a's value: %d %d %d %d", a, *p, **q, ***r);
}
```

Output:

```
Enter a value:4
a's value is:4
Enter a value:5
a's value: 5 5
Enter a value:6
a's value: 6 6 6
Enter a value:7
a's value: 7 7 7 7
```

Pointer to void (void*)

- ✓ A pointer to void is a generic type that is not associated with a reference type, that is it is not the address of a character, an integer, a floating point or any other type.
- ✓ It is compatible for assignment purpose only with all other types.
- ✓ A pointer of any reference type can be assigned to a pointer of void type and a pointer to void type can be assigned to any reference type.
- ✓ As void pointer has no specific object type, it cannot be dereferenced unless it is cast.
- ✓ The following program demonstrate the usage of void pointer in programming:

// Demonstration of void pointer – Generic Pointer

```
#include<stdio.h>
void main(void)
{
    int a = 20;
    char c = 'A';
    void* ptr;
    ptr = &a;
    //printf("a=%d",*ptr); //dereference of void pointer is not allowed, generated an error
    printf("a=%d",*(int*)ptr);
    ptr = &c;
    printf("\nc=%c",*(char*)ptr);
}
```

Output:

```
a=20
c=A
```

Self-Referential Structures:

- ✓ Self-Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.
- ✓ In simple words, structures pointing to the same type of structures are self-referential in nature.
- ✓ A self-Referential structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own type.
- ✓ See the following declaration of a tagged structure:

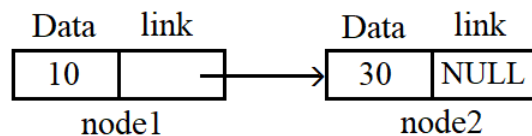
```
struct NODE
{
    int data;
    struct NODE* link;
};
```

- ✓ The above illustrated structure declaration describes one node that comprises of two logical segments. One of them stores data and the other one is a pointer indicating where the next component can be found. .
- ✓ Several such inter-connected nodes create a chain of structures known as linked list.

Linked list Notation:

- ✓ The above declaration of structure can be used to create a chain of nodes, where each node is going to hold some data and a link to another node.
- ✓ Several of such nodes can be created and connected to form a data-structure known as linked list.
- ✓ The above node is having only one data element and a link to the one other node of the same type. As the node is having only a single linking pointer between the nodes, we can also call the data structure as single linked list.
- ✓ The following program demonstrate the concept of single linked list:
- ✓ A linked linked list is a chain of structures where each node points to the next node to create a list. To keep track of the starting node's address a dedicated pointer (referred as start pointer) is used. The end of the list is indicated by a NULL pointer. In order to create a linked list of integers, we define each of its element (referred as node) using the following declaration
- ✓ Different operations that can be possible on a linked list are: C
 - Creating a Linked List.
 - Displaying its contents.
 - Inserting an element into a list.
 - Deleting an existing element.

//Demonstration of Single Linked List



Linked List with two nodes

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

```
struct NODE {
    int data;
```

```

    struct NODE* link;
};

void main(void)
{
    struct NODE node1; // Node1

    // Initialization
    node1.link = NULL;
    node1.data = 10;

    struct NODE node2; // Node2

    // Initialization
    node2.link = NULL;
    node2.data = 30;

    // Linking node1 and node2
    node1.link = &node2;

    printf("Node1 data: %d", node1.data);
    // Accessing data member of node2 using node1
    printf("\nNode2 data: %d", node1.link->data);
}

```

Output:

Node1 data: 10

Node2 data: 30

- ✓ Self-referential structures are very useful in applications that involve linked data structures, such as linked lists.
- ✓ Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted.
- ✓ Operations like insertion or deletion of nodes in a self-referential structure involve simple and straightforward alteration of pointers.

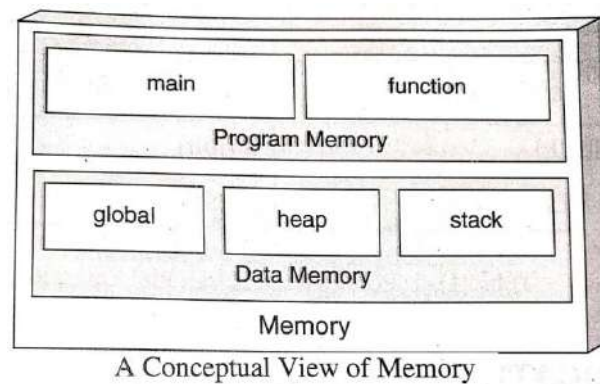
MEMORY ALLOCATION FUNCTIONS:

In C, there are two ways to reserve memory locations for variables.

1. Static memory allocation
2. Dynamic memory allocation

Memory Usage:

- ✓ Memory allocated to a program is divided into program memory and data memory.
- ✓ Program memory consists of memory used for main and all called functions.
- ✓ Data memory consists of permanent definitions of global and local data and constants, dynamic data memory.
- ✓ main() function must be in memory all the time. Each called function must be in memory only when it is called and becomes active.
- ✓ If a function called more than once, the copies of the local variables of this function will be maintained inside stack memory.
- ✓ In addition to the stack memory, a separate memory area known as heap is also available. Heap memory is unused memory allocated to the program and available to be assigned during execution. It is the memory pool from which the memory is allocated when requested by the memory allocation functions.

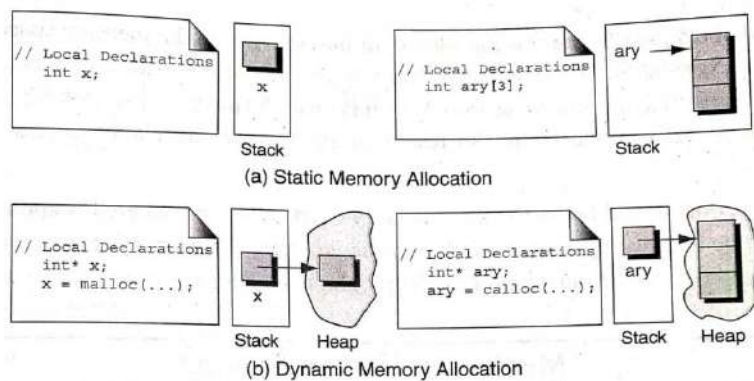


Static Memory Allocation:

- ✓ Static memory allocation requires that the declaration and definition of memory be fully specified in the source program.
- ✓ The number of bytes reserved cannot be changed during run time of the program.
- ✓ So far we used this technique for reserving memory for variables, arrays and pointers.

Dynamic Memory Allocation:

- ✓ Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running.
- ✓ Unlike static memory allocations, dynamic memory allocations have no identifier associated with it. It can be accessed only through an address. That is to access data in dynamic memory we need a pointer.



Memory Allocation Functions

- ✓ Four memory management functions are used with dynamic memory.
- ✓ Three of them malloc, calloc and realloc are used to allocate the memory.
- ✓ The fourth one free is used to return the memory when it is no longer needed.
- ✓ All these memory management functions are found in the standard library file (stdlib.h).

Block Memory allocation (malloc())

- ✓ The *malloc()* function allocates a block of memory that contains the number of bytes specified in its parameter.
- ✓ The allocated memory is not initialized and contains unknown values.
Syntax: `void* malloc(int size);`
- ✓ On successful allocation it returns a void pointer to the first byte of the allocated memory. If the allocation of memory is unsuccessful the function will return a NULL pointer.
- ✓ Calling a *malloc* function with a zero size is a problem, the results are unpredictable. It may return a NULL pointer or it may return some other value. Never call a malloc with a zero size.

//Demonstration of malloc function

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

void main(void)
{
    int* ptr;
    int n, i, sum = 0;

    printf("Enter the number of elements you need from heap:");
    scanf("%d",&n);
    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        printf("Memory successfully allocated using malloc.\n");
        printf("Enter %d elements into the heap:",n);
```

```

for (i = 0; i < n; i++) {
    scanf("%d",&ptr[i]);
}

// Print the elements from the heap
printf("The elements in heap are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}
}
free(ptr);
}

```

Output:

Enter the number of elements you need from heap:4

Memory successfully allocated using malloc.

Enter 4 elements into the heap:5

6

7

6

The elements in heap are: 5, 6, 7, 6,

Contiguous Memory allocation (calloc())

- ✓ The *calloc()* function is primarily used to allocate memory for arrays.
- ✓ It differs from *malloc* only in that; it sets memory to null characters.
Syntax: void* calloc(int element_count, int element_size);
- ✓ On successful allocation it returns a void pointer to the first byte of the allocated memory. If the allocation of memory is unsuccessful the function will return a NULL pointer.

//Demonstration of calloc funciton

```

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

void main(void)
{
    int* ptr;
    int n, i, sum = 0;

    printf("Enter the number of elements you need from heap:");
    scanf("%d",&n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        printf("Memory successfully allocated using calloc.\n");
    }
}

```



```

printf("Enter %d elements into the heap:",n);

for (i = 0; i < n; i++) {
    scanf("%d",&ptr[i]);
}

// Print the elements from the heap
printf("The elements in heap are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}
}
free(ptr);
}

```

Output:

Enter the number of elements you need from heap:4
Memory successfully allocated using calloc.
Enter 4 elements into the heap:5
6
7
6
The elements in heap are: 5, 6, 7, 6,

Reallocation of memory (realloc())

- ✓ The *realloc()* function is advised to be used with much care. As a minor issue in the usage will lead to the crash of the program.
 - ✓ Given a pointer to a previously allocated block of memory, *realloc* changes the size of the block by deleting or extending the memory at the end of the block.
 - ✓ If the memory cannot be extended because of other allocations, *realloc* allocates a completely new block, copies the existing memory locations to the new allocations, and deletes the old locations.
- Syntax: void* realloc(void* ptr, int newSize);

// //Demonstration of realloc function

```

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

void main(void)
{
    int* ptr;
    int n, i, sum = 0;

    printf("Enter the number of elements you need from heap:");
    scanf("%d",&n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
    }
}

```

```

    exit(0);
}
else {
    printf("Memory successfully allocated using calloc.\n");
    printf("Enter %d elements into the heap:",n);

    for (i = 0; i < n; i++) {
        scanf("%d",&ptr[i]);
    }

    // Print the elements from the heap
    printf("The elements in heap are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
}
printf("\nEnter the new size of heap:");
scanf("%d",&n);
// Dynamically allocate memory using realloc()
ptr = (int*)realloc((void*)ptr,n * sizeof(int));

// Check if the memory has been successfully allocated by realloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {
    printf("\nMemory successfully allocated using realloc.\n");
    printf("Enter %d elements into the heap:",n);

    for (i = 0; i < n; i++) {
        scanf("%d",&ptr[i]);
    }

    // Print the elements from the heap
    printf("\nThe elements in heap are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
}

free(ptr);
}

```

Output:

Enter the number of elements you need from heap:2

Memory successfully allocated using calloc.

Enter 2 elements into the heap:4

2

The elements in heap are: 4, 2,

Enter the new size of heap:4

Memory successfully allocated using realloc.

Enter 4 elements into the heap:2

3

4

5

The elements in heap are: 2, 3, 4, 5,

Releasing Memory (free())

- ✓ When memory locations allocated by *malloc*, *calloc* or *realloc* are no longer needed, they should be freed using the *free()* function.
- ✓ It is an error to free memory with a null pointer, a pointer to other than the first element of an allocated block, a pointer that is different type than the pointer that allocated the memory.
- ✓ It is also an error to refer to memory after it has been released.
Syntax: void free(void* ptr);

INTRODUCTION TO FILES:

- ✓ A file is an external collection of related data treated as a single unit. The primary purpose of a file is to keep a record of data.
- ✓ As the contents of primary memory are not permanent, and they will be lost when the computer is shut down, we need files to store the data permanently.
- ✓ Sometimes the collection of data will become too large to be maintained in main memory; in this case files can be used to store the data in secondary memory.
- ✓ Files are stored permanently in auxiliary or secondary memory the most common secondary storage devices are hard disk, CD, DVD and tape.
- ✓ When the computer reads, the data move from the external device to memory and when it writes the data move from memory to the external device.
- ✓ When a file is processed for data input, the data is read from the file upto the end of file (EOF). End file is the terminator of the file which indicates the completion of data processing in the file.
- ✓ The EOF will be automatically read and supplied to the program and it is the responsibility of the programmer to keep track of it.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. If you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

There are two types of files:

1. Text files
2. Binary files

1. Text files

- ✓ Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.
- ✓ When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
- ✓ They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

- ✓ Binary files are mostly the .bin files in your computer.
- ✓ Instead of storing data in plain text, they store it in the binary form (0's and 1's).
- ✓ They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations

In C, you can perform four major operations on the file, either text or binary:

- ✓ Creating a new file
- ✓ Opening an existing file
- ✓ Closing a file
- ✓ Reading from and writing information to a file

Working with files

- ✓ When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

Syntax: FILE *fptr;

Opening a file - for creation and edit

- ✓ Opening a file is performed using the library function in the "stdio.h" header file: fopen().
- ✓ The syntax for opening a file in standard I/O is: ptr = fopen("filename", "mode")
- ✓ **For Example:** fopen("E:\\cprogram\\newprogram.txt", "w");
- ✓ Let's suppose the file newprogram.txt doesn't exist in the location E:\\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.
- ✓ The writing mode allows you to create and edit (overwrite) the contents of the file.

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.

Closing a File

- ✓ The file should be closed after reading/writing.
- ✓ Closing a file is performed using library function fclose().

- ✓ Syntax: `fclose(fptr);` // `fptr` is the file pointer associated with file to be closed.

Reading and writing to a text file

- ✓ For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.
- ✓ They are just the file versions of `printf()` and `scanf()`.
- ✓ The only difference is that, `fprint` and `fscanf` expects a pointer to the structure `FILE`.

Example 1: Write to a text file using `fprintf()`

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

void main(void)
{
    int num;
    FILE *fptr;
    fptr = fopen("C:\\program.txt", "w");

    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter num: ");
    scanf("%d", &num);

    fprintf(fptr, "%d", num);
    fclose(fptr);
}
```

Output:

- ✓ This program takes a number from user and stores in the file `program.txt`.
- ✓ After you compile and run this program, you can see a text file `program.txt` created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file using `fscanf()`

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

void main(void)
{
    int num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt", "r")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }
}
```

```
fscanf(fptr,"%d", &num);

printf("Value of n=%d", num);
fclose(fptr);
}
```

Output:

- ✓ This program reads the integer present in the program.txt file and prints it onto the screen.
- ✓ If you successfully created the file from Example 1, running this program will get you the integer you entered.

Getting data using fseek()

- ✓ If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.
- ✓ This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().
- ✓ As the name suggests, fseek() seeks the cursor to the given record in the file.
- ✓ Syntax: void fseek(FILE * stream, long int offset, int whence)
 - The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different Whence in fseek	
Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example 3: fseek()

```
// C Program to demonstrate the use of fseek()
#include <stdio.h>

void main(void)
{
    FILE *fp;
    fp = fopen("test.txt", "r");

    // Moving pointer to end
    fseek(fp, 0, SEEK_END);

    // Printing position of pointer
    printf("%ld", ftell(fp));
}
```

test.txt

Output:

"Someone over there is calling you. we are going for work. take care of yourself."	81
------------------------------------------------------------------------------------------	----

ftell() in C with example

- ✓ ftell() in C is used to find out the position of file pointer in the file with respect to starting of the file.
- ✓ Syntax of ftell() is: long ftell(FILE *pointer).
- ✓ Consider below C program.
- ✓ When the fscanf statement is executed word "Someone" is stored in string and the pointer is moved beyond "Someone". Therefore ftell(fp) returns 7 as length of "someone" is 6.

// C program to demonstrate use of ftell()
<pre>#include<stdio.h> void main(void) { /* Opening file in read mode */ FILE *fp = fopen("test.txt","r"); /* Reading first string */ char string[20]; fscanf(fp,"%s",string); /* Printing position of file pointer */ printf("%ld", ftell(fp)); }</pre>
Output: 7

rewind() in C with example

- ✓ The rewind function sets the file position indicator (file marker) to the beginning of the file irrespective of its current location.
- ✓ General Syntax: void rewind(FILE *fp);
- ✓ It is equivalent to: void fseek(stream, 0L, SEEK_SET);

<pre>#include <stdio.h> int main () { char str[] = "This is tutorialspoint.com"; FILE *fp; int ch; /* First let's write some content in the file */ fp = fopen("file.txt" , "w"); fwrite(str , 1 , sizeof(str) , fp); fclose(fp); }</pre>

```

fp = fopen( "file.txt" , "r" );
while(1) {
    ch = fgetc(fp);
    if( feof(fp) ) {
        break ;
    }
    printf("%c", ch);
}
rewind(fp);
printf("\n");
while(1) {
    ch = fgetc(fp);
    if( feof(fp) ) {
        break ;
    }
    printf("%c", ch);

}
fclose(fp);

return(0);
}

```

Output:

This is ISL Engineering College.
This is ISL Engineering College.

file.txt

This is ISL Engineering College.