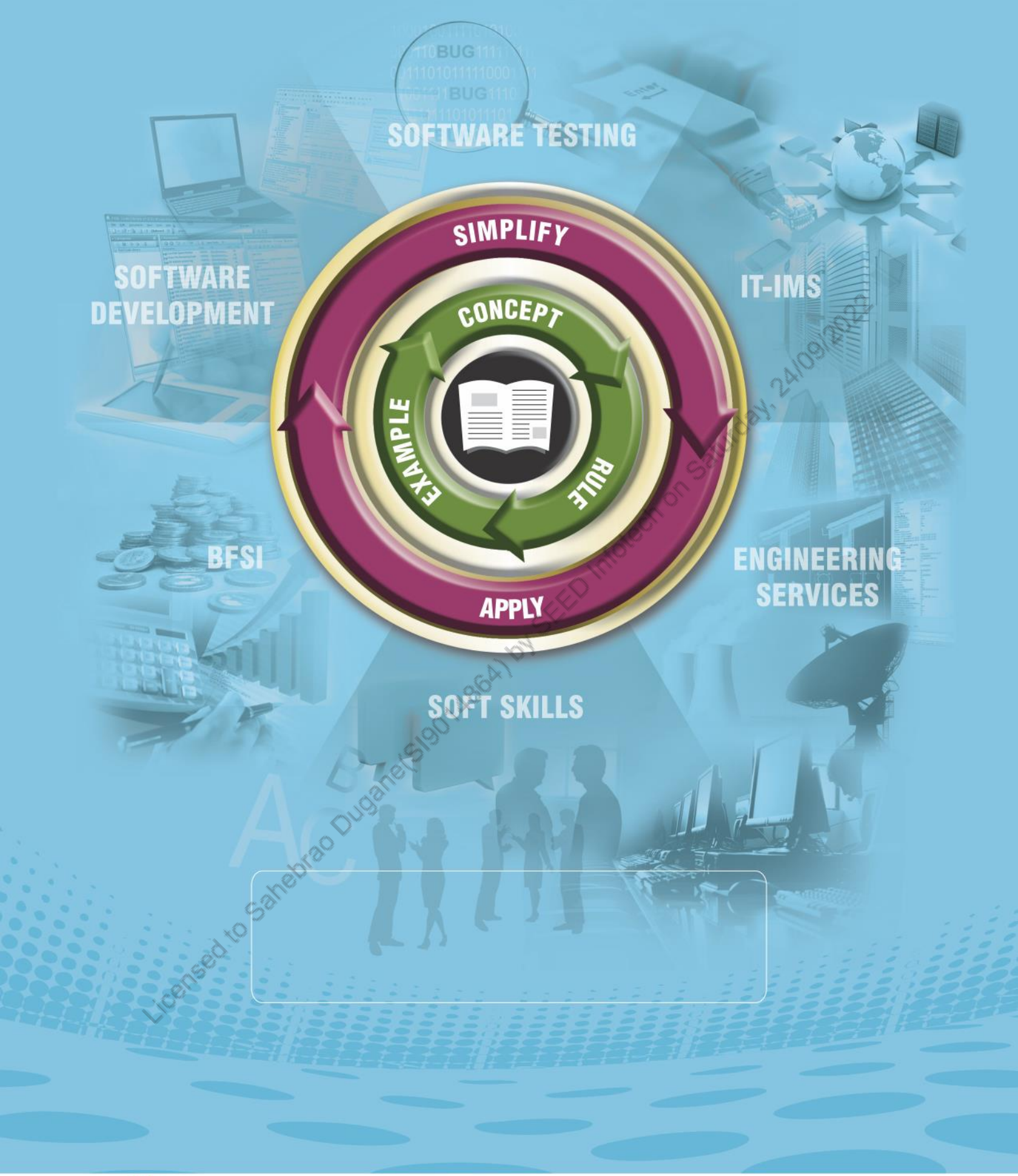


N·S·D·C National Skill Development Corporation
Transforming the skill landscape
Training Partner

seed
Official Curriculum

Advanced ‘C’ Programming

Student Book and Lab Manual

Module Code: [M-ILT-Obj-00061-AdvC-I-En]

Author: Ms Anagha Walvekar

This edition has been printed and published in house by Avani Publications.

Print Edition: July 2015

Issue No./ Date: 01/Nov. 14, 2011 Revision No. & Date: 02/August 16, 2013

Avani Publications

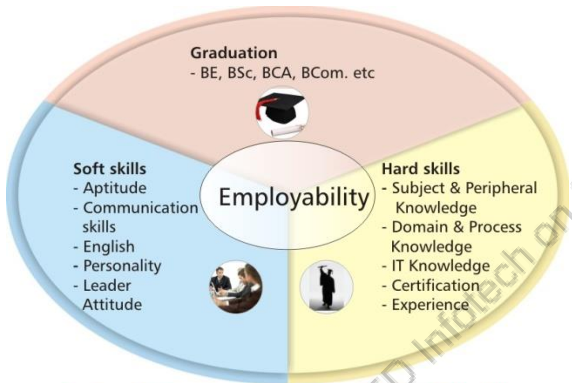
15A-1/2/4, Anandmayee Apartment, Off Karve Road, Pune 411004

Contents

Enhancing Employability

Employability depends on the knowledge, skills and abilities that individuals possess, the way they use these to solve problems and contribute to the growth of the employing organization and the society. Employability skills are those skills that are necessary for getting, keeping and doing well in a job.

Employability can be defined as an equation.



Employability = Graduation + Hard skills + Soft skills

Promise of this book is to help strengthen your "C programming skill" which can be classified under **Hard Skills** of employability equation.

Why learn 'C' programming?

'C' programming is a very popular programming language that beginners learn. Individuals seeking to make a career in software take the first step by learning the 'C' language. Following are the reasons:

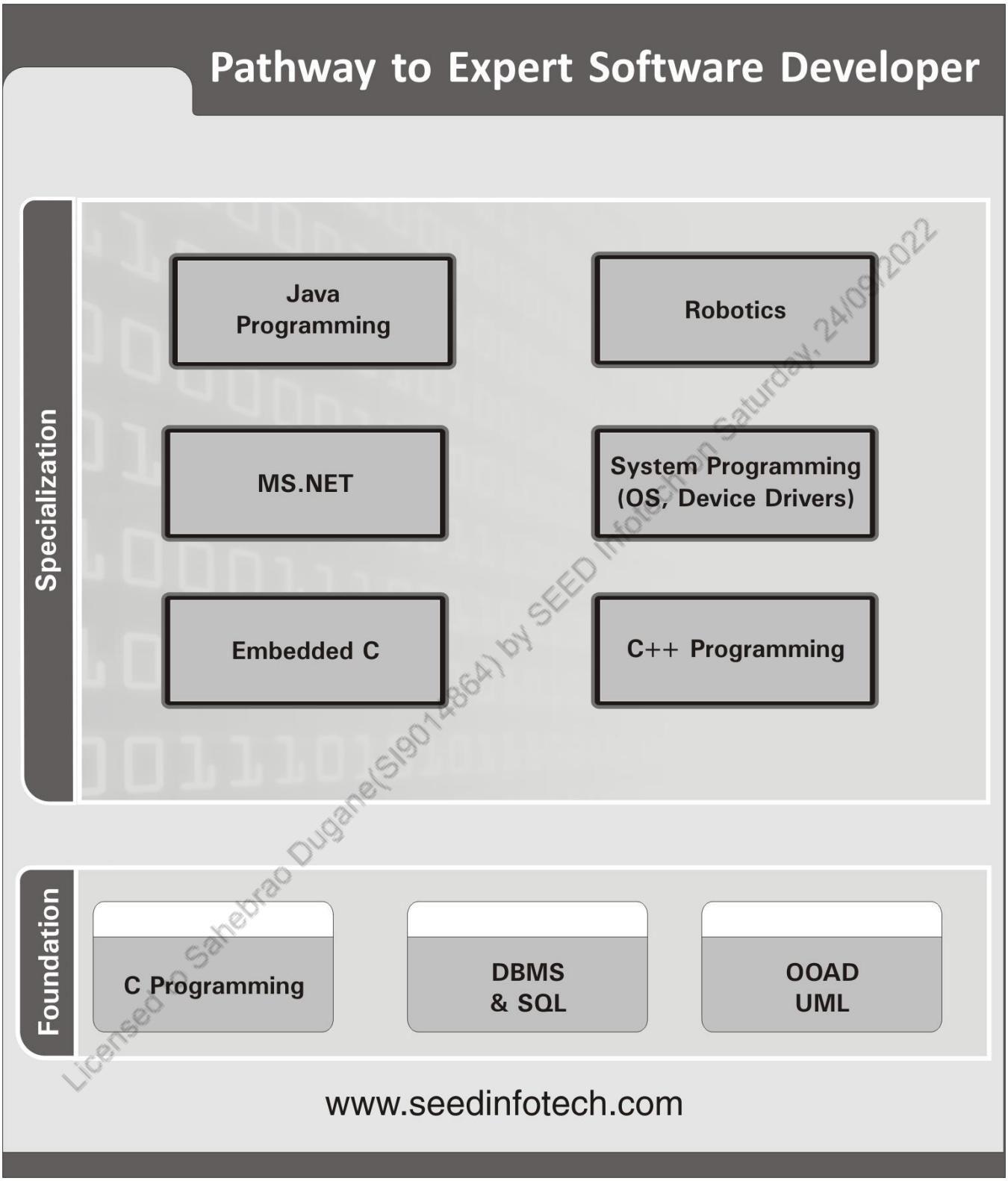
a) Learning C programming builds problem solving ability through programs.
b) Higher level languages like C++, Java, C# have similar looking syntax as of C.
c) C is part of most of the college curriculums.
d) C programming is used for developing embedded systems, device drivers etc.
e) Most of the assessment tests for placement and entrance exams are based on 'C' language.

Role of this book

This book is a step by step guide to master C programming language skills and would aid and reinforce the learning in the classroom. To master any programming language one needs hands-on practice along with clarity of concepts. This student book combined with lab manual is designed to serve the purpose.

The smart tips embedded in the form of Tech App, Interview Tip, Additional Reading, Best Practices, etc. would help increase your curiosity and also help you to become expert with knowledge of peripheral concepts.

We have strived hard to make technical contents of this book error free. However, some mistakes might have slipped our attention. We request the reader to send us any such errors that are sighted which will help us in improving this book further. Your issues or suggestions are welcome on email at ***product-issues@seedinfotech.com***



“Beyond Obvious” Icons

In the student book, we have included special icons (Beyond Obvious Icons) in the form of footnotes that are interspersed in the study material to give you the precise context of the concept you are learning. As you get accustomed to this way of learning, you will enjoy the fun of it which will make this learning highly productive!

This icon indicates a particular best practice which is followed while developing the applications, in design, in coding etc. Knowledge of best practices makes one a good developer or designer.

This icon indicates the points which are important from your technical interview. Before interview, you may visit these small tips as quick revision pointers.

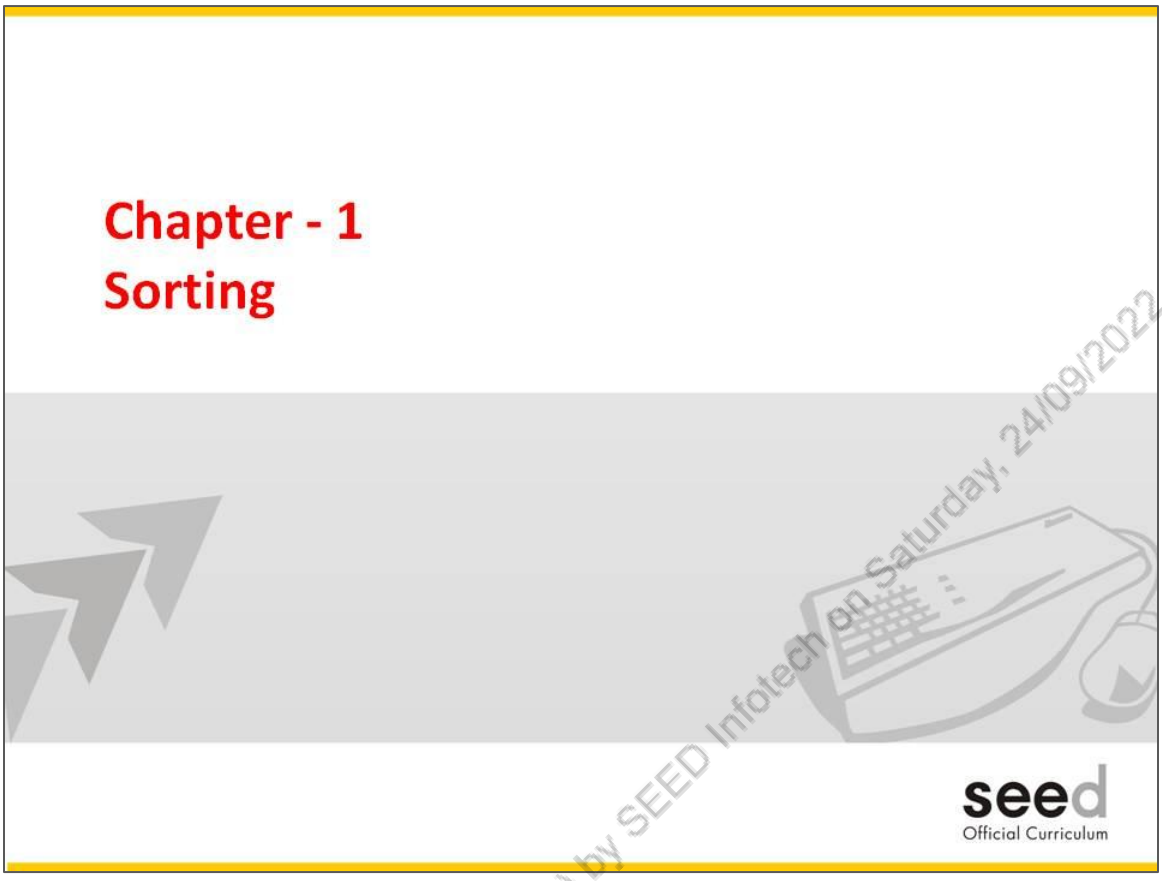
This icon suggests that it is good for the student to go through this additional reading material to bring more clarity to the concepts.

This icon indicates that this is a group discussion or group exercise. This is added for collaborative learning and problem solving. In the job environment one needs to take part in such discussions to arrive at the solution.

This icon indicates that more details are provided with respect to application of the technology/tool/concept which you are learning. This is application of technology learned to solve the real world problem.

This icon indicates a important note or tip from the application design perspective.

This icon indicates quiz to be solved in the classroom. This is for reinforcement of what you have learnt by challenging you through the questions.

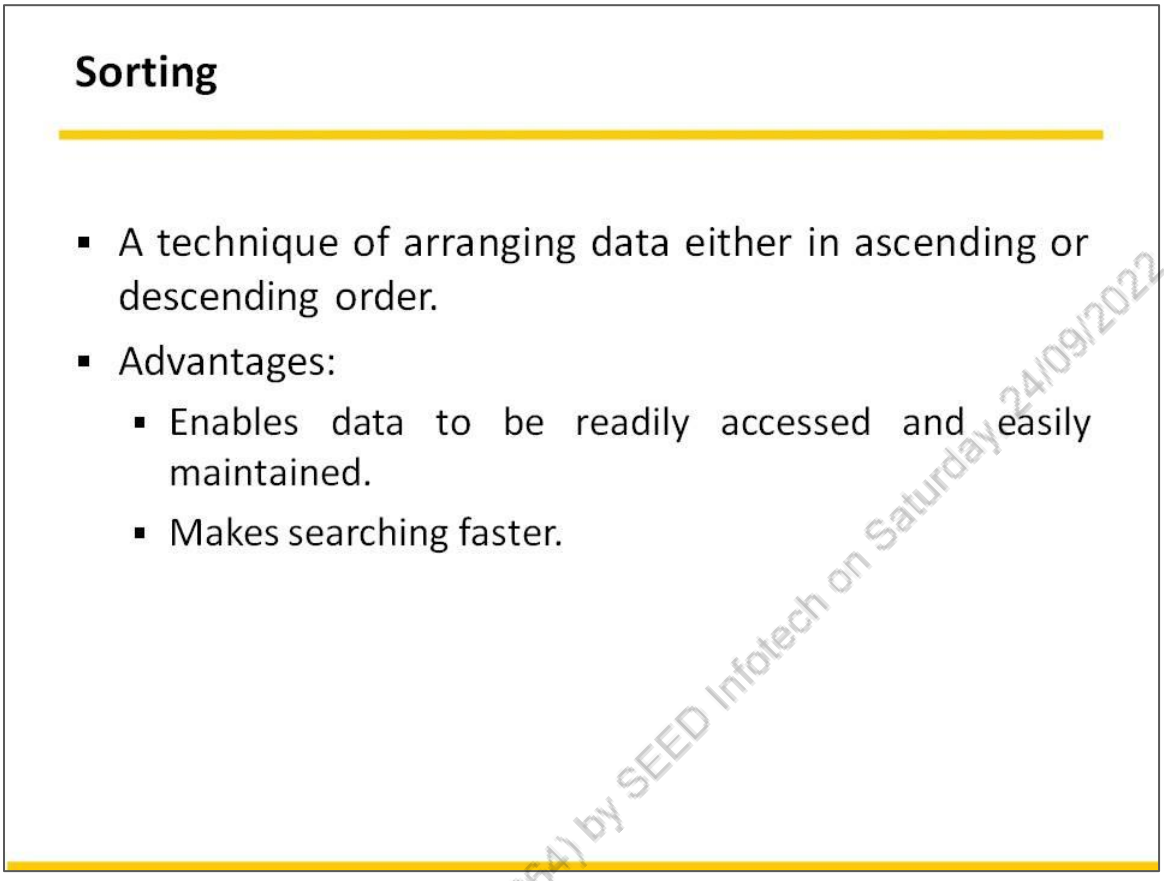


In this chapter, the need of sorting (arranging) data and its different techniques will be covered. Three of the methods of sorting, viz. bubble sort, insertion sort and selection sort will be discussed in detail along with their code.

Objectives

At the end of this chapter you will be able to:

- Define sorting
- List different sorting methods
- Construct programs for sorting using different sorting techniques like
  - Bubble sort
  - Selection sort
  - Insertion sort

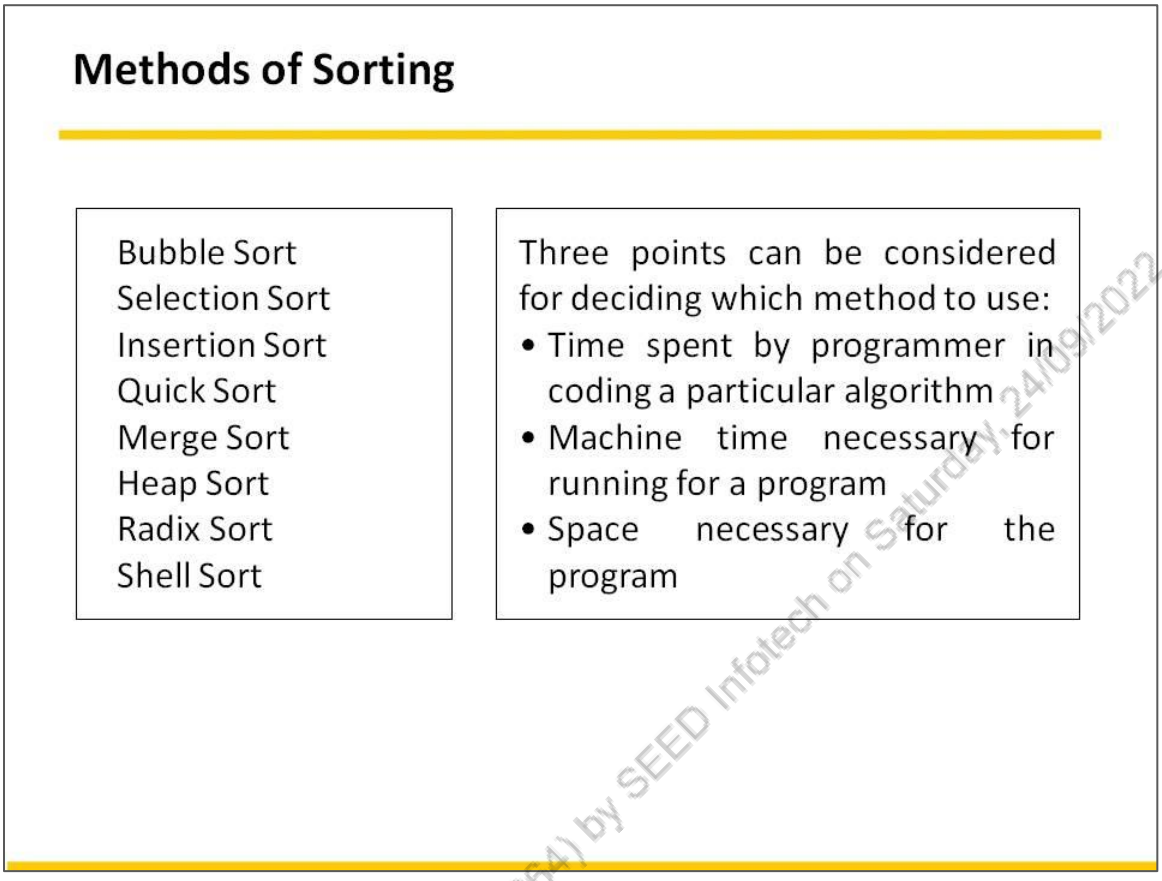


Many times it is required to search for some information from the stored data. If the data is arranged in a particular order, it takes less time and effort to find the desired information.

Data can be either alphabetical or numerical. For example, if names of students in a class are to be arranged, they will be arranged alphabetically. If marks of students are to be arranged then they will be arranged numerically.

Sorting is the technique used to arrange data in either ascending or descending order. Its advantages are –

- It enables data to be easily maintained and readily accessible.
- It also makes searching faster



There are a number of sorting techniques that can be applied for arranging data. Each method has different logic. The following points can be considered for deciding which sorting technique to use –

- Difficulty level of algorithm of the sorting technique
- Machine time required to run the program
- Space required to store the program

Optimization should be done in time or space, probably at the cost of the other.

Note: Finding time and space complexity is beyond the scope of this course.

Sorting is a very helpful technique and there are various methods devised to serve the purpose. Following is the list of some sorting techniques –

- Bubble sort
- Selection sort
- Insertion sort
- Quick sort

- Merge sort
- Heap sort
- Radix sort
- Shell sort

The first three methods have been discussed in detail in the following paragraphs.

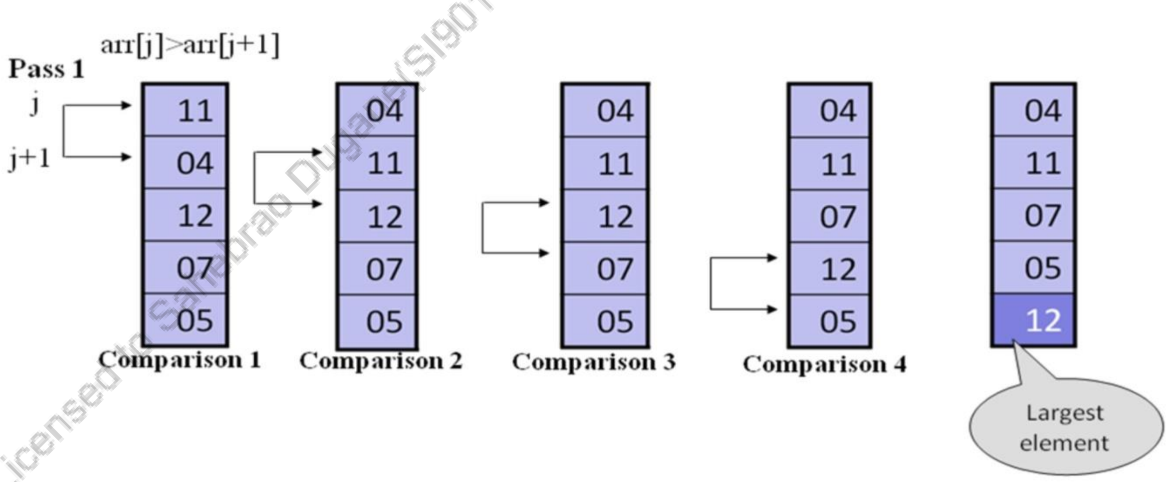
Bubble Sort

This is the simplest sorting algorithm. It works like this – if there are numbers to be sorted in ascending order, the largest number is pushed at the end of the list of numbers. The process is continued till all numbers are placed in their proper order.

Each number is compared with the succeeding number and their places are exchanged if the first number is larger than the succeeding number.

Scanning through the entire list once is called a pass. At the end of the first pass, though the entire list is not sorted, the largest number is placed at the end of the list. The passes are continued until the entire list is sorted.

The technique of bubble sort has been explained here with an example and reference diagrams. If the following list is to be sorted `11 04 12 07 05`, the bubble sort method can be applied as follows –



`11` and `04` are compared. Since `11` is larger than `4`, they are exchanged. The list now becomes

`04 11 12 07 05`

`11` and `12` are compared next. There is no exchange required. The list remains as it is.

```
04  11  12  07  05
```

12 and 07 are compared next. 12 is larger than 07, so they are exchanged. The list is now

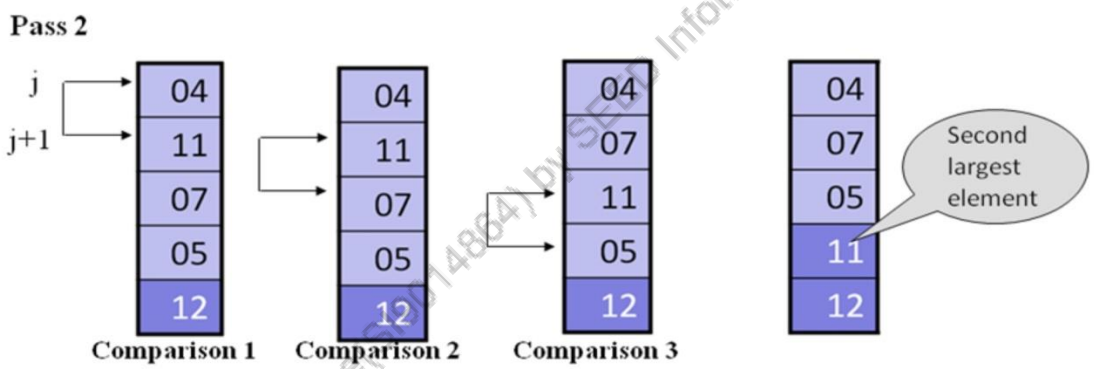
```
04  11  07  12  05
```

12 and 05 are compared next. 12 is larger than 5, so they are exchanged and the list becomes

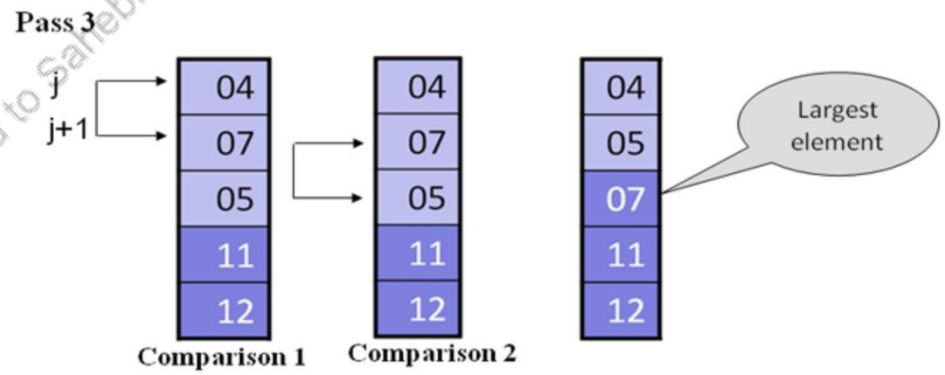
```
04  11  07  05  12
```

This is the end of Pass 1

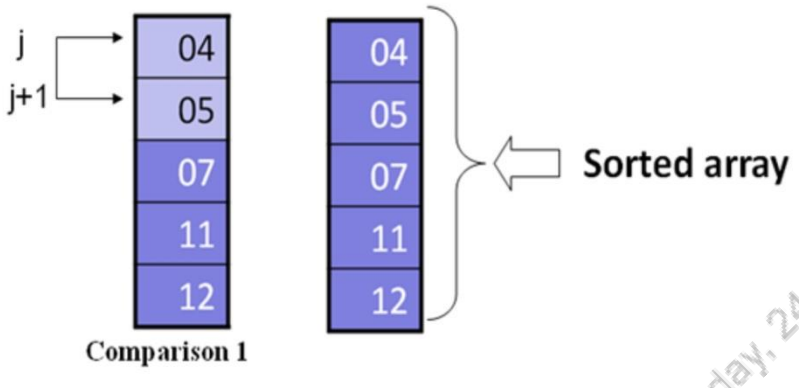
At the end of pass 1, notice that the largest number in the list is placed at the end. The process continues; at the end of second pass, the second largest number will be placed at the second last place.



There are three comparisons in 2nd pass. The second largest element occupies its respective position.



In 3rd pass, there are two comparisons.



In 4th pass, there is only one comparison.

Thus it can be seen that, if there are 5 (n) numbers to be sorted, 4 (n-1) passes are required and there are (n-1) comparisons in each pass.

A point to be noted is that after each pass one number is set at its proper place. After 1st pass, last element is not considered in comparison, after 2nd pass last two numbers are not considered for comparison since both will be in their proper place and so on.

Original list	Pass 1	Pass 2	Pass 3	Pass 4
11	04	04	04	04
04	11	07	05	05
12	07	05	07	07
07	05	11	11	11
05	12	12	12	12

Code for Bubble Sort

```
#include<stdio.h>
int main()
{ int arr[5]={11, 4, 12, 7, 5};
  int i, j, temp;
  int n = 5;    //no of elements
  printf("Using Bubble sort\n\nBefore Sorting\n\n");
  for(i=0; i<n ;i++)
    printf("%d\t", arr[i]);
  //sorting
  for(i=0;i<n-1;i++)
  {
    for(j=0;j<n-1-i;j++)
    {
      if(arr[j] > arr[j+1])
      {
        temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
  printf("\n\nAfter Sorting\n\n");
  for(i=0; i<n ;i++)
    printf("%d\t",arr[i]);
  return 0;
}
```

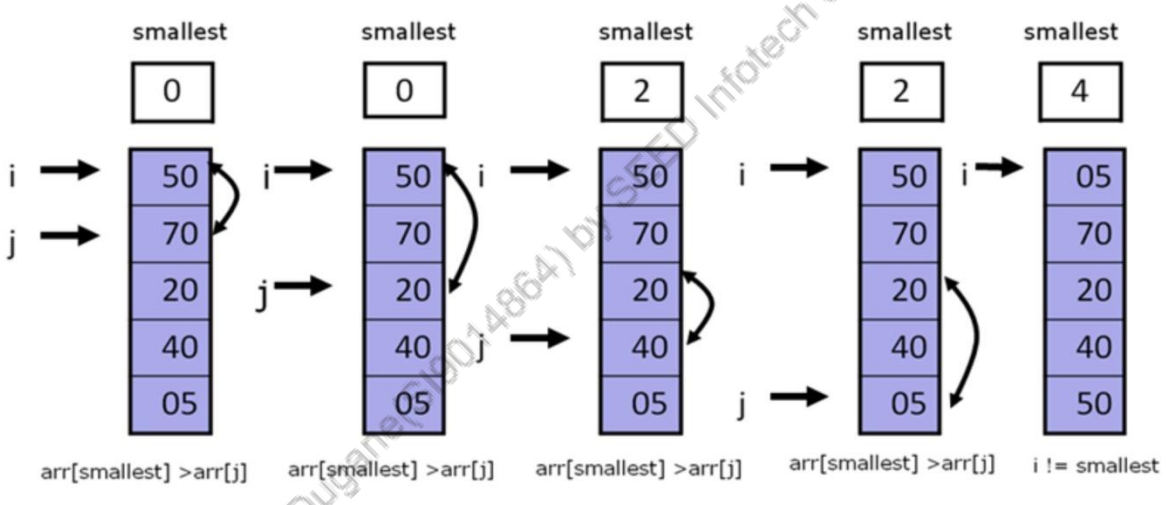

Selection Sort

This sorting method is faster than bubble sort, since data movement is minimized. Its logic is as follows - if a list of `n` data items is to be arranged in ascending order, the smallest number in the list is selected first and placed at the beginning. This is done by interchanging places of first element in unsorted list with the smallest element.

In the next pass the smallest element in the remaining elements is searched and placed in the second place. This process is continued till all the elements are in proper order. In each pass, the number of comparisons for searching smallest element will be reduced by 1, since one element is sorted and placed in position after each pass. Consider the following list of numbers to be sorted – **50 70 20 40 05**. There are five elements to be sorted i.e., `n = 5`.

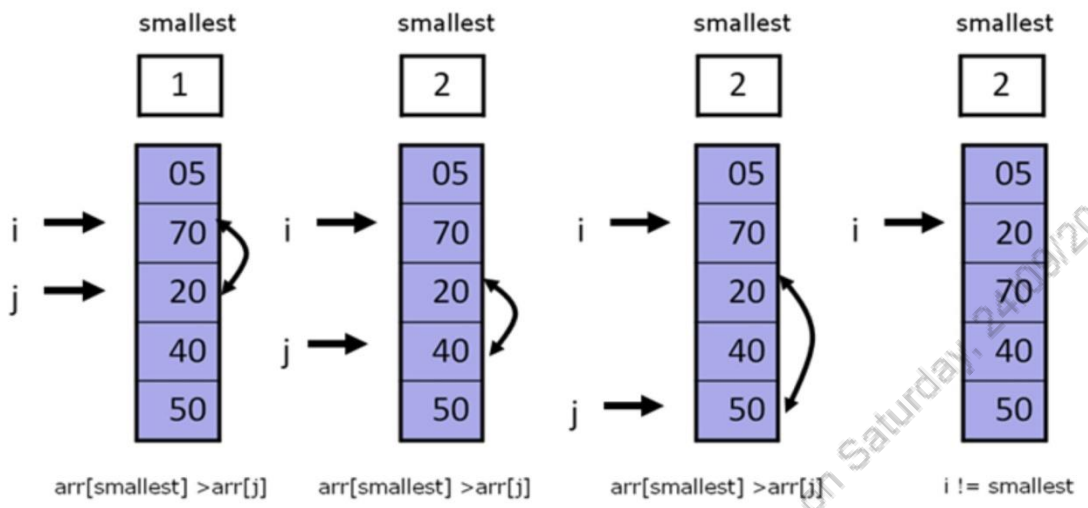
Pass 1

First element will be compared with all other elements. If an element smaller than the first element is found, its index is stored in smallest. When there are no more elements to compare and the first element is larger than the element at the smallest index, they are swapped.



There is one element in its sorted place. Now consider the remaining four elements. 20 is the smallest element in the remaining list. It should be placed in the second last place.

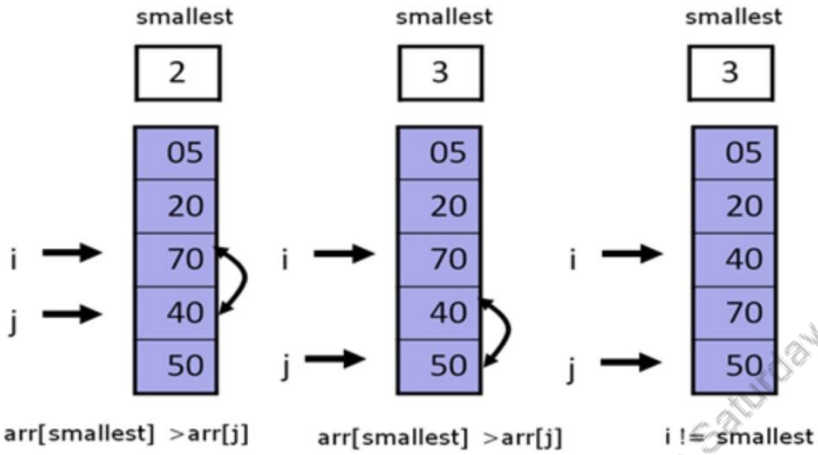
Pass 2



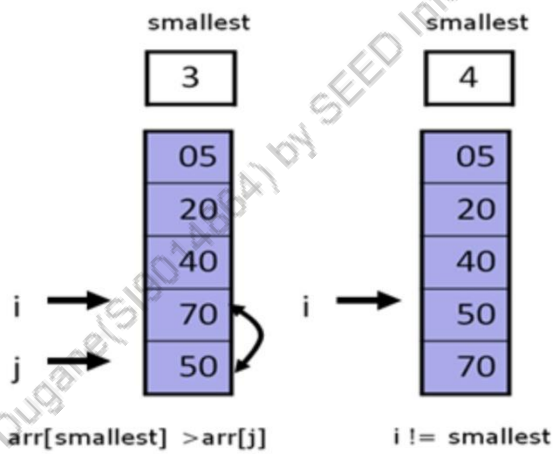
The comparison in pass 2 begins with the element in second position. All the remaining elements are compared with the element in the second position. If an element smaller than the second element is found, its index is stored in smallest. The value of smallest will now be 2. When there are no more elements to compare and the second element is larger than the element at the smallest index, they are swapped.

The same procedure is followed until all elements are sorted. If there are 5 elements to sort, there will be four comparisons and the number of comparisons will reduce by one after each pass.

Pass 3



Pass 4



Thus there are two steps involved in each pass -

- Search for the smallest element
- Place it in the proper place.

Code for Selection Sort

```
#include<stdio.h>
int main()
{
  int arr[5]= {50, 70, 20, 40, 5};
  int i, j, temp, smallest;
  int n = 5;                    //no of elements
  printf("Using Selection sort\n\n");
  printf("Before Sorting\n\n");
  for(i=0; i<n ;i++)
    printf("%d\t",arr[i]);
  for(i=0; i < n-1 ; i++)
  {
    smallest = i;
    for(j=i+1; j<n ; j++)
    {
      if(arr[smallest] > arr[j])
        smallest = j;
    }
    if(i != smallest)
    {
      temp = arr[i];
      arr[i] = arr[smallest];
      arr[smallest] = temp;
    }
  }
printf("\n\nAfter Sorting\n\n");
  for(i=0; i<n ;i++)
    printf("%d\t",arr[i]);
  return 0;
}
```

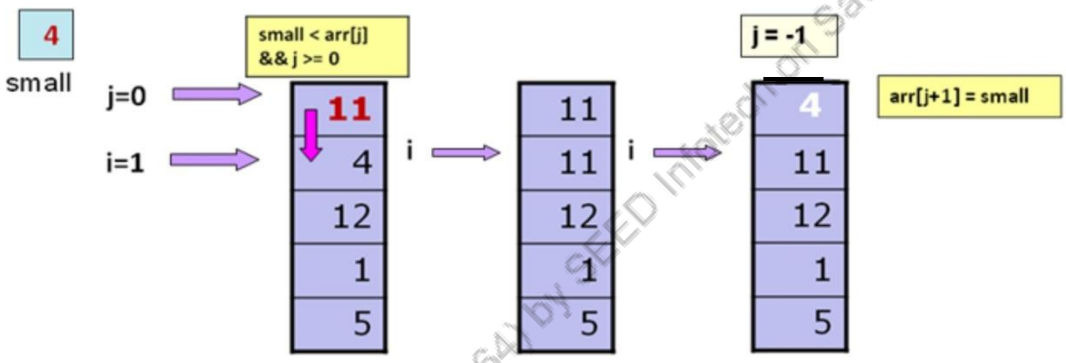

Insertion sort

Insertion sort can be of three types - simple insertion sort, list insertion sort, insertion sort using binary search. In this chapter, simple insertion sort has been discussed. Simple insertion sort is more efficient than bubble sort.

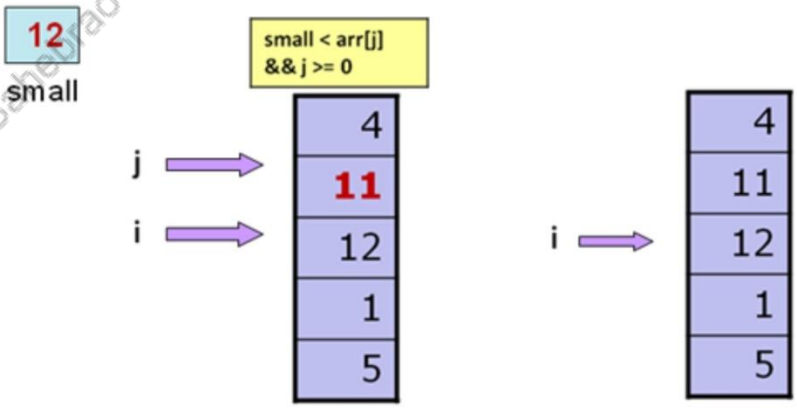
Each element is picked up and inserted in its proper place. If some element in the list is picked for inserting, all elements before it will be in their proper order.

Consider the list to be sorted using this technique – **11 4 12 1 5**

First, the 2nd element is picked, i.e. 4 and placed in the temporary variable named `small`. `4` is then compared with each element preceding it and as long as `small` is less than any element, the elements are shifted downwards till a place is found for `small`. 11 is greater than 4, so it is shifted to the next position. `Small` is inserted at the first position. Beginning of the list is reached.

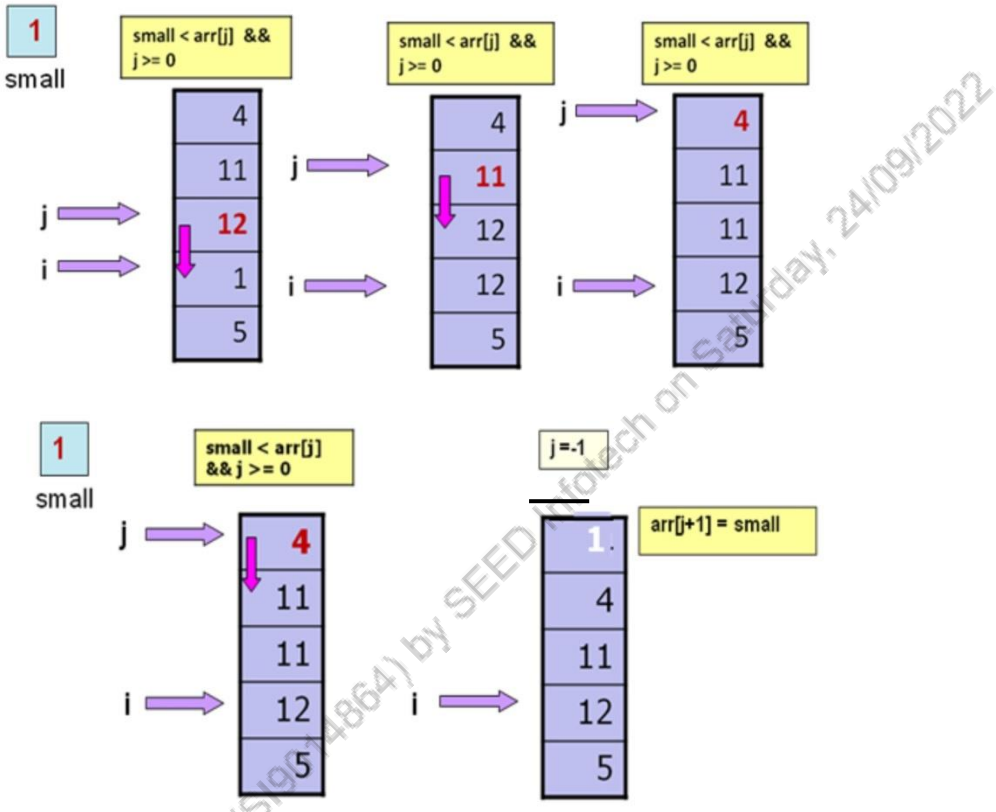


In the second pass, 3rd element is picked first, i.e in this case 12 and placed in the temporary variable named `small`. `small` is first compared with element preceding it, i.e., 11. 11 is smaller than `small`. No further elements need to be compared.

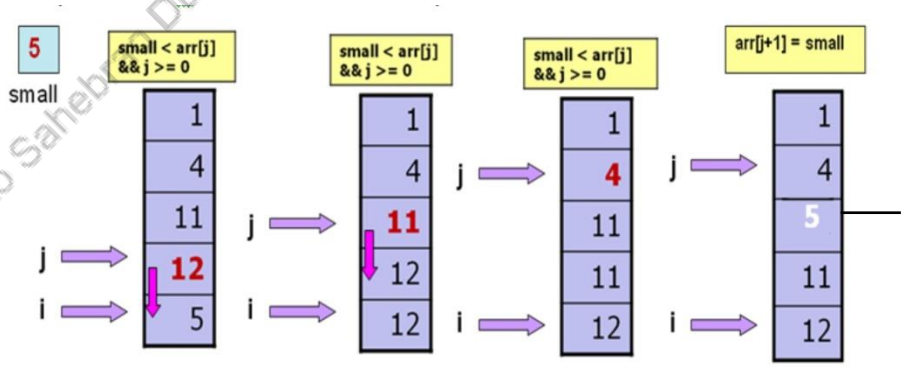


In the third pass, fourth element is picked first, i.e. 1 and placed in the temporary variable named `small`. First, `small` is compared with the element preceding it, i.e 12. Since 12 > 1, it is shifted to the next position. Again, `small` is compared

with 11. 11>1, so it is also shifted to the next position. `small` is compared with 4. 4>1, so it is also shifted to the next position. Beginning of the list is reached and so no more comparisons are possible. `small` is inserted at the correct position.



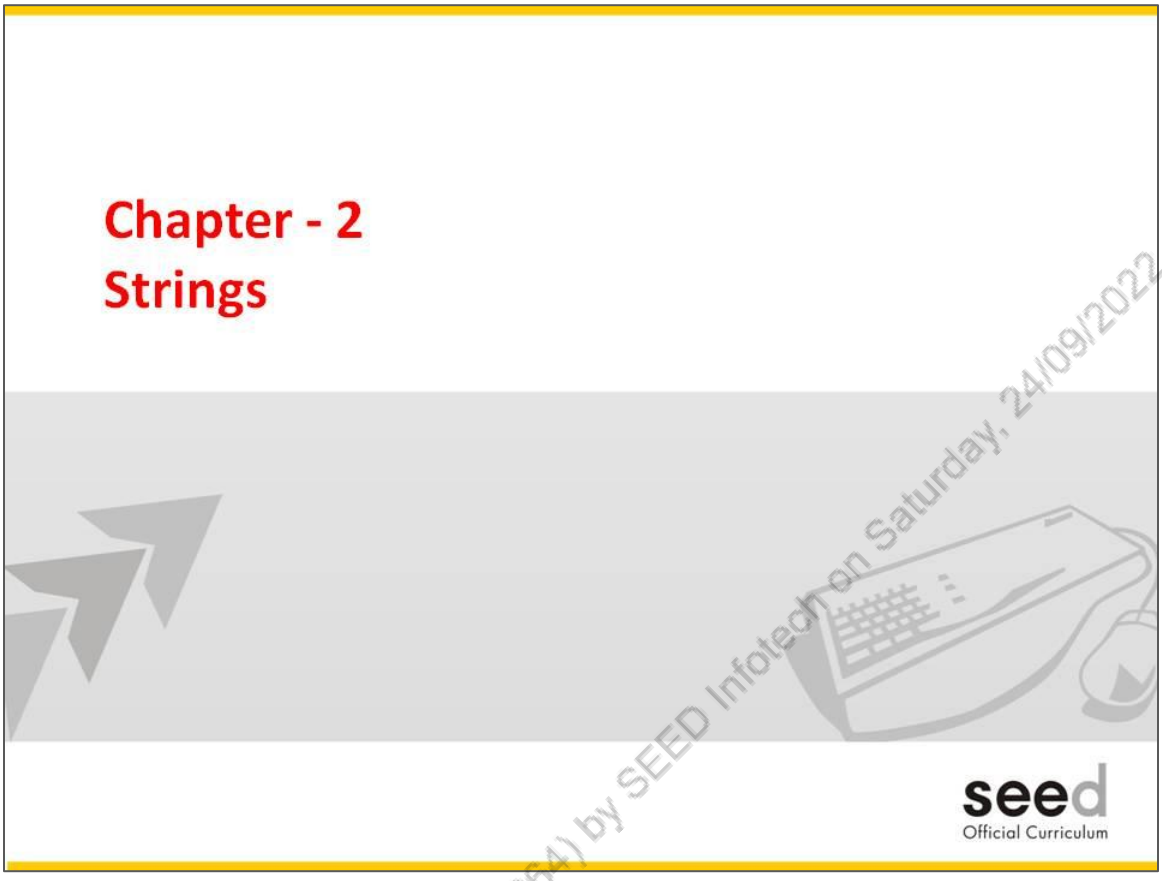
Similarly in the last pass too, 5 is inserted at the correct position.



Code for Insertion Sort

```
#include <stdio.h>
int main()
{
  int arr[5] = {11, 4, 12, 1, 5};
  int i, j, small;
  int n = 5;         //no of elements

  printf("\nBefore Sorting\n\n");
  for(i=0; i<n ;i++)
    printf("%d\t",arr[i]);
  for(i = 1;i < n ; i++)
  {
    small = arr[i];
    for(j = i-1; j>=0 && small < arr[j]; j--)
      arr[j+1] = arr[j];

    arr[j+1] = small;
  }
  printf("\n\nAfter Sorting\n\n");
  for(i=0; i<n ;i++)
    printf("%d\t", arr[i]);
  return 0;
}
```

You should know when to use which sorting technique.



This chapter covers strings and the ways to access them. It also covers some built-in string manipulation functions like `strcpy`, `strcat`, etc. The concept of pointers is covered in this context.

Objectives

At the end of this chapter you will be able to:

- Declare and initialize a string.
- Perform operations on a string using [ ] operator and a pointer
- Use built-in string manipulation functions.
- Construct user-defined functions equivalent to built-in string functions.

What is a String?

- One dimensional array of characters is called a string.

```
char   stringName[MAXSIZE];
```

data type name of the array upper limit

- Every string terminates with a '\0' character (ASCII value is 0).

```
char custName[15] = "NIKHIL";
```

N	I	K	H	I	L	\0

`int`, `float` and `char` respectively represent integer, floating point and character constants. "RAHUL" is a string constant. String is not a basic data type in C language. It is implemented as a one dimensional character array. A set of characters is called a string.

```
char custName[15] = "NIKHIL";
```

String is enclosed in double quotes. Unlike integer array, strings terminate with `'\0'` character whose ASCII value is zero. `'\0'` or also called `NULL` character is automatically added by the compiler. In the example mentioned, `custName` is the name of the variable that represents the string "NIKHIL". The size of the array is 15. This indicates that the maximum number of characters that can be stored in the array is 15 (including `'\0'`).

Each character occupies 1 byte in memory. 'X' is a character whereas "X" is a string consisting of one character. Note the difference between the two.

Strings can be initialized at the time of declaration. It is not necessary to mention the size of the array at the time of initializing. But the size should be mentioned if the string is accepted at runtime.

```
char string[] = "SEED";
char custName[15] = "Nikhil";//Array size larger than
the string size is allowed
```

Consider the following declaration.

```
char city[10] =  {'N', 'E', 'W', ' ', 'D', 'E', 'L',
'H', 'I', '\0'};
```

'\0' character is explicitly added so that `city` is considered as a string. If it is not added, it is just a sequence of characters and would give erroneous results with built-in string manipulation functions.

Initialization cannot be separated from the declaration.

```
char str[5];
str = "SEED";                    // not allowed
```

Accepting Strings

- Using `scanf()`
  - `%c` specifier – `'\0'` character has to be explicitly appended.

```
for(i=0; i<5; i++)
  scanf("%c", &custName[i]);
custName[i] = '\0';
```

  - `%s` specifier – stops accepting when a white space character is entered.

```
scanf("%s", custName);
scanf("%5s", custName);
```

- Using `gets()` – accepts string with spaces and there is no need to append `'\0'`.

```
gets(custName);
```


In case of integer array, individual data items are manipulated using the subscript operator. But in case of strings, set of characters is treated as a single entity and accessed. The name of the array represents the entire string. It actually points to the base address of the array.

As mentioned earlier, strings can be accepted from the user at runtime.

1. Character by character using `%c` specifier.

```
for(i=0;i<10;i++)
    scanf("%c", &custName[i]);
custName[i] = '\0';
```

   Note that `NULL` character is added explicitly. The definite number of characters in a loop has to be specified. This is not a convenient way to accept strings.

2. As a set of characters using `%s` specifier.

```
char custName[20];
printf("Enter your name");
```

```
scanf("%s", custName); //does not accept white
spaces
printf("%s", custName);
```

Unlike previous `scanf()` calls, in the case of character arrays, the ampersand (&) is not required before the variable name. The argument is the name of the character array which itself is the base address. If the name entered is "Nikhil Shah", only Nikhil will be scanned and accepted and Shah will be ignored or discarded.

3. As a set of characters using `gets()` library function.

   The limitation of `scanf()` function is that it accepts a string without white spaces. Instead, `gets()` function should be used since it accepts a string with white space characters and terminates with newline character (when the user presses enter key on the keyboard).

```
gets(custName);
```

   In this case, the entire string "Nikhil Shah" would be scanned. However, mixing of `gets()` and `scanf()` should be avoided.

4. Using `scanf` with edit set conversion code `%[..]`

   C supports a format specification known as the edit set conversion code `%[..]` that can be used to read a line containing a variety of characters, including a white space. To do so, '`s`' type conversion character within the control string (%s) is replaced by a sequence of characters enclosed in square brackets, designated as […]. Whitespace characters may be included within the brackets, thus accommodating strings that contain such characters.

   When the program is executed, successive characters will be read from the standard input device as long as each input character matches one of the characters enclosed within the brackets. The order of the characters within the square brackets need not correspond to the order of the characters being entered. Input characters may be repeated.

The string terminates when an input character that does not match any of the characters within the brackets is encountered. A null character (`'\0'`) is then automatically added to the end of the string.

A variation of this feature which is often more useful is to precede the characters within the square brackets by a circumflex (i.e. ^). This causes the subsequent characters within the brackets to be interpreted in the opposite manner. In the example mentioned below, the string entered from the standard input device can contain any ASCII characters except the newline character. This statement would accept a string with whitespace characters.

```
scanf("%[^\n]", name);
```

Restricting the number of characters

Field width can also be specified using the form `%ws` in the `scanf` statement for reading a specified number of characters from the input string as shown in the following syntax.

```
scanf("%ws", stringVariable);
```

Example

```
scanf("%5s", custName);
```

There are two cases to be considered:

5. The width `w` is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.
6. The width `w` is less than the number of characters in the string. The excess characters will be truncated and left unread.

Accept a string using `gets()` function instead of `scanf()`. This will help to scan a string with white spaces.

Displaying Strings

- Using `printf()`
  - `%c` specifier

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

  - `%s` specifier

```
printf("%s", custName);
printf("%10s", custName);
```

- Using `puts()`

```
puts(custName);
```


A string can be displayed using formatted I/O function – `printf()` as well as `puts()`.

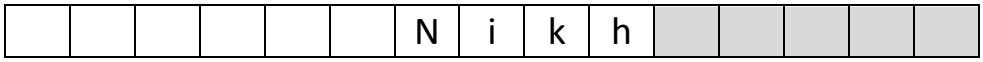
1. Using `printf()` with conversion specifier - %c
2. The above mentioned example with %c as specifier would print the first 10 characters of the string, character by character.
3. Using `printf()` with conversion specifier - %s
4. The format %s can be used to display an array of characters that is terminated by a null character. For example,

```
printf("Hello!!! ");
printf("%s", custName);        // prints Nikhil Shah
```

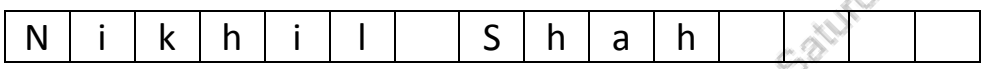
The above example prints "Hello!!! Nikhil Shah". `printf()` function requires explicit newline character to print on a new line.

The precision with which the array is displayed can also be specified.

a. The specification `%10s` shown on the slide indicates that the string would be displayed up to ten characters. But will show the entire string if it is longer than 10 characters.
b. For instance, the specification `%10.4s` indicates that the first four characters are to be printed in a field width of 10 columns, right justified.



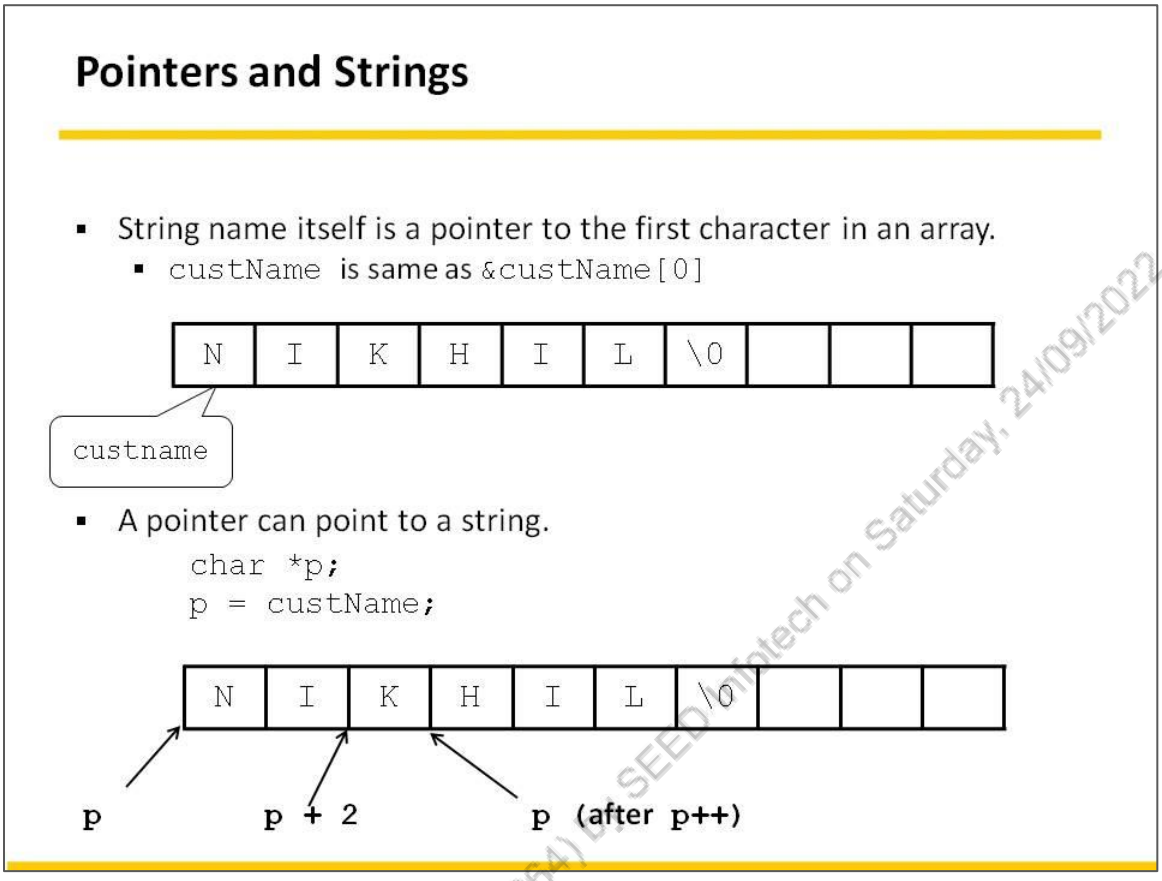
c. However, if we include the minus sign in the specification (%-15s), the entire string will be printed left justified in a width of 15. Remaining spaces will be left blank.



d. When the field width is less than the length of the string, the entire string is printed.

5. Using `puts()` function

This function prints the string and takes the cursor to the next line automatically.



String name represents the base address of the array. This is same as a pointer to the first character in an array.

If there is an array `char custName[10];` 'custName' gives base address of this array. If `char *p;` is declared, then the statement `p = custName;` causes `p` to point to the beginning of the array `custName`.

- Contents of array can be obtained as `*(p + 0)`, `*(p + 1)`, which gives the same result as `custName[0]`, `custName[1]`…..
- Address of array elements can be obtained as `p`, `(p + 1)` or `&custName[0]`, `&custName[1]`,…..

Or instead of using an array of characters of given size and setting a pointer to this array, pointer to a character can be used and sufficient memory can be allocated to it.

A pointer can point to a string. Consider the following code snippet.

```
char *p;                    // declares a pointer variable
char custName[20] = "Nikhil";
p = custName;               //assigns the base address to
the pointer
while(*p != '\0')
{
  printf("%c", *p)
  p++;                      //incrementing pointer
}
```

p jumps to the next memory location after p++ till '\0' character is encountered.

String Functions

- There are many in-built string-handling functions declared in string.h file.
- Common ones are

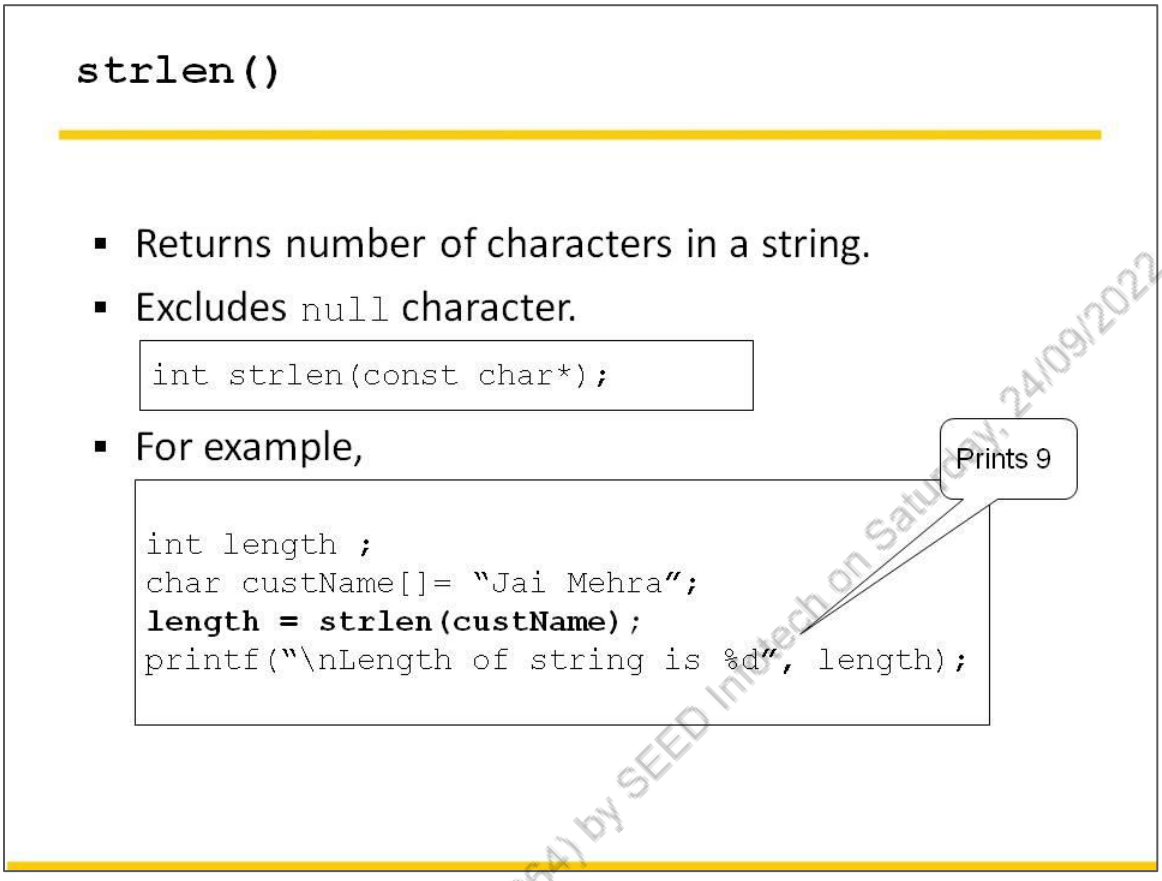
Function	Purpose
`strlen()`	Returns the length of a string
`strcpy()`	Copies one string into another
`strcat()`	Concatenates two strings
`strcmp()`	Compares two strings
`strrev()`	Reverses the string

The string.h file contains many built-in functions to handle strings. Some of them are given in a tabular form below.

Function	**Purpose**
`strlen`	Finds length of a string. NULL character not counted.
`strlwr`	Converts a string to lower case.
`strupr`	Converts a string to uppercase.
`strcat`	Appends one string at the end of another.
`strncat`	Appends first n characters of a string at the end of another.
`strcpy`	Copies a string into another.
`strncpy`	Copies first n characters of one string into another.
`strcmp`	Compares two strings.
`strncmp`	Compares first n characters of two strings.

Note: This chapter covers only three functions: strlen(), , strcpy(), and strcat() in detail.

Read documentation of functions in string.h and practice using them.



`strlen()` function is used to get the length of a string i.e. the number of characters in a string, excluding the terminating character `'\0'`. The argument to the function is a constant string.

```
int strlen(const char *s);
```

Implementing `strlen()` function

`strlen()` function can be implemented in two ways – using array and using pointers

1. Using array

```
int aStrLen(char s[])
{
  int i=0,len=0;
  while(s[i] !=
  {
    i++;
    len++;
```

Loop terminates when the null character is found

```
   }
 return len;
}
```

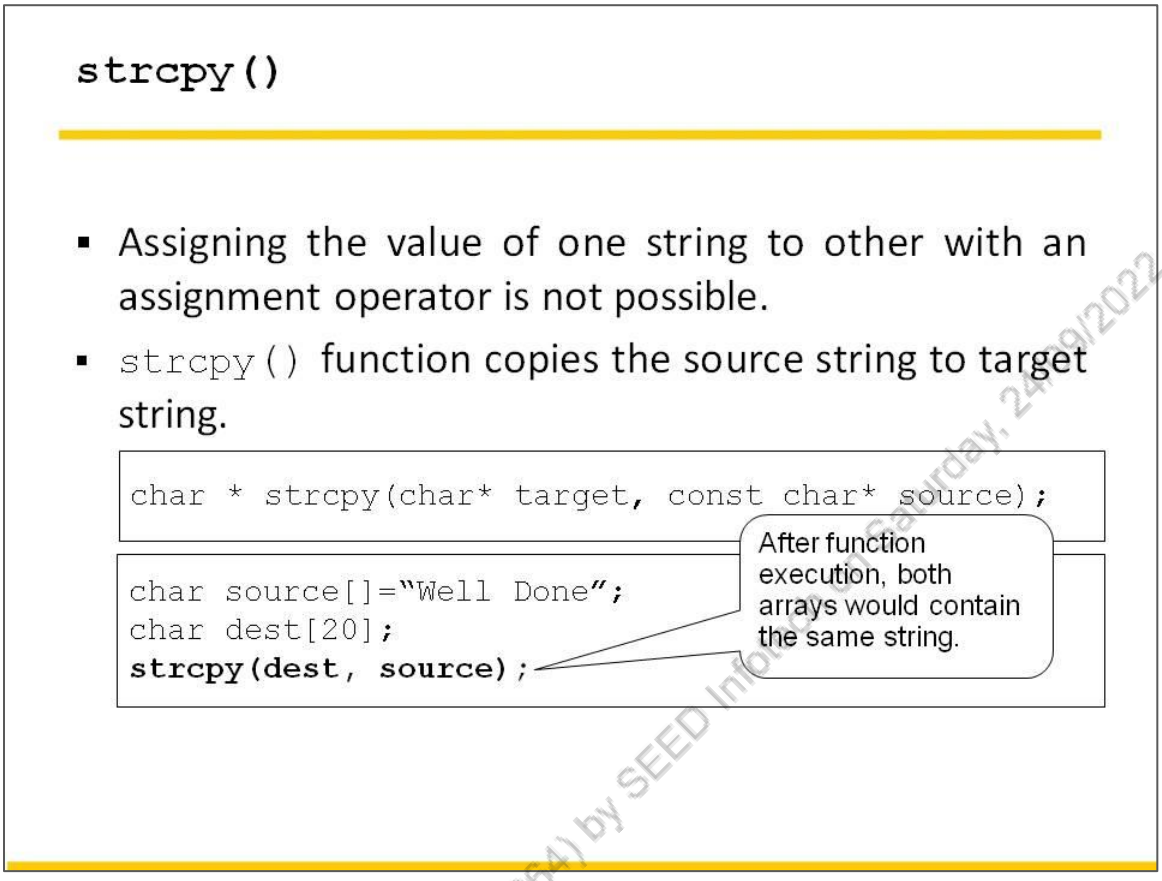
2. Using pointer notation

```
int pStrLen(char *s)
{
  int len =0;
  while((*s) !=
  {
     len++;
     s++;
  }
  return len;
}
```

Pointer incremented till null character is found

Using the function

```
int main()
{
  int len;
  char str[20]= "SEED";      //initialization along with
declaration
  len = aStrLen(str);        // len is assigned 4
}
```



One string cannot be assigned to another using assignment operator. C treats strings as arrays. String name represents the base address of the array. It is a constant pointer to the first character of the array.

```
string2 = "SEED";    // invalid
string2 = string1;   // invalid
```

strcpy() function

strcpy() function copies the contents of source string to destination string. If the destination string already contains some value, it is overwritten. The function takes the base address of destination and source strings as the first and second parameter respectively. The size of the destination string should be larger or equal to the source string. The function returns the address of the destination string.

```
char * strcpy(char *destination, const char *source);
```

The return value is required when the function is nested within another function. The inner function returns the address of the string to the outer function as shown in the example mentioned below.

```
strcpy(string2,strcpy(destination,source));
```

Note: Here the function `strcpy()` returns the address of the first character of the string so that it can be used in cascading. For example, `(strcpy(c1,strcpy(c2,c3));`

Implementing `strcpy()` function

`strcpy()` function can be implemented in two ways – using array and using pointers

3. Using subscript notation

```
char* aStrCpy(char dest[] ,const char source[])
{
  int j=0;
  while((dest[j] = source[j]) != '\0')
    j++;
  return dest;
}
```

Copying character by character

4. Using pointer notation

```
char* pStrCpy(char *dest ,const char *src)
{
   int j=0;
   while((*dest = *src)!= '\0')
   {
         dest++;
         src++;
   }
    return dest;
}
```

5. Using function

```
int main()
{
   char source[20]= “Seed”;
   char dest[20]= “Welcome to ”;
   aStrCpy(dest, source);
   puts(dest);  //displays Welcome to Seed
}
```

strcat()

- Concatenates the source string at the end of target string.

```
char* strcat(char *target, const char *source);
```

- For example,

```
char source[] = "Seed";
char dest[] = "Welcome to ";
strcat(dest, source);
```

`strcat()` function concatenates the source string at the end of the destination string. This function takes two parameters – destination and source string. It also returns the address of the destination string.

```
char* strcat(char *destination, const char* source);
```

The source string is appended to the destination string. If the destination string does not contain any value, source string is added to destination string. In the example shown on the slide, the destination string would contain the string `"Welcome to Seed"`.

Implementation of `strcat()` function

`strcat()` function can be implemented in two ways – using array and using pointers. The following code snippet demonstrates the use of array.

```
char* aStrCat(char dest[], char src[])
{
   int j = 0, k = 0;
   while(dest[j] != '\0')
```

```
        dest[j++];                    // making j=length
of destination string
// copying to destination string
   while((dest[j] = src[k]) != '\0')
   {
     src[k++];
     dest[j++];
   }
}
//Using function
int main()
{
   char source[20]= "Seed";
   char dest[20]= "Welcome to ";
   aStrCat(dest, source);
   puts(dest);  // prints Welcome to Seed
}
```

strcmp()

- Compares two strings character by character and returns -1, 1, or 0 when it finds the first non-matching characters.

```
int strcmp(const char *string1,const char *string2);
```

- For example,

```
a = strcmp("Apple", "Zoo");
// a = -1 since string1 is smaller than string2
a = strcmp("Zoo", "Apple");
// a = 1 since string1 is greater than string2
a = strcmp("Apple", "Apple");
// a = 0 since string1 is equal to string2
```


`strcmp()` function compares two strings identified by the parameters. It returns an integer value based on the comparison.

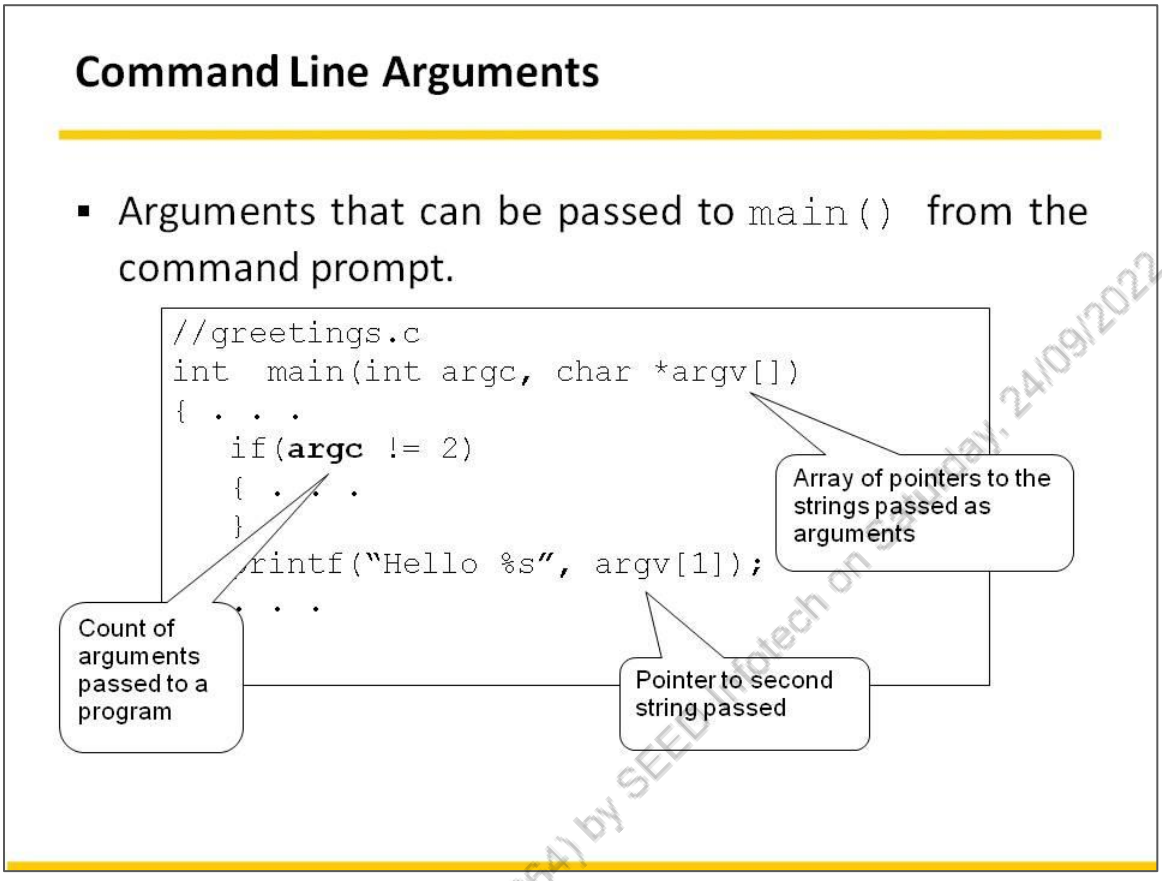
```
int strcmp(const char *string1, const char *string2);
```

It returns a negative value, 0 or positive value if `'string1'` is less than, equal to or greater than `'string2'` respectively. Strings `string1` and `string2` are compared character by character. The first character where both the strings do not match, the function returns a value. There are three cases to be considered depending on return value.

1. `string1` is less than `string2`: returns a negative value
2. `string1` is equal to `string2`: returns 0
3. `string1` greater than `string2`: returns a positive value

Write a function to find the number of vowels in a string.

Master string handling for all built-in string functions.



Strings can be used to execute C programs or applications from the command prompt. Such programs are also called console applications. Strings are passed as arguments to such applications using the command prompt. These arguments are called command line arguments.

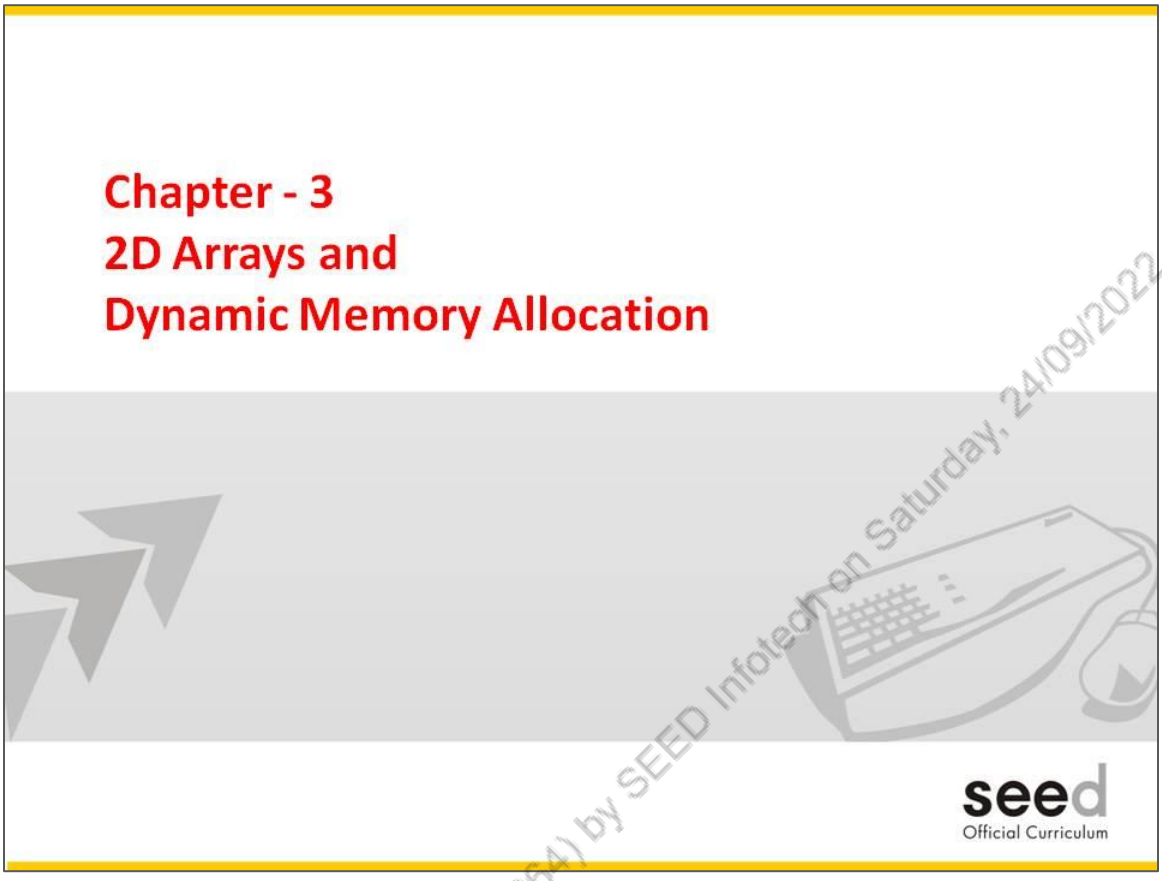
```
int main(int argc, char *argv[])
{
   . . .
}
```

- `argc` is an integer which contains the count of or number of parameters passed to a program.
- `argv` is an array of pointers to strings which contains all the parameters passed. The name of program occupies the place of `argv[0]` followed by each of the parameters.

The example mentioned on the slide can be executed by specifying the arguments to be passed. The method has been explained in Appendix B.

For example, if the argument passed is “Snehal”, the output will be `Hello Snehal`. The name of program will be passed by default in Visual Studio. Before printing the message, the count of arguments passed `(argc)` is checked. If it is not correct, the program will not execute.

For details of how to execute such a program please refer Appendix B.

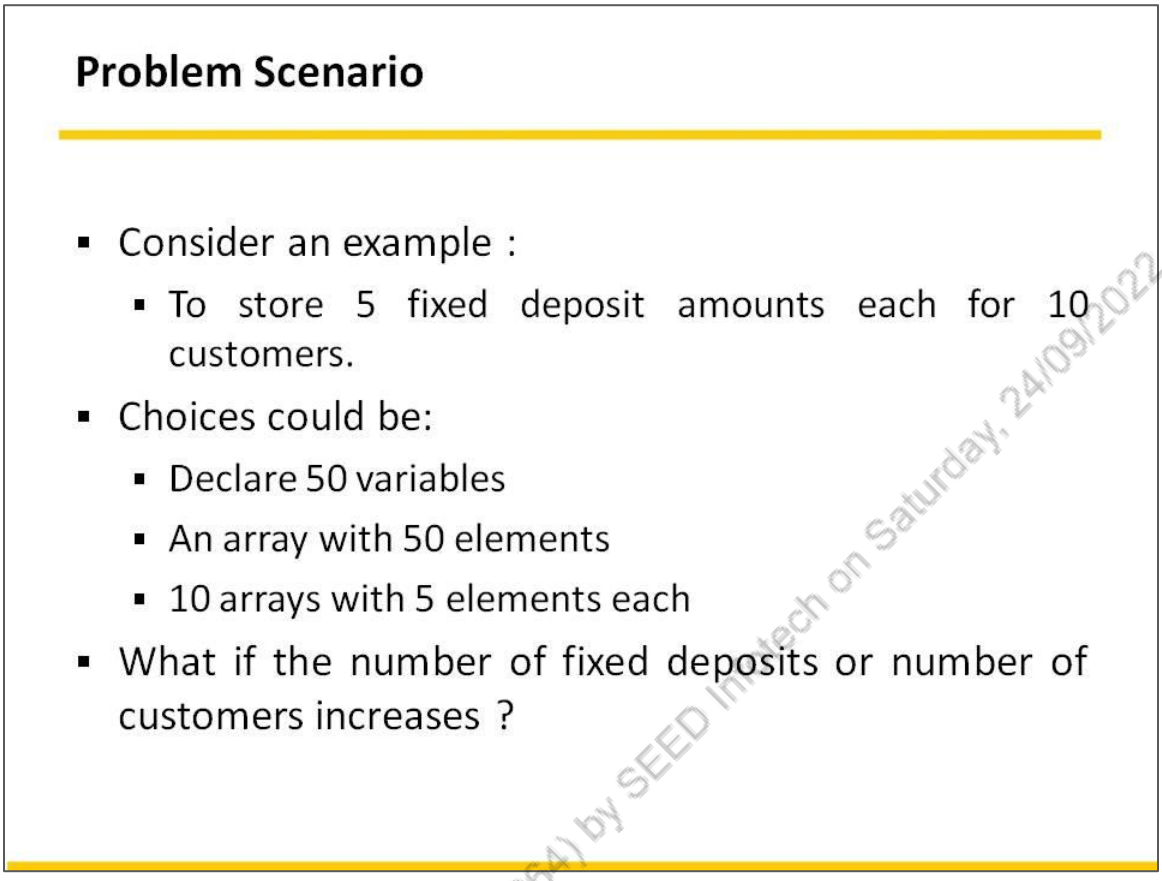


This chapter covers the need of two dimensional arrays, its declaration and usage. The chapter focuses on array of pointers, pointer to an array of elements and pointer to a pointer. It also covers dynamic memory allocation for a pointer variable to store two dimensional arrays.

Objectives

At the end of this chapter you will be able to:

- Identify the need of 2D Arrays.
- Declare, initialize and use 2D Arrays.
- Perform 2D array I/O operations.
- Pass 2D array to functions.
- Use Array of pointers, pointer to an array of integers, pointer to a pointer.
- List built-in functions for allocating memory dynamically.
- Allocate memory dynamically to store the 2D Array.



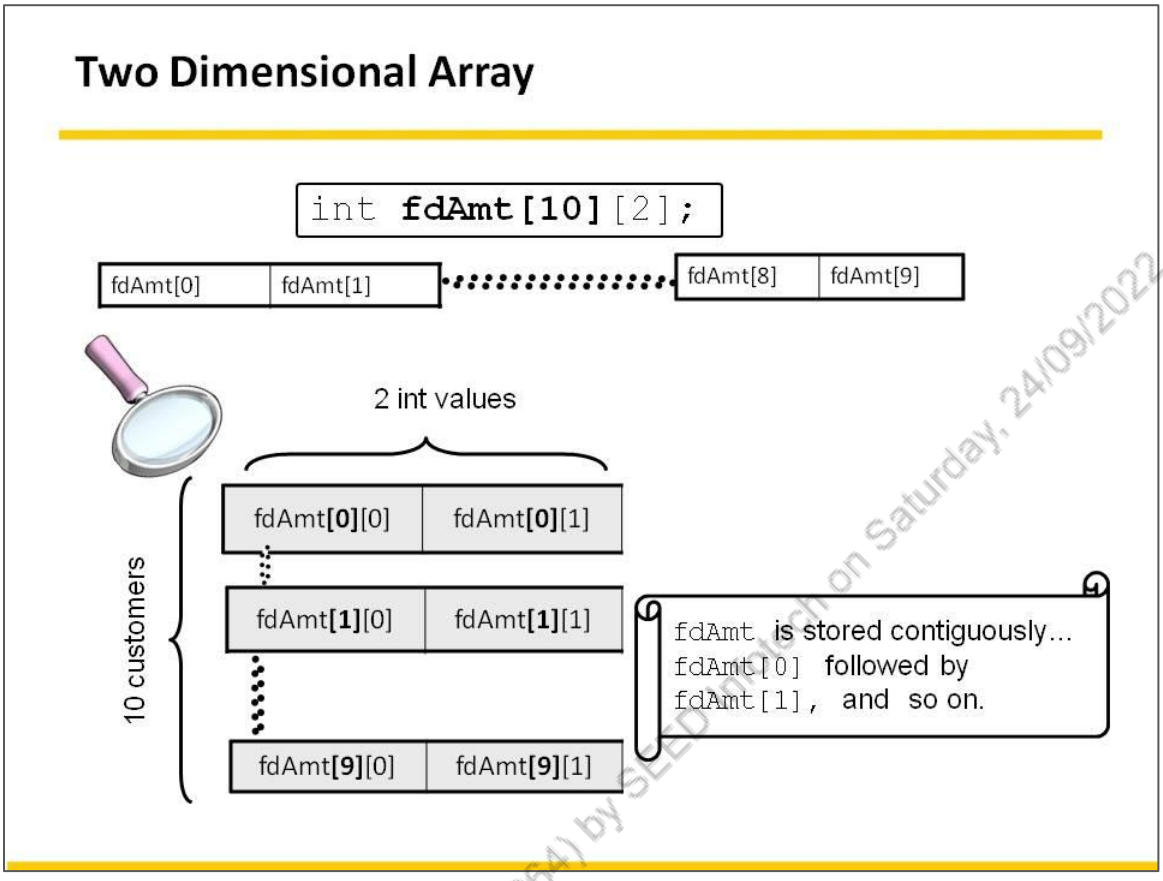
Consider an example before going ahead with the concept of two dimensional arrays. If you want to store 5 fixed deposit amounts each of 10 customers, there are different solutions.

- Declare 50 variables (10 customers * 5 fixed deposits).
- An array with 50 elements.
- 10 arrays with 5 elements each.

None of them is a feasible solution. As the number of customers increase, the variables would be difficult to manage in the code.

The solution is a two dimensional array.

	fdAmt1	fdAmt2	fdAmt3	fdAmt4	fdAmt5
Customer 1	1000	2000	-500	-1000	2000
⋮	1500	-200	1500	-1000	6000
Customer 5	5000	2000	-1500	-3000	1800
	10000	1000	4500	-5000	-4600
	3000	1600	-3050	-1840	-500



Considering the same example, 5 fixed deposit amounts of 10 customers each, can be represented using a two dimensional array. Two dimensional array is also called an array of arrays. It is represented as shown below:

```
[Storage-class] data-type arrayName[rowSize][colSize]
```

`rowSize` and `colSize` must be positive integers. Storage class is optional.

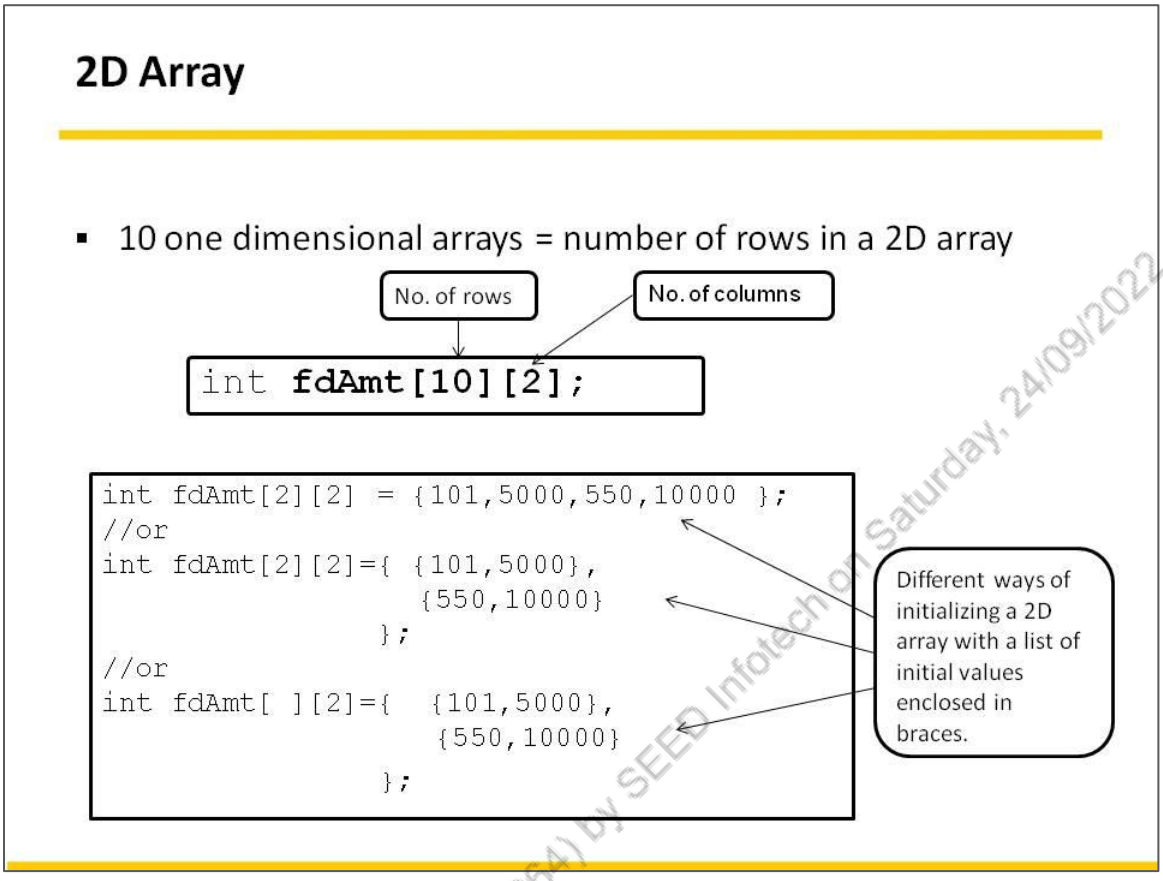
For example,

```
int fdAmt[10][5];
```

- `fdAmt` is an array of 10 elements.
- Each element of the array `fdAmt` is an array of 5 integers.
- `fdAmt[0]` is the first element of `fdAmt` which in turn is an array of 5 integer values.
- Similarly, `fdAmt[1]`, `fdAmt[2]` ….. so on, are the elements of an array.

- `fdAmt[0]` is itself an array, its first element is `fdAmt[0][0]`, second element is `fdAmt[0][1]`, and so on.
- The fixed deposit amount of the third customer can be accessed by writing, `fdAmt[2][0]`, `fdAmt[2][1]`, and so on.
- `fdAmt[0], fdAmt[1], fdAmt[2] ------ fdAmt[9]` are adjacent to each other in memory.

The arrangement of elements of a two-dimensional array in memory is shown on the slide.



The 2D array can be defined and initialized at the same time as shown above. Row size is optional only if the array is defined and initialized at the same time.

- Number of 1D arrays in a 2 dimensional array declaration = number of rows of the 2D array.
- For the definition, `int fdAmt[3][2],` 24 bytes are allocated.
  Note: size of individual element depends on the platform used.
- Memory does not contain rows and columns, both one-dimensional and two-dimensional arrays are stored in contiguous locations.
- A 2-dimensional array is stored "row-wise", in memory. Starting from the first row and ending with the last row, treating each row like a simple array.
- As with the single-dimensional array, each dimension of the array is indexed from zero to its maximum size minus one. The first index selects the row and the second index selects the column within that row. Thus, to access individual elements of a two dimensional array two indices are required.

- If the number of elements stored in the array is less than the actual size of the array, the rest of the memory locations remain unutilized and thus are wasted.

Accessing Elements of a 2D array

Elements of a 2D array can be accessed by using two notations:

1. Subscript notation
2. Pointer notation.

To access any element in a 2D array by using subscript notation is similar to accessing elements in a 1D array. The 2nd element in the 3rd row, can be accessed as `fdAmt[2][1]`.

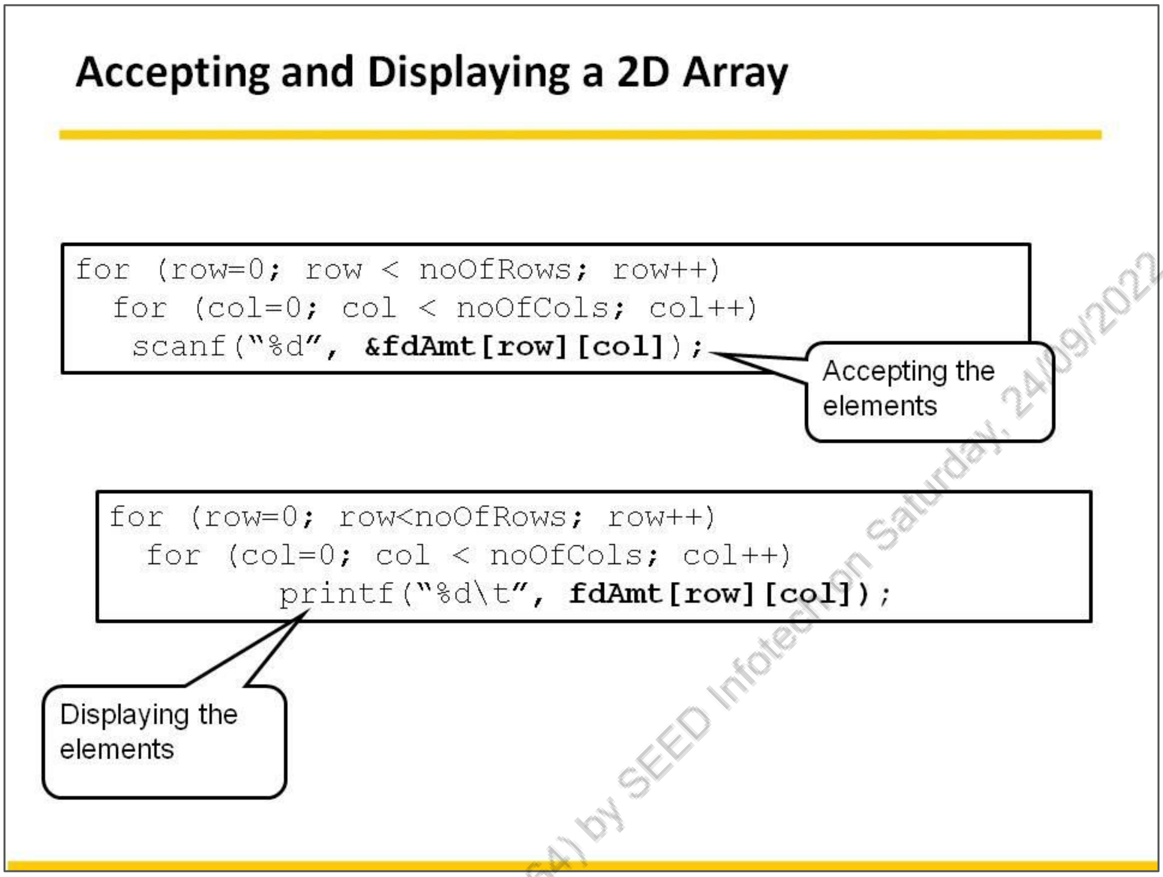
The same element can be accessed using the pointer notation as `(*(*(fdAmt+2)+1))`.

`*(fdAmt+2)` → gives the address of the 3rd row

`*(fdAmt+2)+1` → gives the address of the 2nd element in the 3rd row

`*(*(fdAmt+2)+2)` → gives the value stored at the address of 3rd element in the 3rd row

Give examples of application scenarios where 2D array can be used.



A 2D array can be accepted from the user at runtime. The array is accepted and displayed using nested `for` loops. The inner loop takes care of values in columns whereas the outer loop is used to keep a check on the row size.

The following method can be used to accept and display the array when the column size is fixed and all the columns contain some values.

```
for (row=0; row<noOfRows; row++)
    scanf("%d%d", &fdAmt[row][0],  &fdAmt[row][1]);
for (row=0; row<noOfRows; row++)
    printf("%d\t%d", fdAmt[row][0], fdAmt[row][1]);
```

Arrays can be passed to the function in two ways. In either case, the base address of entire 2D array is passed.

1. Using the subscript operator

```
void dispFdAmtDetails(int [][2],int);        //
function prototype
int main()
```

```
{
    int fdAmt[10][2], noOfCustomers;
    /*accept an array */
    dispFdAmtDetails(fdAmt, noOfCustomers);     //
call to the function
   . . .
}
```

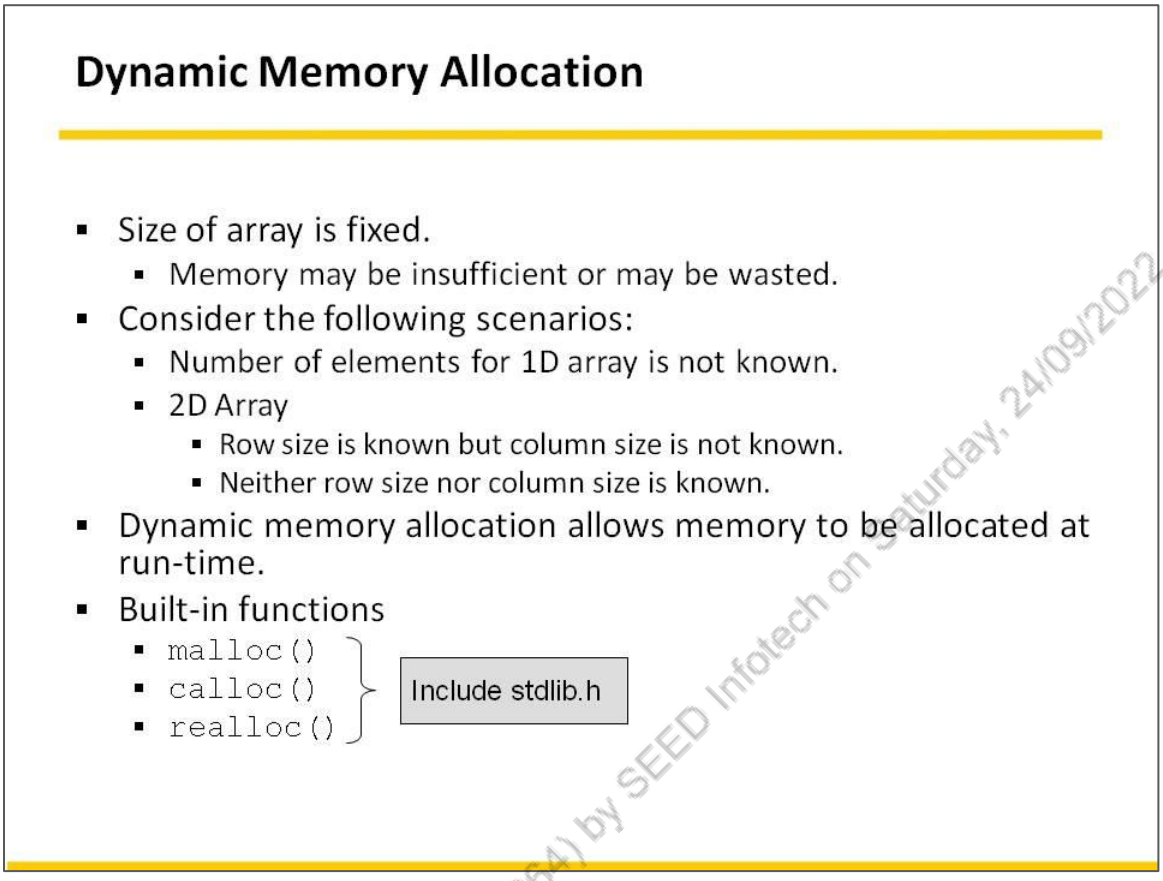
```
void dispFdAmtDetails (int fdAmt[][2], int num) //
func definition
{
 for (row=0; row<num; row++)
    for (col=0; col< 2; col++)
    printf ("%d",fdAmt[row][col]);
}
```

2. Using pointer version

```
void dispFdAmtDetails(int (*pFdAmt)[2],int);
    // function prototype
int main()
{
    int fdAmt[10][2], noOfCustomers;
    /*accept elements of array */
    dispFdAmtDetails (fdAmt, noOfCustomers);
    // call to a function
  . . .
}
void dispFdAmtDetails (int (*pFdAmt)[2], int num)
    // func  definition
{
  for (row=0; i<num; row++)
    for (col=0; col< 2; col++)
     printf ("%d",*(*(pFdAmt+row)+col) );
}
```

`int(*pFdAmt)[2]` is a pointer to an array of two integer values. Why the parentheses? Because, subscript operator (`[]`) has a higher precedence than the dereferencing operator (`*`).

How can you pass a 2D array to a function using a pointer?



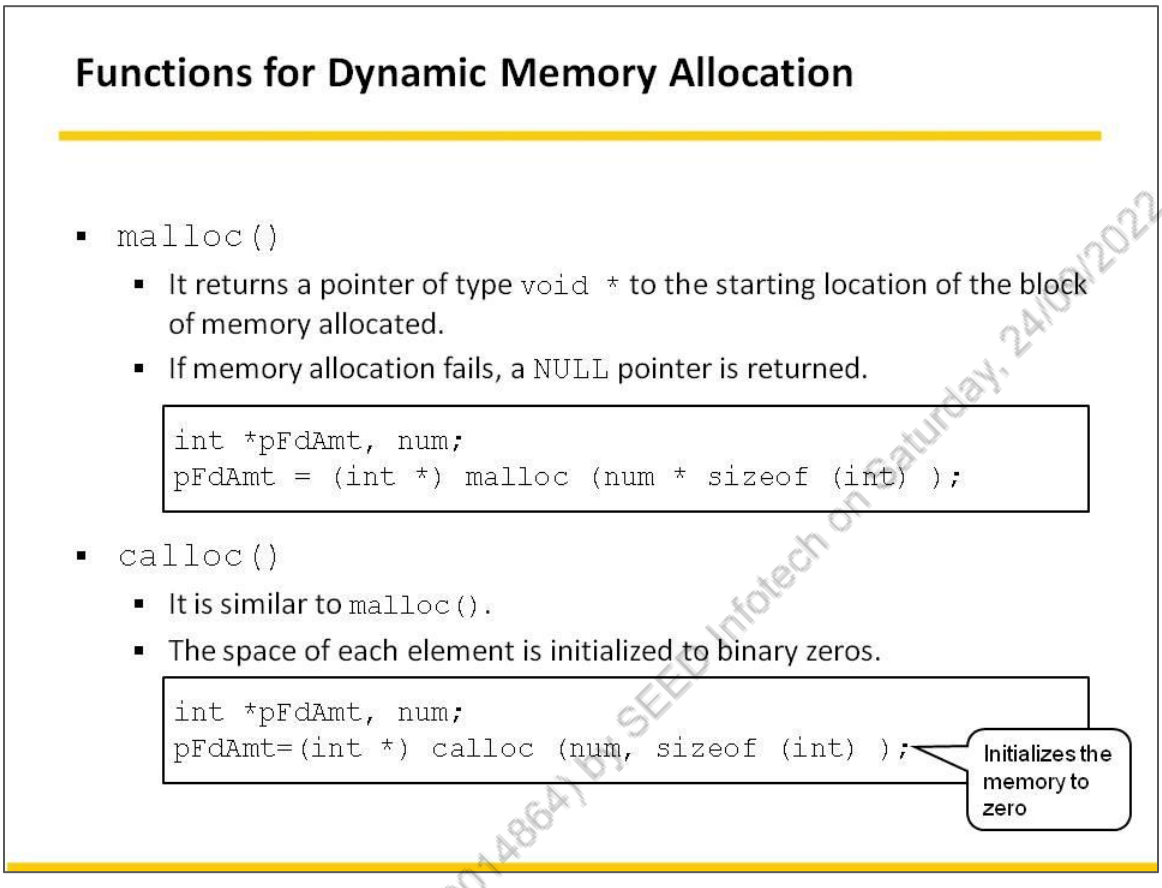
All programs have to set aside enough memory to store the data they use. For the definition, `int fdAmt[20]` the compiler allocates 20 * 4 = 80 bytes of memory for the array. If all 20 memory locations are not used a large amount of memory is wasted. In such a case, memory needs to be allocated at runtime for the required number of elements.

Consider the following scenarios.

1. Row size is known but the column size is not known.
2. Neither row size nor column size is known.

In either of the cases, 2D arrays cannot be declared. This can be done by allocating the memory dynamically to hold the values. Allocating memory at run-time is known as dynamic memory allocation. This memory is allocated on heap. C provides library functions to allocate and free memory dynamically i.e. during program execution.

Functions to allocate memory are `malloc()`, `calloc()` and `realloc()`. The function `free()` is used to free memory that has been dynamically allocated.



The C library functions which are used to allocate memory at run-time allocate a block of memory.

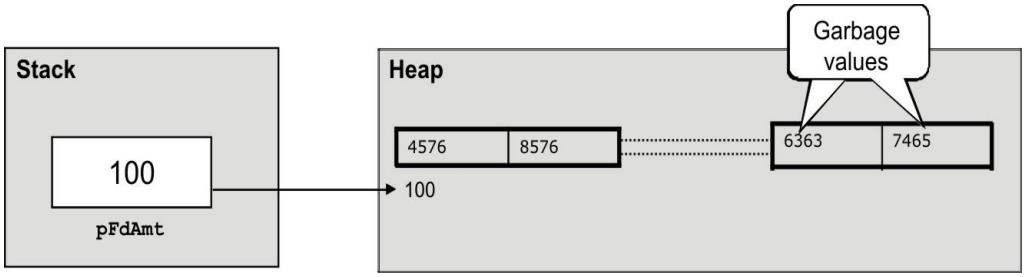
malloc()

This function is used to allocate memory dynamically.

```
void *malloc(size_t * numberOfElements)
```

It allocates a block of memory equivalent to `size_t * numberOfElements` bytes.

It returns a pointer of type `void *` to the starting location of the block of memory allocated. If memory cannot be allocated a `NULL` pointer is returned.




```
int *pFdAmt, num;
pFdAmt = (int *) malloc (num * sizeof (int) );
for(i=0;i<num;i++)
  scanf("%d", &pFdAmt[i]);
```

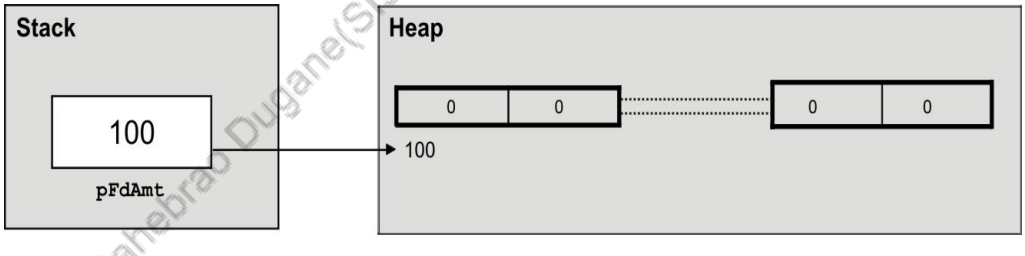
The above statement assigns a block of memory to `pFdAmt` whose size will depend on the value of `num`. The pointer is typecast to integer.

`calloc()`

`calloc()` is also used to allocate the memory dynamically.

```
void *calloc(size_t elements, size_t sz);
```

It allocates space for an array of elements, each of which occupies `sz` bytes of storage. The space of each element is initialized to binary zeros.



In other words, `calloc()` is similar to `malloc()`, except that it handles arrays of elements rather than a single chunk of storage and that it initializes the storage allocated to binary zeroes. The following example allocates memory for an array of 100 integers using `calloc()`.

```
int *pFdAmt = (int*) calloc (100, sizeof(int));
```

realloc ()

This function is used to append new memory to the existing memory block. If 20 bytes of memory have been assigned to `pFdAmt` using `malloc` and later 20 more bytes are required to add to `pFdAmt`, `realloc()` allocates 20 bytes in addition to the existing 20 bytes.

```
void *realloc(void *pFdAmt, size_t s)
```

This function changes the size of space pointed by `pFdAmt` by the amount `s`. The new memory could be adjacent to the existing block or an entirely new block of memory may be allocated, depending on the availability. The contents of the existing block of memory are copied bit by bit to the new memory location.

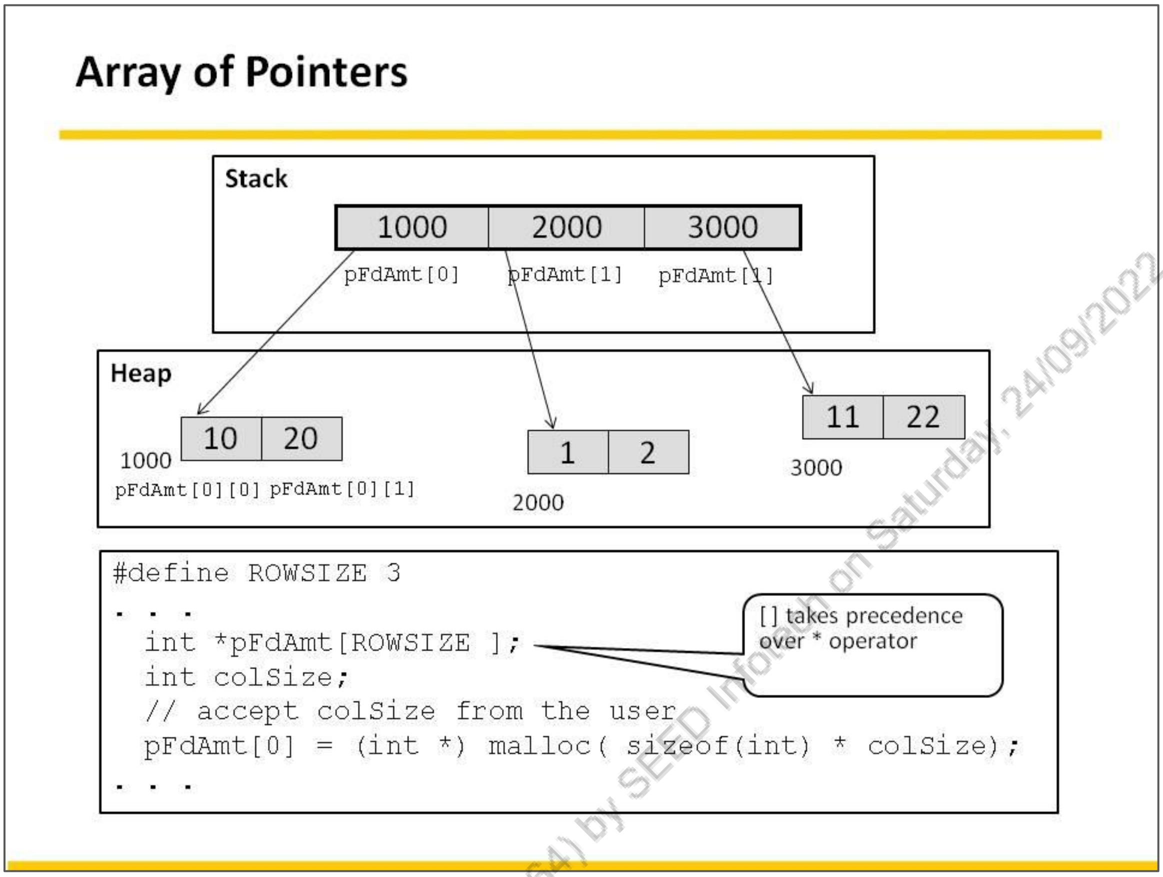
free ()

`free()` frees the specified memory block to be used for another purpose.

```
void free(void *pFdAmt)
```

This function de-allocates the space pointed to by `p`. This must have a value returned by a previous call to `calloc`, `malloc or realloc`.

1. It is the responsibility of programmer to free dynamically allocated memory. If not done, then there will be memory leakage.
2. Do not allocate and de-allocate memory in a loop as this may slow down the program and may sometimes cause security problems.



In some cases of 2D arrays, the size of a column is not known at compile time. Only the row size is known. In such a scenario, array of pointers is declared.

```
int *pFdAmt[ROWSIZE];
```

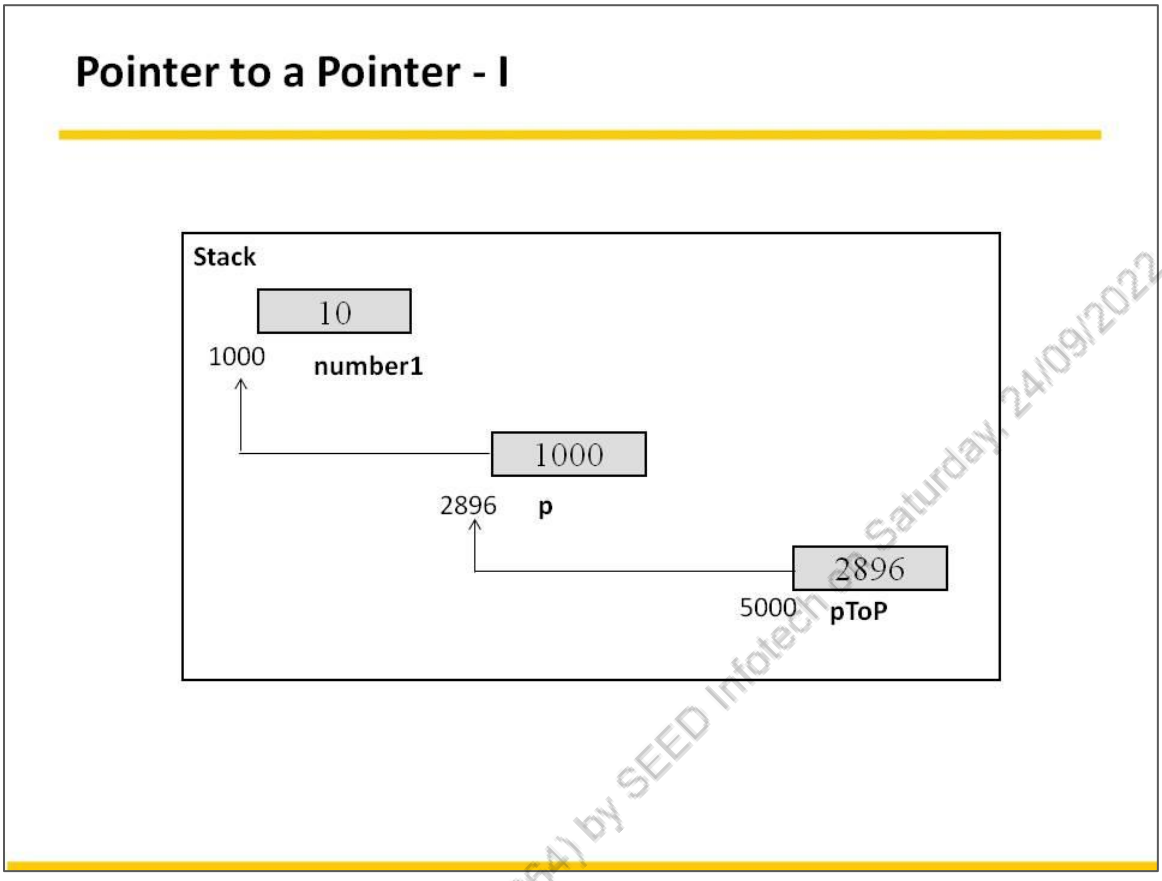
- `[]` operator has higher precedence than `*` operator, hence array of pointers
- In the example, `pFdAmt` is an array of pointers to integers.
- `pFdAmt` is an array of pointers on stack.
- Number of columns is accepted from the user.
- Necessary memory is allocated on heap.

Example

```
#define ROWSIZE 3
int main() {
int *pFdAmt[ROWSIZE] , i=0, row=0,col=0,colSize;
                                  //row size if known
//accept colsize from user
```

```
for(i=0; i < ROWSIZE ; i++)
   pFdAmt[i]=(int*)malloc(sizeof(int) * colSize);
                                    //allocate memory for
each row

for(row = 0; row < ROWSIZE; row++)
  for(col = 0;col < colSize;col++)
    scanf("%d",&pFdAmt[row][col]);
. . .
   for(i=0; i < ROWSIZE ; i++)
     free(pFdAmt[i]);               // free the memory on
heap
}
```

Since `pFdAmt[0]` is a pointer to a block of memory on heap, `(pFdAmt[0]+0)` would be the memory location of the first element of the 1D array and `*(pFdAmt[0]+0)` or `pFdAmt[0][0]` would be the value at that location. Similarly `pFdAmt[0][1]` would be the value at the next location and so on. Thus accessing the elements is similar to 1D arrays.



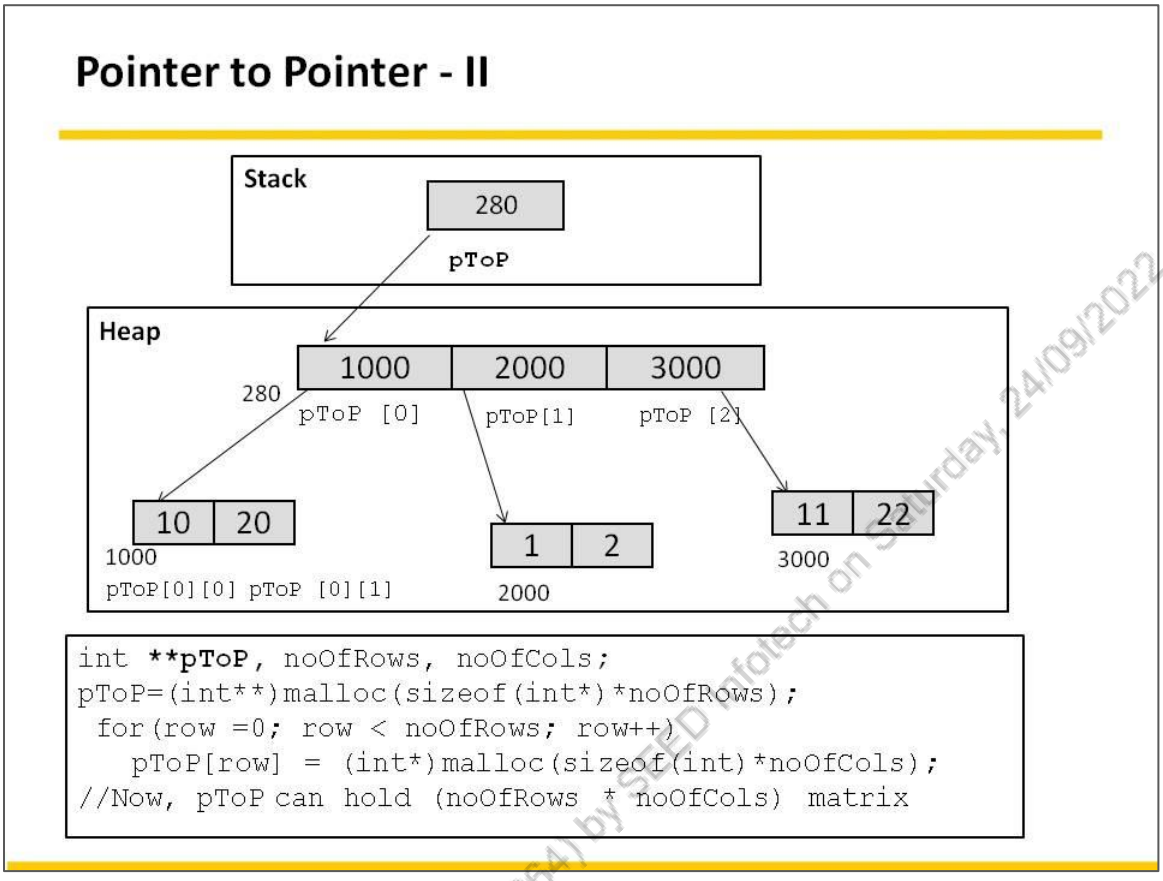
A pointer to a pointer holds the address of another pointer. It is said to be pointing at another pointer like the one shown below.

```
int **pToP;
int number1 = 10;
int *p;
p = &number1;
pToP = &p;
```

- `pToP` is a pointer to a pointer. It holds the address of another pointer.
- `pToP` is on stack.
- To dereference the value, * operator is used.

Consider the pictorial representation of memory locations of variables shown on the slide to study the table given below.

Variable	Value
`number1`	10
`p`	Address of `number1` i.e 1000
`*p`	Value pointed by `p` i.e 10
`pToP`	Address of `p` i.e. 2896
`** pToP` Equivalent to *(*(2896)) Or *(1000)	Value of `number1` i.e 10

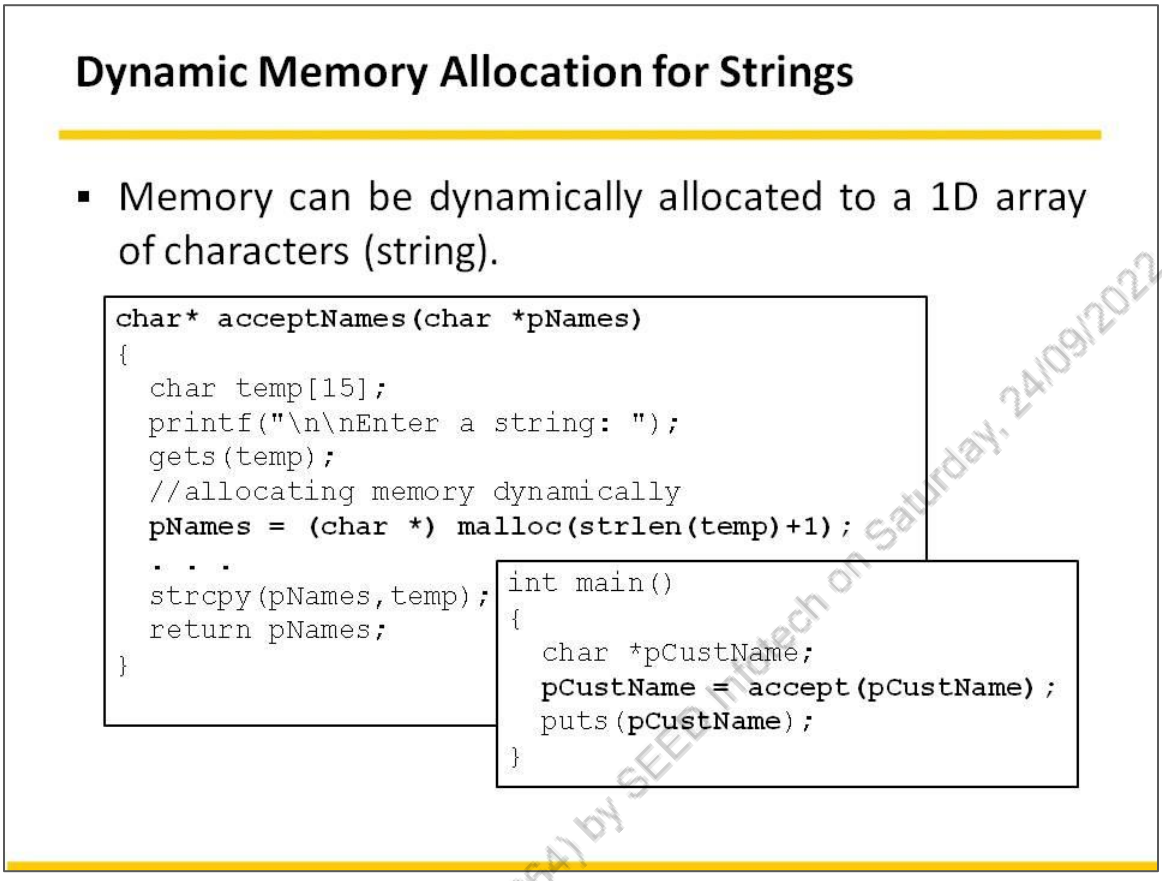


Sometimes the size of rows as well as columns is not known at compile time. In this case too, memory needs to be allocated dynamically. A pointer to a pointer is declared.

```
/*pointer to a pointer */
 int **pToP, noOfRows, noOfCols;
 int row, col;
//accept the number of rows and number of columns from
the user
/* allocate memory for pointers equivalent to the
number of rows */
pToP = (int**)malloc(sizeof(int*) * noOfRows);

/* allocate memory for each row */
 for(row =0; row < noOfRows; row++)
   pToP [row] = (int*)malloc(sizeof(int) * noOfCols);
```

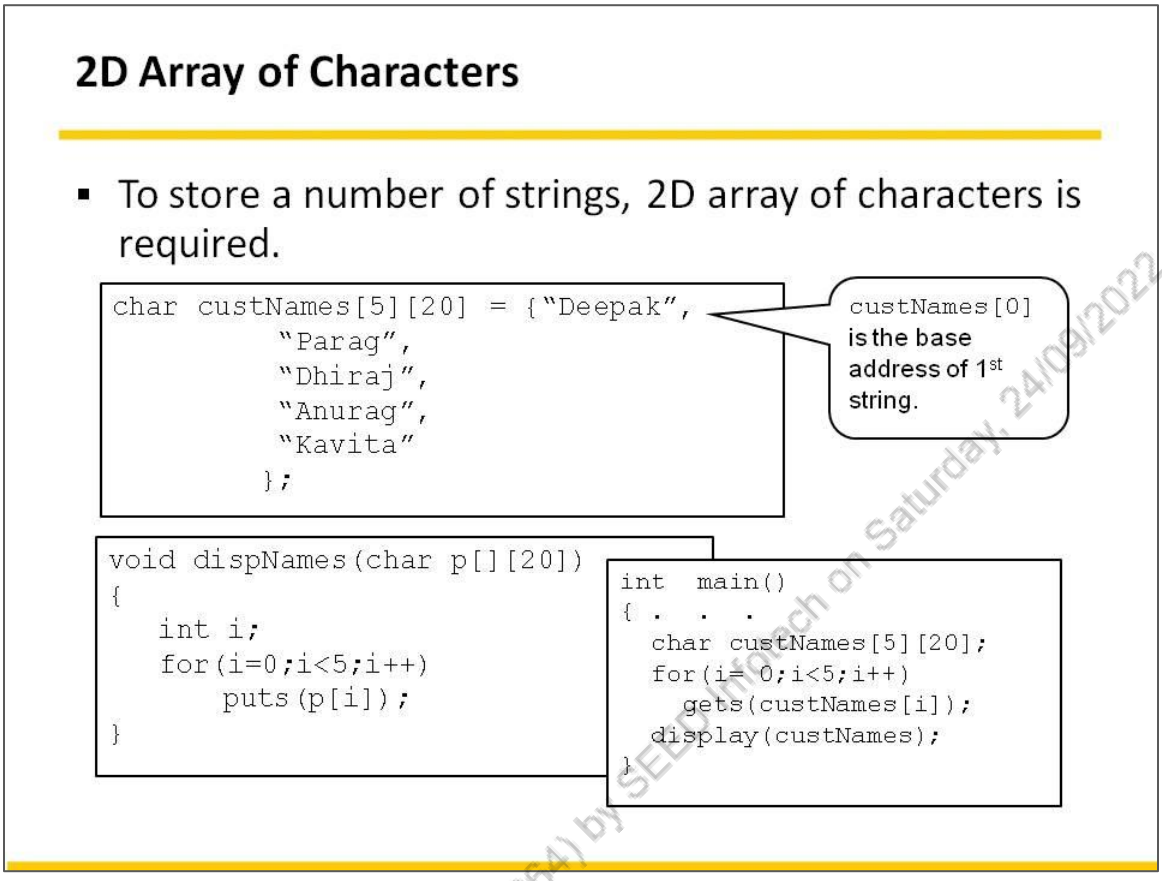
```
//Now, pToP can hold number of noOfRows * noOfCols
matrix
for(row = 0; row < noOfRows; row++)
    for(col = 0;col < noOfCols; col++)
        scanf("%d",(pToP[row]+ col));

for(row =0; row< noOfRows; row++)
    for(col =0;col< noOfCols; col++)
        printf("%d",*(pToP[row]+ col));

// code to free the allocated memory
for (row=0; row<noOfRows; rows++)
   free(pToP[row]);

free(pToP);
```

As seen in the code snippet, `(pToP[row]+col)` is the address where the element is scanned and *(`pToP[row]+col`) is the actual value.



A string in C is said to be a 1D array of characters. Memory can be dynamically allocated to strings too. The same functions - `malloc()`, `calloc()` and `realloc()` are used to allocate memory dynamically to strings. These functions return the generic pointer which is then type cast to `char*`.

The code snippet on the slide demonstrates this. The function `acceptNames()` is passed a pointer to character (`*pNames`) as an argument from `main()`. In the function, `malloc()` is used to allocate memory dynamically. The generic pointer returned by `malloc()` is stored in `pNames`. This is returned by value to the calling function `main()`. The string accepted in the function is then displayed in `main()`.



Like integers, strings can also be stored as 2D array of characters. A number of strings are stored at contiguous memory locations. The array can be declared and initialized as shown below.

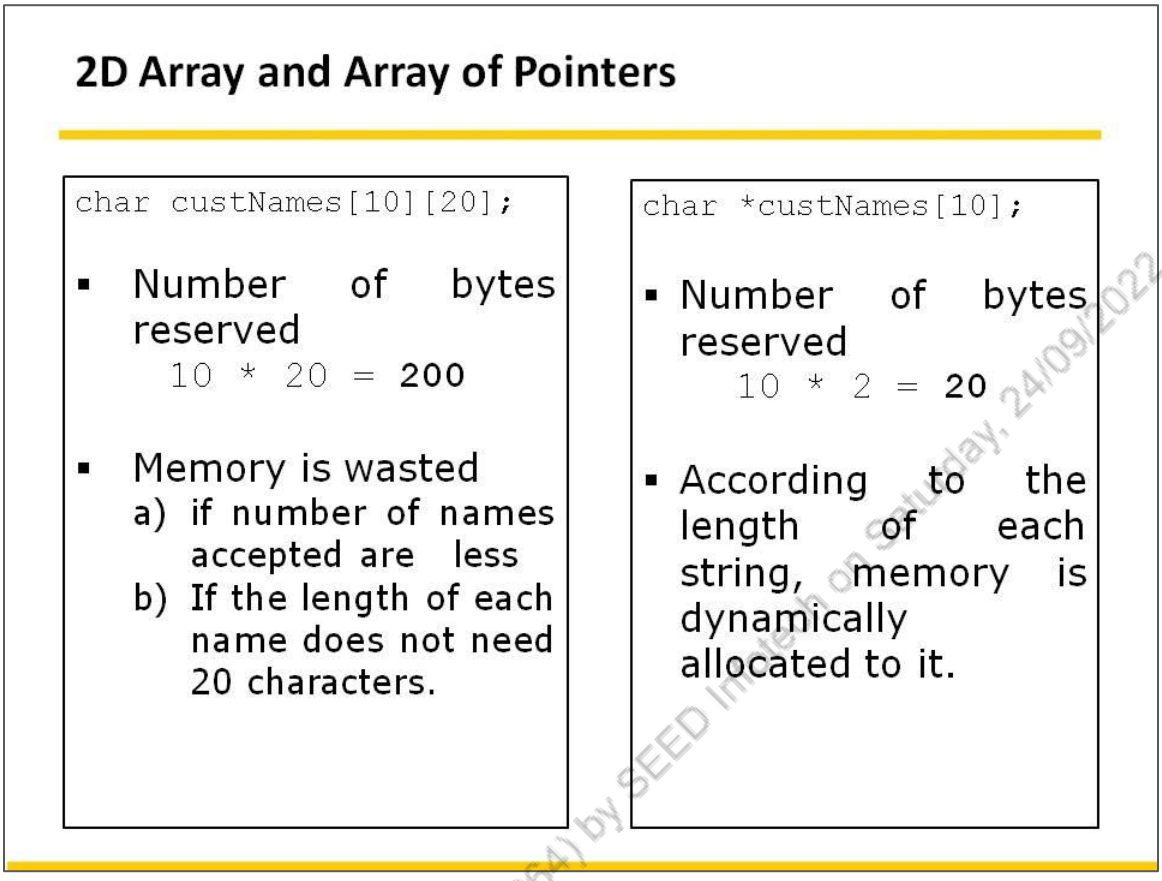
```
char custNames[5][20] = {"Deepak", "Parag", "Dhiraj",
"Anurag", "Kavita" };
```

The following code snippet shows how to accept and display strings.

```
int main()
{ . . .
  char custNames[5][20];
  for(i= 0;i<5;i++)
    gets(custNames[i]);
  dispCustNames(custNames);
}
void dispCustNames(char cuNames[][20])
{
   int i;
```

```
    for(i=0;i<5;i++)
      puts(cuNames[i]);
}
```

`custNames[0]` is the first string, `custNames[1]` is the second string, and so on. Each string is allocated 20 characters.



Allocating memory for strings in 2D array often leads to waste of memory. It can be due to less number of strings accepted or if the length of each string is less than the number of bytes allocated.

In the example mentioned above, memory for 10 strings is allocated and each string is 20 characters long. A total of 200 bytes are allocated. If the number of strings accepted is 5, then 100 bytes will be wasted. Also the length of each name may not be equal to 20.

Both these issues can be addressed by the use of pointers.

- If the number of strings (rows) is not known at compile time, pointer to pointer declaration is helpful. Allocate the memory in the same way as was declared for 2D array of integers earlier in the chapter.

```
void acceptNames(char **names)
{
  int num, i;
  char temp[25];
```

```
  printf("\nEnter number of strings : ");
  scanf("%d", &num);
  names = (char**)malloc(sizeof(char*)*num);     //for
rows
  for(i=0; i<num; i++)
  {
    printf("\nEnter string : ");
    fflush(stdin);
    gets(temp);
    names[i] = (char*)malloc(strlen(temp)+1);    //for
columns, as required
    strcpy(names[i],temp);
  }
/* code to free memory*/
. . .
}
```

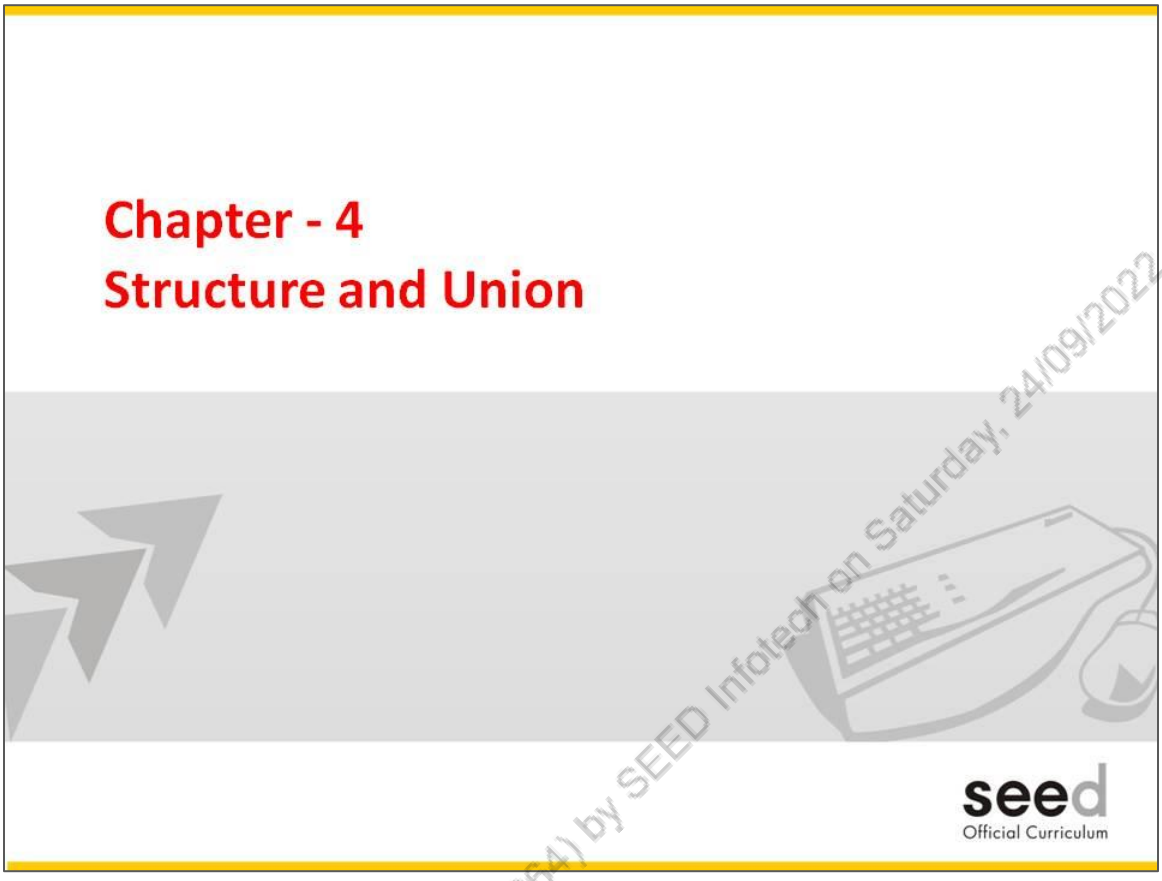
- Memory can also be dynamically allocated for each string depending on its length as shown in the snippet below.

```
int main()
{ .  .  .
  char *pCustNames[10];
  acceptNames(pCustNames);
  //code to display the strings
  . . .
  //free the dynamically allocated memory when not
required
  . . .
}

void acceptNames(char *name[])
{
   char temp[20];
   for(i = 0;i < 10; i++)
   {
    gets(temp);
```

```
        name[i]=(char*)malloc(sizeof(char)*
strlen(temp));
        strcpy(name[i],temp);
    }
}
```

Read and implement how pointer to pointer concept can be applied to strings. Check the link given below:

http://www.taranets.net/cgi/ts/1.37/ts.ws.pl?w=329;b=283



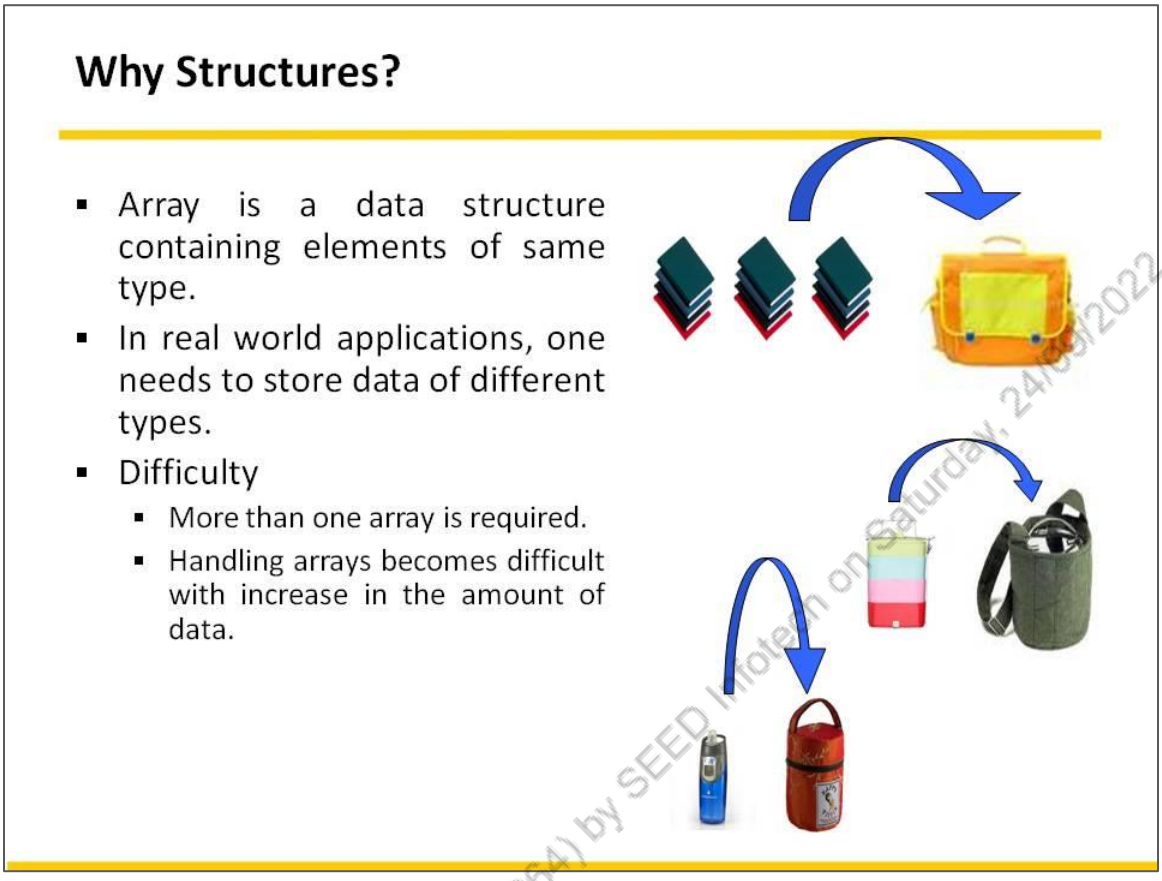
This chapter covers structure in detail - to define a structure and use it in a program, pass structure to function, array of structures, and nested structures. The chapter also covers unions.

Objectives

At the end of this chapter you will be able to:

- List limitations of Array.
- Define a structure and state its use.
- Use `typedef` to assign a new name to existing data types.
- Construct a program to store and display data of a structure.
- Construct a program to pass a structure by value and by address to a function.
- Declare an array of structures to store data.
- Declare and use nested structure.
- Allocate memory dynamically to a structure.

- Define a union and state its use.
- Declare and use a function pointer.
- Differentiate between structure and union.
- Construct a program that declares and uses a union.
- Define `enum` and construct a program that uses it.



Array is a data structure that contains elements of the same type. But in real world applications, different types of information are needed to be stored. For example, to store information about a customer, customer id, name, account balance, etc will be stored. If an array is defined for each of these attributes, handling a number of records at a time is a problem and also many arrays will have to be defined. The following program shows how a record of four customers is kept using arrays.

```
/* Program to process an object which contains entities
of different types. Arrays are used to do the
processing */
#include<stdio.h>
#include<string.h>
main()
{
  char *custName[4],temp[25];
  int custId[4];
  float balAmt[4];
```

```
  int j, len;
  printf("\nEnter name, customer id and balance for 4
customers :");
  for(j = 0; j < 4; j++)
  {
       scanf("%s%d%f", temp, &custId[j], &balAmt[j]);
       len = strlen(temp);
     custName[j] = (char*)malloc(len);
       strcpy(custName[j],temp);
       printf ("\n");
  }
  printf("\nOutput ");
  for(j = 0; j < 4; j++)
    printf("\n%s\t%d\t%f ", custName[j], custId[j],
balAmt[j]);
  for(j=0;j<=3;j++)
     free(custName[j]);
}
```

Since, 3 attributes have been considered: customer name, id and account balance only 3 arrays are required. The number of arrays will go on increasing as more and more items are added. Maintenance will then become difficult. This problem can be solved by defining a structure.

What is a Structure?

- User defined data type.
- Convenient way of grouping several pieces of related information together.
- May contain any number of members of different types.

Customer		Saving Account	
Field	**Data type**	**Field**	**Data type**
Customer name	Character	Customer ID	Integer
Address	Character	Name	Character
Balance amount	Float	Balance amount	Float

Structure is a user defined type that maps to a real world entity. Entity could be tangible like student, customer, table, or intangible like date, or transaction. Structure is a convenient way of grouping several pieces of related information together.

These entities have a set of characteristics represented by structure members.

Structures are useful, not only because they can hold different types of variables, but also because they can form the basis for more complex constructions, such as linked lists.

A structure may contain any number of members of different types. The programmer must define what a particular structure would look like. It is called as structure definition.

```
struct <tagName>
{
  member 1;
  member 2;
  …
  member n;
};
```

- tagName is a name that identifies structure. It should be singular, representing Employee, Book, Point, File, and so on.
- The definition ends with a semicolon.
- Each member is declared with its own data type. Array, integer, float, pointer, etc. can be members of a structure. The name of each member should be distinct.
- The members cannot be initialized inside structure declaration.
- There is no formal distinction between a structure definition and a structure declaration; the terms are used interchangeably.
- In large programs, structures are usually defined in header files.

Structures for Customer and for Saving Account and Fixed Deposit are defined in the following way:

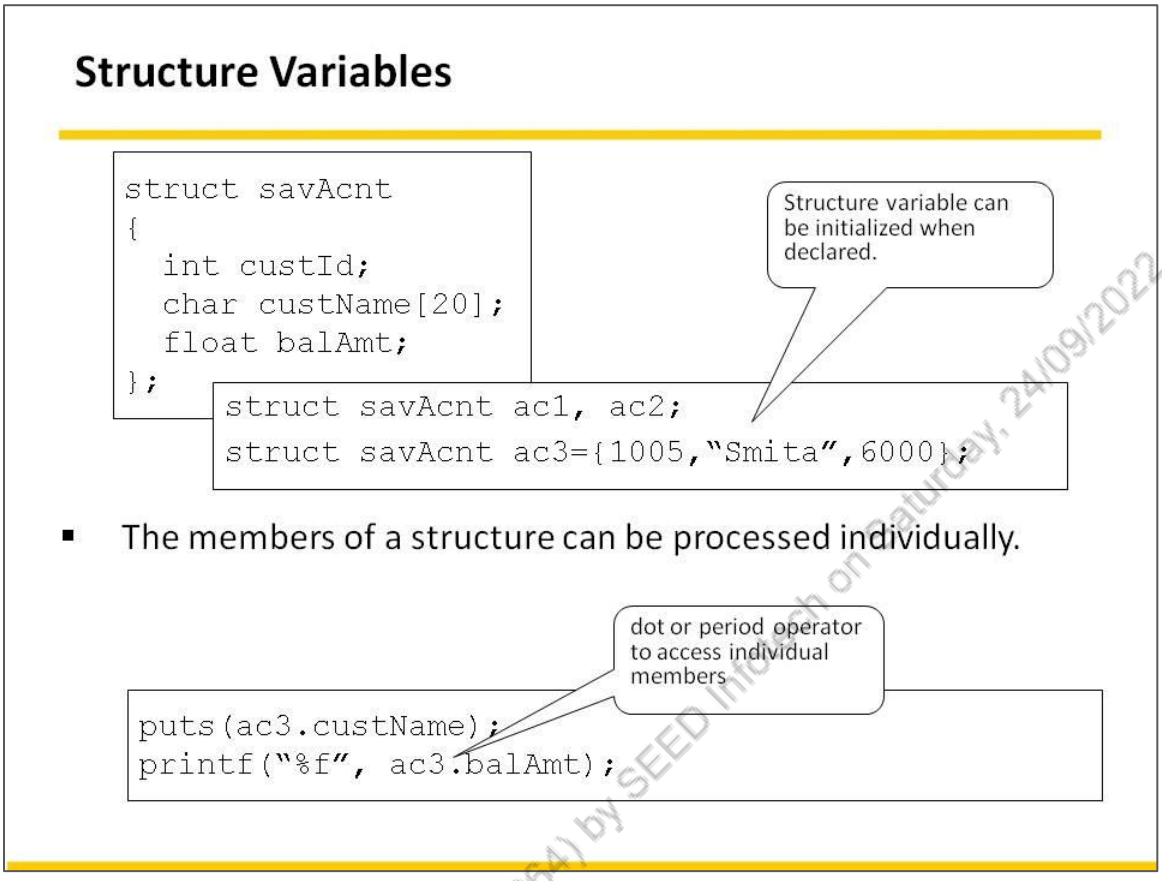
```
struct customer
{
    char custName[20];
    int custId;
    float amt;
    char address[25];
    char phone[10];
    char typeOfAccount[10];
};
```



```
struct savAccount
{
    int custId;
    char acNo[10];
    float balAmt;
};
```

```
struct fixDeposit
{
    int custId;
    char acNo[10];
    int amt;
};
```

Name any 3 real-world entities and define their characteristics.



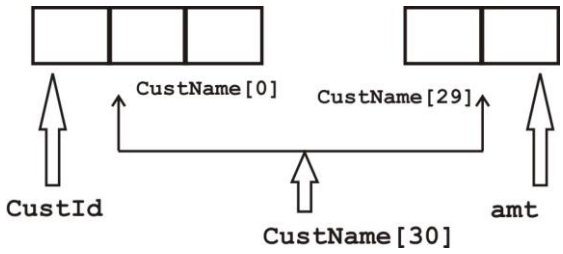
Structure declaration acts as a template. Structure variable has to be defined to hold values.

```
storage-class struct <tag> variable1, variable2,….,
variable n;
```

For example,

```
struct savAcnt s1,s2;
struct savAcnt s3 = {1, "Gita", 6000};
```

- `s1, s2 and s3` are called structure variables.
- Like other variables, they can be initialized at the time of definition. The order of values enclosed in braces must match the order of members in the structure definition.
- Memory allocated for each variable is the sum of bytes required to hold each member.



For example, memory space assigned for `s1` = 4 + 30 + 4 (4 for integer + 30 for character array + 4 for float) = 38 bytes

- A structure variable can also be defined like as shown below.

```
struct savAcnt
{
  char custName[30];
  int custId;
  float balAmt;
} emp1;
```

- Individual members of a structure can be accessed or processed in the code using 'dot operator' or 'period operator' with a structure variable. The individual members are treated as simple variables.

```
int main()
{
   struct savAcnt s1;
   struct savAcnt s2 = { 102, "Amit", 15000};
   printf("Enter customer id: ");
   scanf("%d",&s1.custId);
   printf("Enter name of the customer: ");
   gets(s1.custName);
   printf("Enter the amt: ");
   scanf("%f",&s1.balAmt);

  // print the details of second employee
    printf("%d", s2.customer Id);
    puts(s2.custName);
    printf("%f", s2.balAmt);
 . . .
}
```

- `&s1` gives the address of the entire structure. Individual members are accepted from the user at their address: `&s1.custId`, `&s1.balAmt`, and `s1.custName`.

typedef

- Assigning a new name to an existing data type or a complex user defined type.
- Variables can be declared using the new type name.

```
typedef  int  tool;
tool x , y;
```

- Declaring the same structure using `typedef`

```
typedef struct{
   int custId;
   char custName[20];
   float balAmt;
}savAcnt;
savAcnt s1, s2;
```

Declaring variables using the new name

At this point the feature called `typedef` has been introduced. Using `typedef`, a new name can be assigned to an existing data-type or a complex data-type. Variables can be defined using this user defined data type after it is established. No new data type can be created. You can use `typedef` with only existing data types, whether built in or user defined.

Its general form is

```
typedef type new-type;
```

Example

```
typedef int tool;
tool x, y, z; // in place of  int x, y, z;
```

`tool` is the new name given to `int` data type and can be used in place of `int`.

`typedef` feature is particularly convenient for structures since its use eliminates the repeated use of `struct` tag while defining variables. Also the new name often suggests the purpose of the structure.

Example,

```
typedef struct{
   int custId;
   char custName[20];
   float balAmt;
}savAcnt s1, s2;
```

Variables of the above mentioned structure can be declared as

```
savAcnt s1, s2, s3;
```

Besides purely aesthetic issues, there are two main reasons for using typedefs:

1. The first is to parameterize a program against portability problems. If typedefs are used for data types that may be machine dependent, only the typedefs need to be changed when the program is moved. Types like `size_t` and `ptrdiff_t` from the standard library are examples.
2. The second purpose is to provide better documentation and readability in a program.

Assignment of Structure Variables

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

```
struct point p1 = { 5,10 } , p2;

 p2 = p1;    // structure variables can be
             //assigned like any other variables
```


Two variables of the same structure type can be copied in the same way as ordinary variables. If `p1` and `p2` are variables of same structure, then the following statement is valid.

```
p1 = p2;
```

This statement copies the entire structure into another structure. However, member wise copy takes place; wherein the values of each member are copied bit by bit.

The following statements are not permitted. C does not permit any logical operations on structure variables.

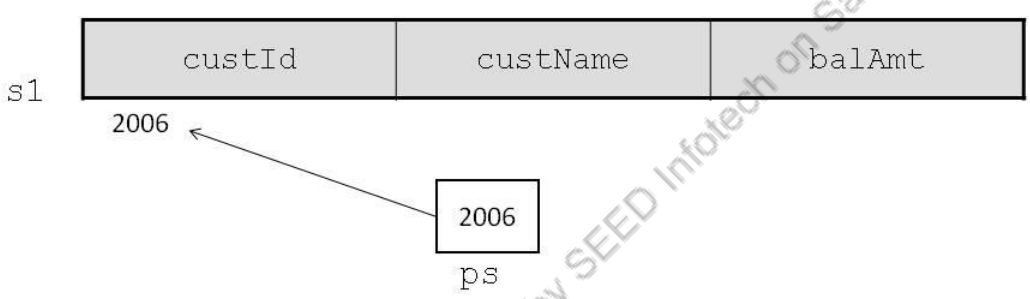
```
p1 == p2
p1 != p2
```

In case, it is needed to compare them, it may be done by comparing members individually.

Pointer to a Structure

```
savAcnt s1;
savAcnt *pS;
pS = &s1;

...

puts(pS→custName);  //accessing members using pointer
```

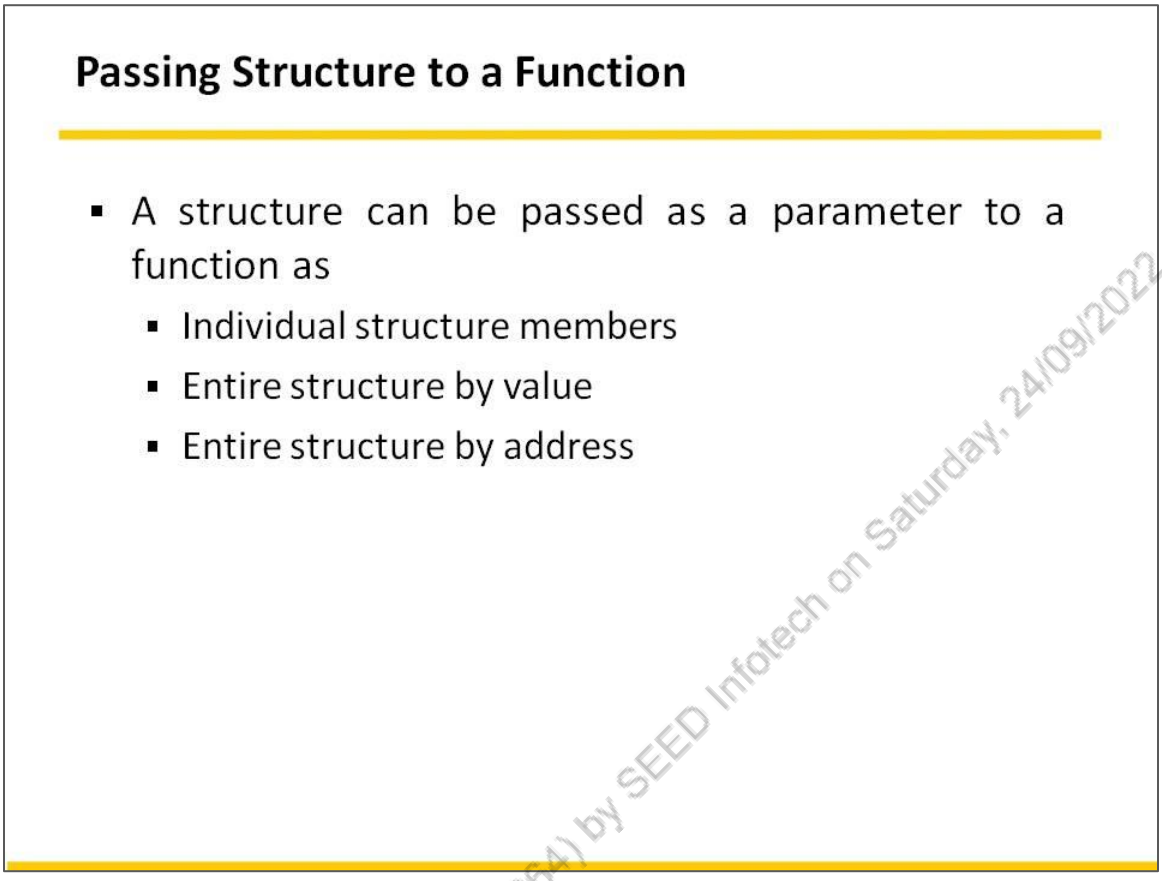
Pointer to a structure



Like pointer to variables of built-in data types, pointer to a structure can also be declared. The pointer points to the entire structure. This is useful when the structure is passed to a function. Only the base address of the structure is passed to the function instead of entire structure by value.

In the example mentioned on the slide, some arbitrary value 2006 is the address of the entire structure `s1`. Pointer `pS` points to a structure `s1`. The members of the structure are accessed using the structure pointer operator → also called the arrow operator.

```
int main()
{
   Employee s1 = {101, "Amit", 15000};
   Employee *pS = &s1;
   printf("%d", pS→custId );
   puts(pS→custName);
   printf("%f", pS→balAmt );
}
```



Structure can be passed to a function as parameter and can also be returned from functions using three approaches.

1. Individual members of structure
2. Entire structure by value
3. Address of the structure or pointer to structure

Passing individual members

Since individual structure members can be accessed and act as individual entities, they can be passed and returned from functions.

```
//function prototype
float calcInt(int, char *, float);
int main()
{ . . .
  savAcnt s1 = {105, "Snehal", 17000};
  s1.balAmt = calcInt(s1.custId, s1.custName,
s1.balAmt); // function call
```

```
   . . .
}
float calcInt (int id, char *nm ,float amt) {
   float interest;
   . . . //code to calculate simple interest
   . . .
   return interest;
}
```

Thus it is seen that passing individual members to and from functions is analogous to ordinary variables.

Passing Entire Structure by Value

Passing a large number of parameters to a function is tedious and also against good coding practices. C permits an entire structure to be passed to a function as a parameter and also to return it via the return statement. The entire structure is passed by value to a function. The function operates on the local copy of actual data passed. So it becomes necessary to return the structure by value to the calling function if the changes are to be reflected in the calling function. The returned structure is copied member-wise. The code snippet mentioned below explains this.

```
savAcnt getInfo(savAcnt);    // passing structure by
value
void dispInfo(savAcnt e);
int main()
{
   savAcnt s1;
   s1 = getInfo(s1);
   dispInfo(s1);
    . . .
}
savAcnt getInfo(savAcnt s)
{
  scanf("%d", &s.custId);
  // scan other data
  return s;          // returning a structure by value
}
```

```
void dispInfo(savAcnt e)     // returning a structure
not required
{
  printf("%d", s.custId);
  . . .
}
```

In the example mentioned above, `dispInfo()` function does not modify the data. It accepts a structure variable as a parameter that is passed by value and just displays the data. So the return type is `void`. But `getInfo()` function modifies the data. It accepts the value of the data members of the `savAcnt` structure. These values need to be reflected in the calling function, that is `main()` in this case. Since the structure variable is passed by value, the local copy '`s`' has to be returned to `main()`.

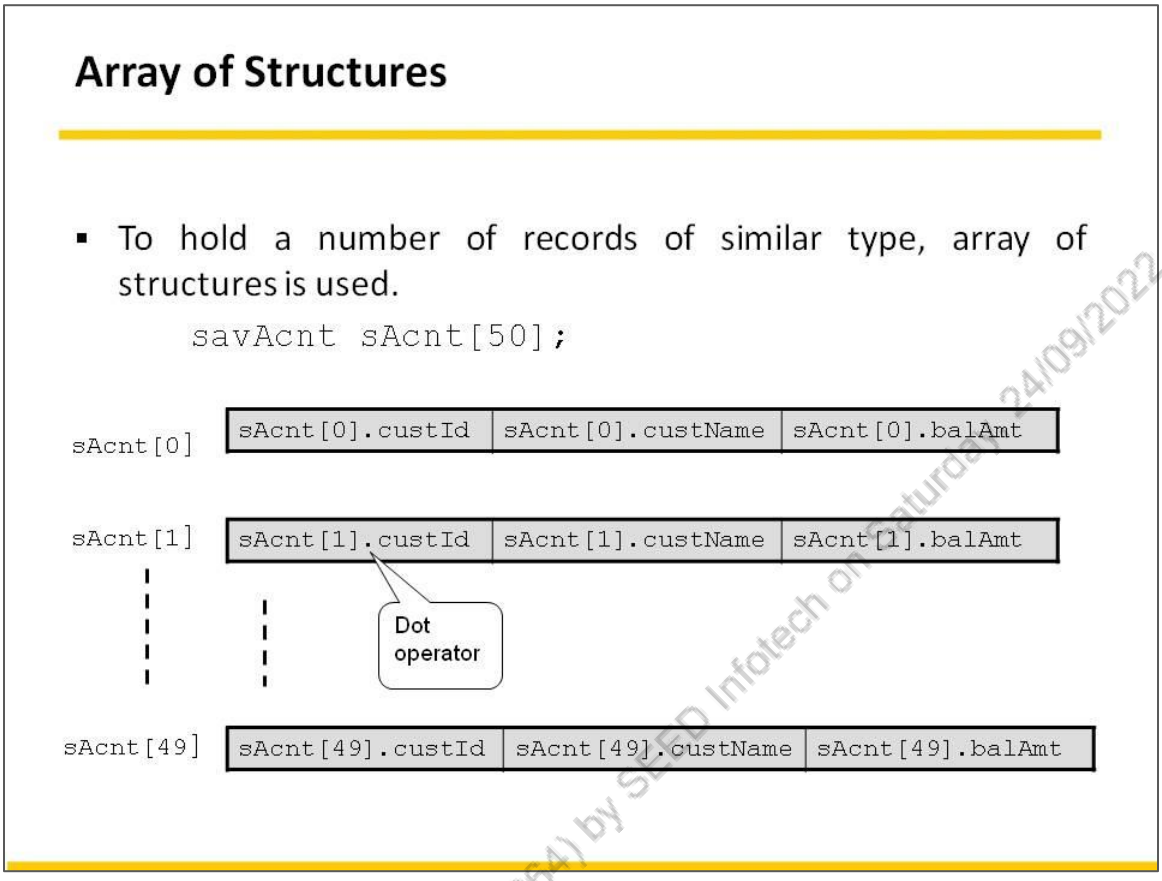
Passing Entire Structure by Address

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. A complete structure can be transferred to a function by passing the address of the structure. The function can indirectly access the entire structure and work on it. Hence, if any of the structure members are modified within the function, the modifications will be recognized outside the function. The code snippet mentioned below explains this.

```
void getInfo (savAcnt *);    //prototype
int main()
{
     . . .
   getInfo(&s1);
}
void getInfo (savAcnt *pS)
{
  scanf("%d", &pS→custId);
  gets(pS→custName);
  . . .// scan other data
}
```

A pointer to a structure can also be returned from a function. This is useful when several structures are passed to the function and only one is returned.

Pass the structure by address to a function if its size is large. This avoids creating local copy and also saves time.



A structure variable can be said to represent a record. If a number of records are to be stored, then the obvious choice would be an array. Array of structures would hold all records of same type at contiguous memory locations. Operations on the array are performed just like array of built – in data types.

The general format for array definition is

```
Data-type arrayName[size of array];
```

For example, to store records of 50 employees, an array of structure employee is declared. The array can be initialized at the time of declaration.

```
savAcnt sAcnt[50];
savAcnt cList[3] = { {101, "Amit", 1500}, {102,
"Chetan", -200}, {103,
                        "Abhishek", 1250} };
```

Array elements are accessed using index of arrays. In case of array of structures, elements of array are structures. Each member of structure is accessed by '.' (dot) operator. Hence a member can be obtained using

```
arrayName[index].memberName
```

For example, `sAcnt[2]` represents the entire structure for third customer. To access the name of the third customer in the array, `sAcnt[2].custName` is written. The following code snippet would clarify the usage.

```
int main()
{
  savAcnt sAcnt[50];
  . . .
  for( i=0;i<50;i++)
  {
    // prompt the user to enter the details
    scanf("%d", &sAcnt[i].custId);
    gets(sAcnt[i].custName);
    scanf("%f", &sAcnt[i].balAmt);
  }
. . . // code to print the records
}
```

Pointer to array of structures is easy to operate over a set of records.

```
int main()
{
  savAcnt sAcnt[10];
  savAcnt *pS = sAcnt;
  // accepting data for sAcnt
  for( i=0;i<10;i++)
  {
     printf("%d",pS→custId);
     puts(pS→custName);
     printf("%f", pS→balAmt);
     pS++;
  }
}
```

Nested Structures

- One structure can be a member of another structure.

```
struct address
{
  char phone[15];
  char city[15];
  int pin;
};
```

```
typedef struct
{
    int custId;
    . . .
    struct address addr;
}savAcnt;
```

```
int main()
{
. . . .
savAcnt s1;
puts(s1.addr.city); // to access the city
 . . ..
}
```


Member of a structure can be a structure itself. When a member of a structure has to be further broken down into entities, that member can be declared as a structure. Such a structure is said to be nested within the other structure. This is called nesting of a structure. Suppose address of an employee is to be added in the employee records and the address consists of phone, city and pin. Hence, a nested structure can be defined as

```
typedef struct
{
   char phone[15];
   char city[15];
   int pin;
}address;
typedef struct
{
    int custId;
    . . .
```

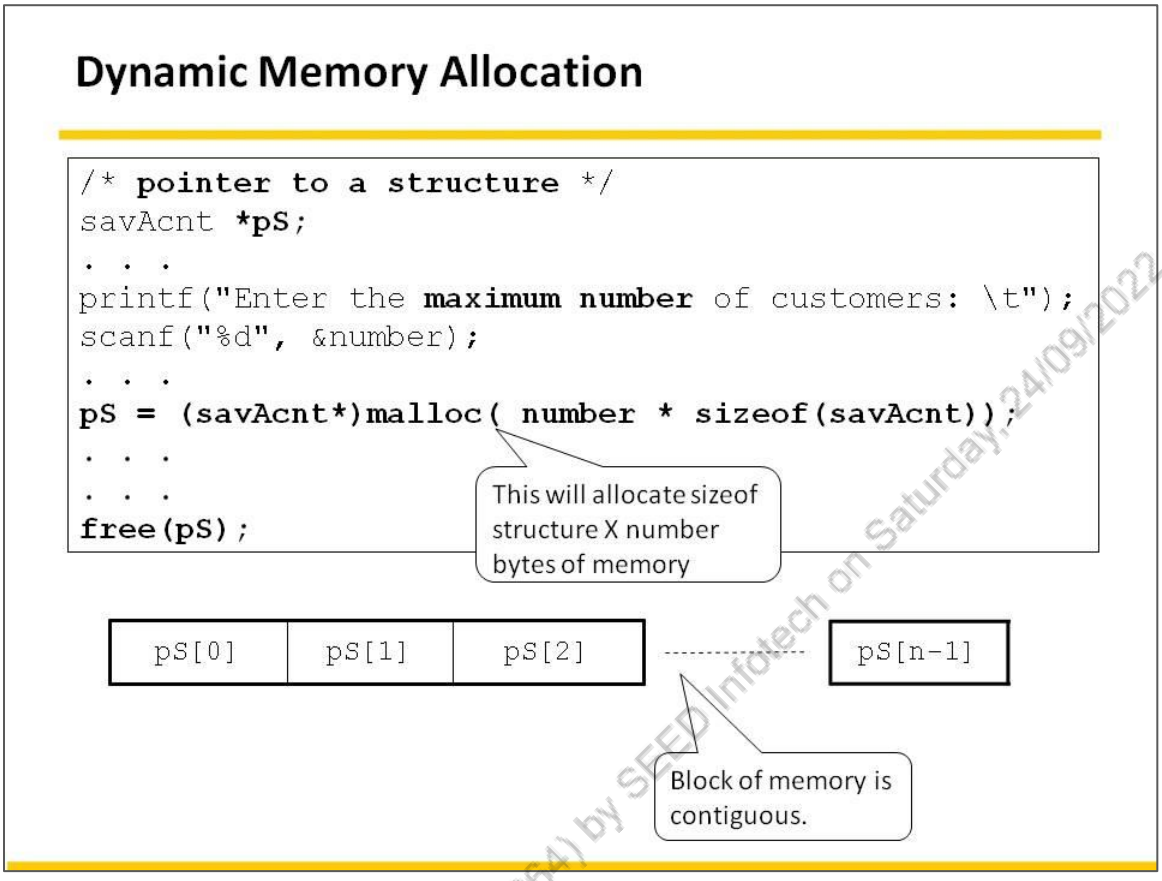
```
    address addr;
}savAcnt;
```

Thus member 'addr' is itself a structure. In such a case, declaration of embedded structure must appear before the declaration of the outer structure. A variable of type savAcnt can now be initialized as:

```
savAcnt s1 = {122, "Jai", 18844.70, "022-5334716",
"Mumbai", 400421};
```

The members of embedded structures can be obtained with an additional dot operator. The city from employee's address can be obtained by writing s1.addr.city.

Nested structures can be a powerful way to create a complex data type.



Structures are user defined types that can be mapped to records. This is done by defining an array. But sometimes the number of records is not known at compile time. It can be obtained from the user at runtime. For this, memory should be allocated dynamically as shown in the example mentioned below.

```
/* pointer to a structure */
  savAcnt * pS;
  . . .
  printf("Enter the maximum number of customers: ");
  scanf("%d", &number);
  . . .
  pS = (savAcnt*)malloc( number * sizeof(savAcnt));
  // get the data from the user
  scanf("%d", &pS[0].custId);
  gets(pS[0].custName);
  scanf("%f", &pS[0].balAmt);
  . . .
  // print the data
```

```
  puts(pS[0].custName); // represents the name of first
employee
  . . .
  // free the memory
  free(pS)
```

Union

- A user defined type that stores different data types in the same memory space (but not simultaneously).
- Declared in the same way as structures.
- Size of the union is the size of its largest data type.

```
union table
{
  int number;
  char alphabet;
  double temperature;
};
```

Size of union will be the largest data type

Unions are similar to structures as both are used to group number of different types of elements. But union stores different data types in the same memory space (though not simultaneously). The size of the union is the size of its largest data type. A typical use is a table designed to hold a mixture of types in some order that is neither regular nor known in advance.

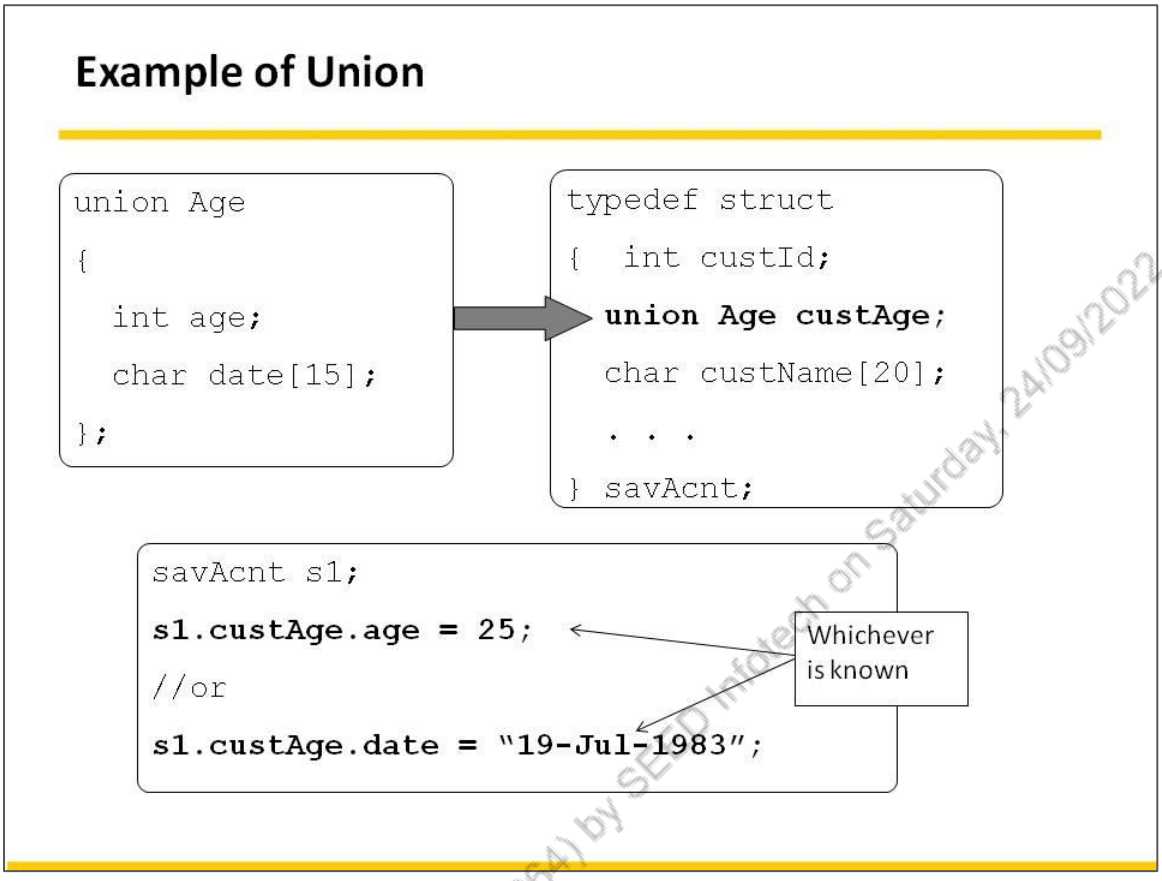
Unions are declared in the same way as a structure except that the keyword `union` is used.

```
union tag
{
 member 1;
 member 2;
 . . .
}unionVariables;
```

- Union elements are accessed using the same method used for accessing structure elements, using the dot or period (.) operator or arrow (→) operator.
- The name of a structure member represents the offset of that member from the start of the structure. In a union all members start at the same location in memory.
- The members of a union may themselves be structures and the members of a structure may themselves be unions.
- The operations performed on unions are same as on structures - accessing union members, taking address, assignment of entire union variable to other union variable.

Use of union

Union aids in the production of machine-independent code because the compiler keeps the track of the actual sizes of the variables which make up the union.



In the example mentioned on the slide, union is a member of a structure. The union defined has two members - `age` and `date`. The size of this union is 15 bytes. While using the structure, either of the members of the union can be assigned a value, depending on the value known by the user.

Use a union when you will be storing the value of only one member of the union at any given point of time. Use a structure when you have to store the values of all the members of the structure.

enum

- A user defined data type which consists of a set of named integer constants.

```
enum acType{ SAVING = 100, FIXED, RECURRING};
```

- Each member starts from 0 by default and is incremented by 1 for each next member.

```
enum acType at;
at = SAVING;
printf("%d", at);  // prints 100
```

The enumerated type is a user defined type used to declare symbolic names to represent integer constants. By using the `enum` keyword a new "type" can be created and the values it may have can be specified. The purpose of enumerated types is to enhance the readability of a program. It is a convention to write the values in capital letters as mentioned on the slide. By default, the constants in the enumeration list are assigned the integer values 0, 1, 2 and so on.

In general, enumerations may be defined as

```
enum tag {member1, member2,….,membern};
```

`enum` is the required keyword, tag is the name of the enumeration and `member1,…. membern` are members (identifiers). Each member is a constant. The constants represent integer values. The member names must differ from each other.

Similar to structures once an enumeration has been defined, variables of that type may be defined.

```
enum tag var1, var2, . . .varn;
```

Definition of `enum` and variable declaration may be combined together.

```
enum tag {mem1, mem2,…..,memn} var1, var2;
//or
enum {mem1, mem2,…..,memn} var1, var2;
```

For example,

```
enum colors {BLACK, BLUE, CYAN, GREEN, MAGENTA, RED,
WHITE} background;
enum {BLACK, BLUE, CYAN, GREEN, MAGENTA, RED}
background, foreground;
```

`colors` is the name of the enumeration.

`black, blue` are the members of the enumeration which are called enumeration constants.

The integer values that are wanted for the constants can be chosen. If a value is assigned to one constant but not to the following constants, the following constants will be numbered sequentially.

```
enum acType { SAVING = 100, FIXED RECURRING };
```

The following example defines a structure with enumerated type as a member.

```
typedef struct
{
  int custId;
  char custName[20];
  enum acType at;
}savAcnt;
int main()
{
  savAcnt s1;
  strcpy(s1.custName, "Amit");
  s1.at = SAVING;
  puts(s1.name);
  printf("%d", s1.at );          // will print 100
  return 0;
}
```

If a user friendly output in the form of strings instead of numeric values is required, then `switch-case` should be used to print the strings explicitly.

Enumeration variables are particularly useful as flags to indicate various options or to identify various conditions. They also increase the clarity of the program.

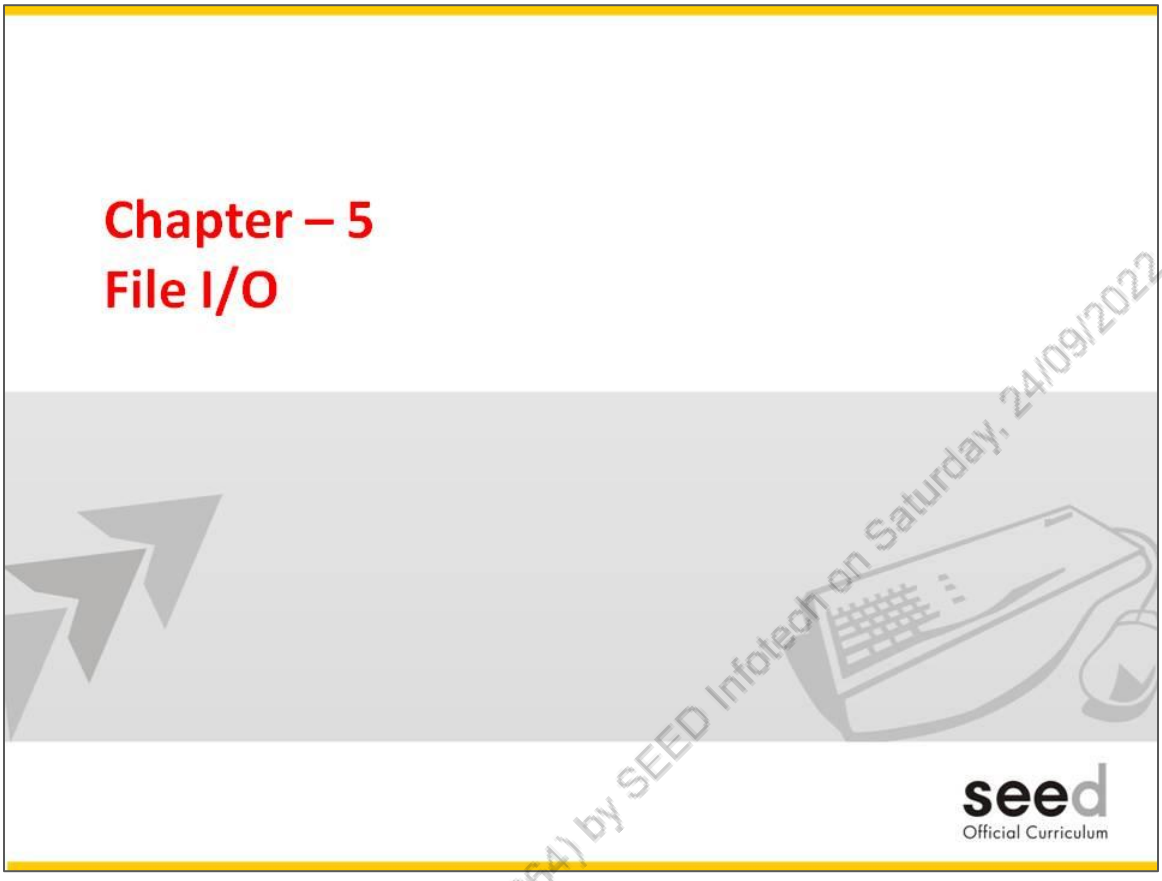
If blue color is required for the background, it is easier to understand,

```
background = blue;
```

than understanding

```
background = 1;
```

Enumerated constants are widely used in applications to increase readability and maintainability of program code.



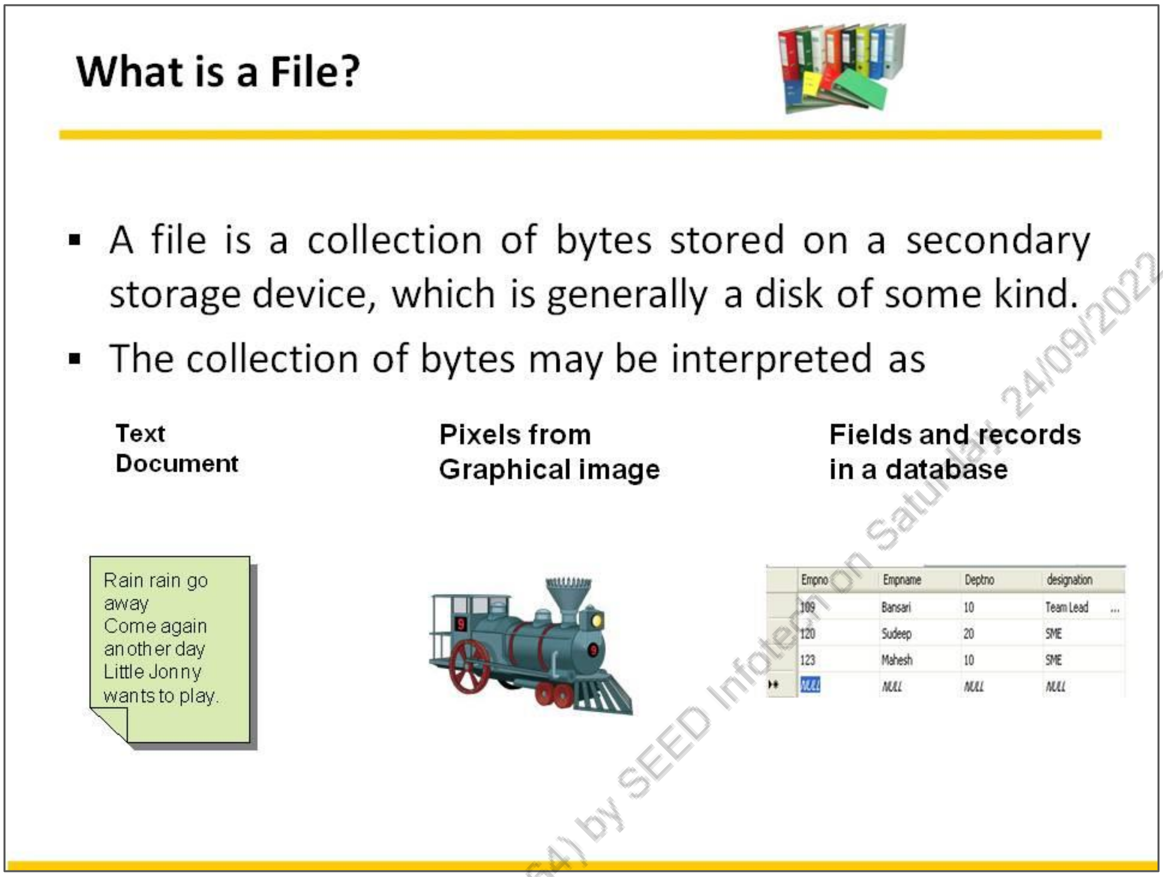
This chapter covers the concept of files and their types – text and binary files. It also covers the use of different standard library functions to perform operations like opening and closing a file, reading from a file, writing to a file, and so on.

Objectives

At the end of this chapter you will be able to:

- Define a file and state its use.
- List the different types of files.
- List different file I/O operations.
- Construct a program to create a new file or modify existing file contents using different modes of opening files.
- Construct a program to perform character and string based file read and write operations.
- Construct a program to perform formatted File I/O operations using functions like `fprintf()` and `fscanf()`.

- Differentiate between text and binary files.
- Construct a program to create a file which stores records that can be accessed randomly.



Any program processes data entered from the keyboard and stores the information in RAM. But many applications require this information for ready reference. This can be done by storing it permanently and then accessing and altering whenever required. Data is stored as files on a secondary storage media like hard disk. A file is a named physical section of storage.

Different file operations can be performed through a ‘C’ program. They are as follows:

- Creating a new file
- Opening an existing file either for reading or writing
- Closing a file

‘C’ library defines a set of functions for opening and closing files. Reading and writing operations can be character based, string based, formatted or record based.

Program: Reading a file

```
/*Displays contents of a text file on screen */
#include<stdio.h>
int main()
{
   FILE *fPtr;
   char ch;
   fPtr = fopen ("custDetails.txt", "r");
   if (fPtr == NULL) {
    printf("Cannot open file.\n");
    exit(1);
   }
   while( (ch = fgetc(fPtr)) != EOF)
     printf ( "%c", ch);
   fclose(fPtr);
   return 0;
}
```


There are different operations that are performed on a file. The file is first opened, each character is read and then on reaching `EOF`, the file is closed.

Consider a sample program to understand simple file I/O operations in text mode. Before using any file it is necessary to create the file. It can be created either by using any text editor directly or by writing a program, to write into the file. In the example mentioned above, it is assumed that the file named "`sample.txt`" is already created using a text editor. The program reads the file character by character and prints each character on the screen.

Opening a file

The process of creating a stream linked to a disk file is called opening the file. This is done using `fopen()` library function. The prototype of `fopen()` is present in stdio.h file. In order to perform any of the file I/O operations, this file must be included in the program.

```
FILE *fopen(const char *filename, const char *mode);
```

- `fopen()` returns a pointer to type `FILE`, which is a structure declared in `stdio.h`. This structure can store important information about a file - like starting buffer address, a character pointer which points to the character being read, etc. A 'C' program can communicate with a file using this `FILE` pointer. Before processing any file, it must be opened. Each file that is opened has its own file structure.
  So a pointer to this structure is declared in a program that establishes a buffer area where the file is opened.

```
FILE *fptr;
```

- `fopen()` opens the file as indicated by the filename passed as an argument, in the specified mode. It searches the file, if it is present, then it is loaded in the memory and the file pointer is set to the first character in the file. If `fopen()` fails due to any one of the following reasons, it returns `NULL`.
  a. Using an invalid filename.
  b. Trying to open a file on a disk that isn't ready (for example, the drive door is not closed or the disk is not formatted).
  c. Trying to open a file in a nonexistent directory or on a nonexistent disk drive.
  d. Trying to open a nonexistent file in mode `r`.

  A file can be opened in one of the following modes:

File - type	Meaning
"r"	Open an existing file for reading only.
"w"	Open a new file for writing only. If a file with the specified *file-name* currently exists, it will be destroyed and a new file will be created in its place.
"a"	Open an existing file for appending (i.e. for adding new information at the end of the file.) A new file will be created if the file with the specified *file-name* does not exist.
"r+"	Open an existing file for both reading and writing.

"w+"	Open a new file for both reading and writing. If a file with the specified file-name currently exists, it will be destroyed and a new file will be created in its place.
"a+"	Open an existing file for both reading and appending. A new file will be created if the file with the specified file-name does not exist.

- `fgetc()` reads a character from the stream and returns the character. On reaching end of file or any error, it returns `NULL`.

```
int fgetc(FILE * ptvar);
```

- `fputc()` writes a character to the file and also returns the same character. A return value of `EOF`, which is -1, indicates an error.

```
int fputc(int ch, FILE *fptr);
```



```
//example – code snippet to copy contents of one file
to another
 FILE *fSource, *fTarget;
 char ch;
 fSource = fopen("custDetails.txt", "r");
 if (fSource == NULL){ . . .}
 fTarget = fopen("copyCustDetails.txt", "w");
 if (fTarget == NULL){  . . . }
 while((ch=fgetc(fSource) != EOF)
   fputc(ch, ft);
 fclose(fSource);
 fclose(fTarget);
```

- The file should be closed explicitly when the processing is over. If the file is not closed explicitly, the compiler automatically closes it at the end of the program execution. On closing a file, the links between the file and the associated buffers are freed so that they can be used by other files.

The syntax is as follows:

```
fclose(fptr);
```

Note: It is seen that once the file is opened, the entire file processing and file closing functions use the file pointer and not the name of the file. Also the `FILE` pointer need not be explicitly incremented to the next character. It is performed by the `fgetc()` function.

Caution: There is often confusion between the special character that is placed to mark the end of file and the macro `EOF` defined in the stdio.h file as follows:

```
#define EOF (-1)
```

A special character that is denoted by Ctrl+Z is used to mark the end of input when accepting input from the keyboard, and thereby the file. This character has the ASCII value 26.

Understand the different modes for opening a file.

Understand the difference between a text file and a binary file and how they are treated by a C program.

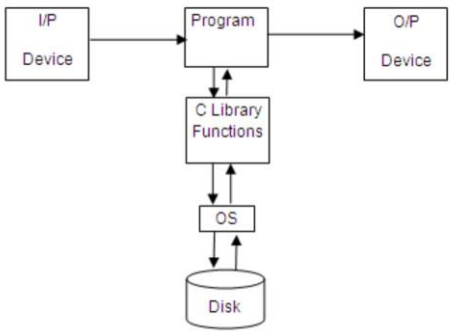
Remember to close a file once the operations are done, otherwise there will be several open file pointers which will lead to crashing of the program.

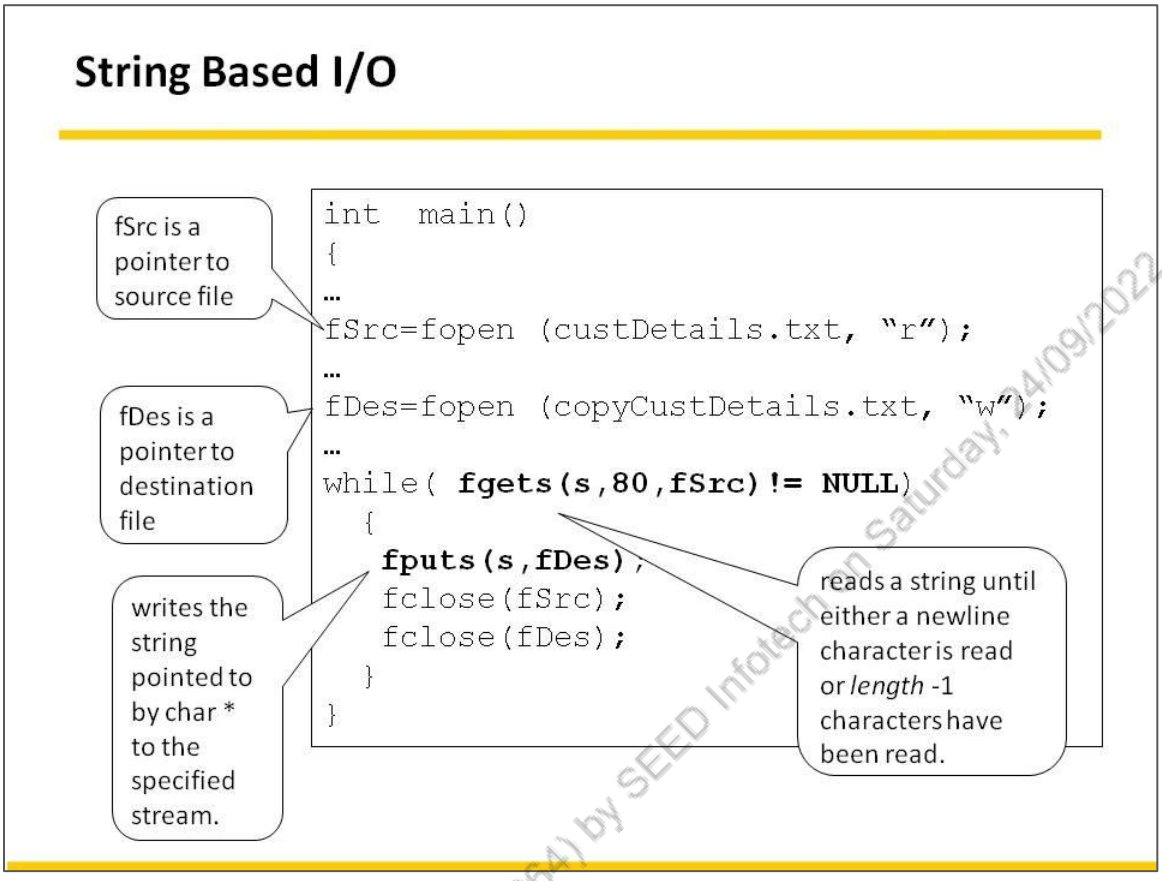
File I/O - Reading and Writing data

Type of I/O	Reading Data	Writing Data	Example
Character-based	`fgetc()`	`fputc()`	Character based I/O
String-based	`fgets()`	`fputs()`	String base I/O
Formatted	`fscanf()`	`fprintf()`	Formatted File I/O
Record	`fread()`	`fwrite()`	Record based File I/O

Each operating system has its own facility for input and output of data to and from files and devices. The programmer writes small programs that link the C language compiler for particular operating systems I/O facility. For this, the programmer uses several I/O functions defined in the library. Some functions perform character by character I/O, some perform string based I/O, some perform formatted I/O and some record I/O. Some of these functions are listed above.

The following flow diagram clearly shows how the data is written to the disk and read from the disk through a 'C' program.





In some applications accessing files character by character is sufficient. Some files can be created and read more easily with programs that utilize string oriented library functions. 'C' provides two string functions for reading and writing strings to files.

In the example mentioned above, file copying is performed using string functions. `fgets()` is used to read the strings from file and `fputs()` is used to write to a file.

```
char* fgets(char *str, int maxlimit ,FILE *fPtr);
int fputs(const char* str, FILE* fPtr);
```

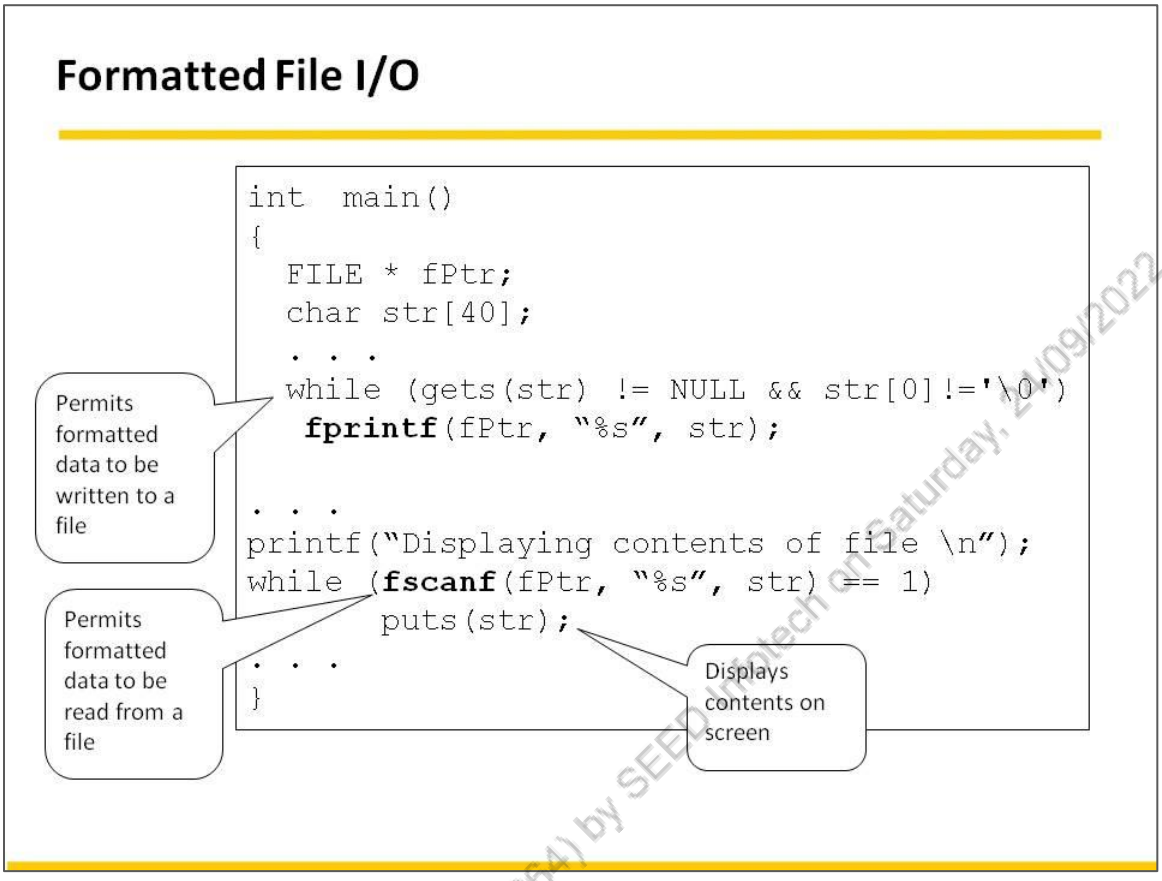
`fgets()`

It is used for reading a string from a file. The function reads a string until either a newline character is read or `maxlimit` characters have been read. `maxlimit` is the length of the string-1. On reaching the end of the file or if an error occurs, it returns `NULL`. To ensure that `fgets()` reads entire strings, stopping only at

newlines, the size of your input buffer and the corresponding value of length passed to `fgets()` must be large enough.

`fputs()`

It is used for writing a string to the file. `fputs()` does not add a newline to the end of the string. It should be included explicitly if required. The function returns a non-negative value if it is successful. On an error, `fputs` returns `EOF`.



To store floating point numbers or combination of all types of data (numeric, character, etc.), formatted file I/O functions – `fprintf()` and `fscanf()` can be used. Both these functions permit formatted data to be written to a file or read from the file. These functions are similar to console I/O functions, `printf()` and `scanf()`.

```
int fprintf (FILE *fPtr, const char *format,……);
int fscanf (FILE *fPtr, const char *format,……);
```

Both the functions take the `FILE` pointer as the first argument to identify the proper file. The file should be a text file. If first argument is given as `stdout` or `stdin` then the data is written to or read from the console instead of the file. The second argument is a pointer to the format string that specifies how `fprintf()` should write the data or `fscanf()` should read the input. The ellipses (...) indicate one or more additional arguments.

`fprintf` returns the number of bytes written. It returns a negative value when an output error occurs.

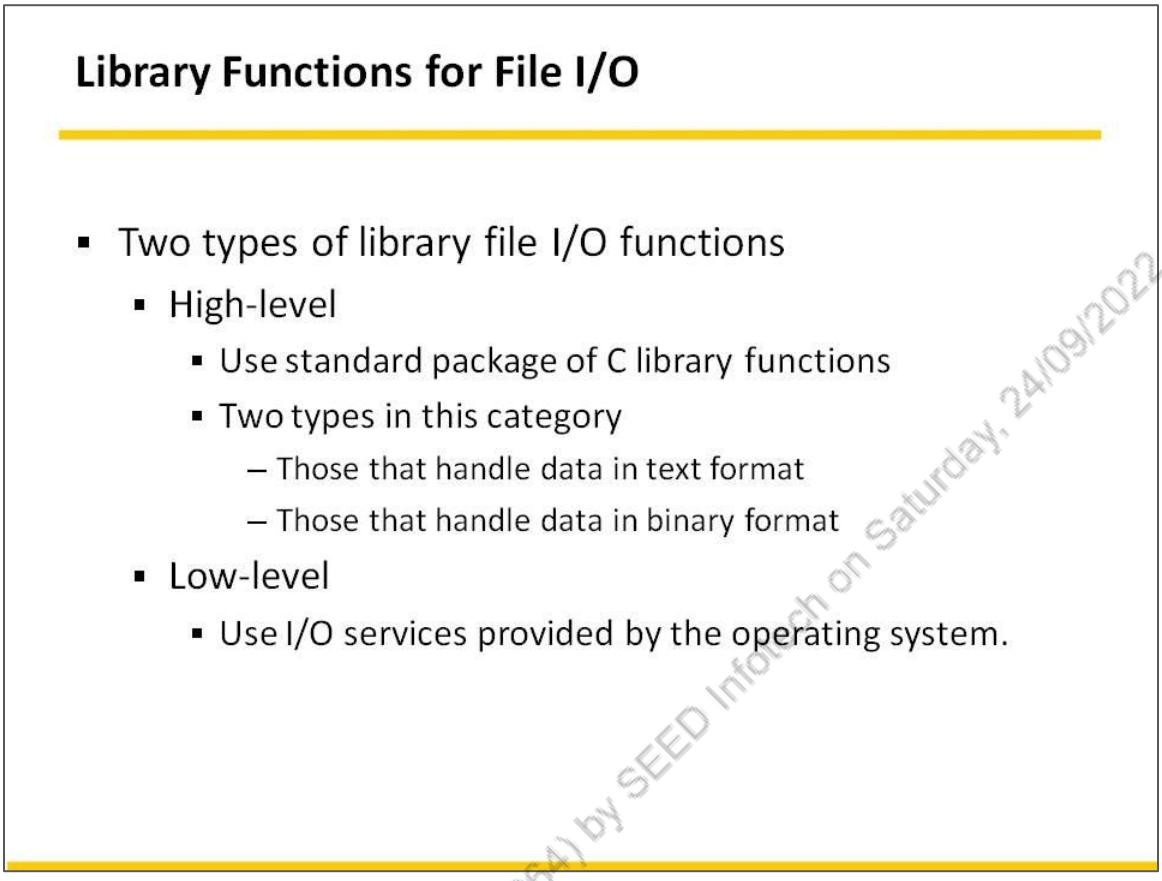
fscanf returns the number of fields successfully converted and assigned. If an error occurs or if the end of the file stream is reached before the first conversion, the return value is `EOF`.

In the example mentioned above, data is accepted from the user and written to the file using `fprintf()` function. The same data is read using `fscanf()` function at some point of the program. The format specifiers like `%d`, `%s` and `%f` are the same that are used with `printf()` and `scanf()` functions.

A more realistic approach to this problem would be to keep them in a file so that records can be processed more easily once stored in a file.

```
typedef struct  cust
{
  int custId;
  char custName[20];
}Record;

int main()
{
  FILE *fptr;
  Record cust;
   . . .
  fptr = fopen("custDetails.txt", "w");
  acceptData(&cust);
  fprintf(fptr, "%d%s", cust.custId, cust.custName);
  . . .
  fclose(fptr);
  . . .
  fptr = fopen("custDetails.txt", "r");

  fscanf(fptr, "%d%s", &cust.custId, cust.custName);
  displayData(&emp);
  . . .
}
```


Read about built-in functions such as `scanf()`, `fscanf()`, `sscanf()` and `printf()`, `fprintf()` and `sprintf()`.



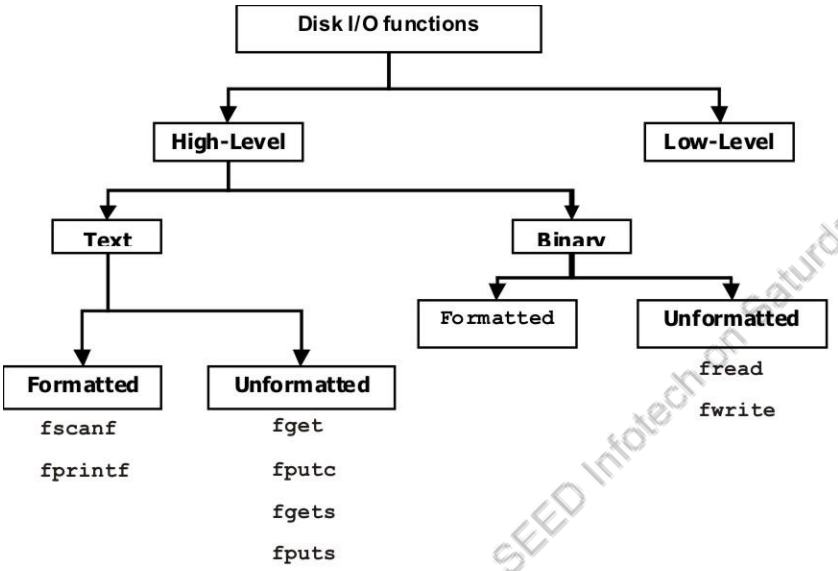
A number of library functions are used to operate on files which are broadly classified as

- High level file I/O functions use a standard package of C library functions and stdio.h header file definitions.
- Low level file I/O functions use I/O services provided by the operating system.

High level file I/O is easier to work with then low level I/O. High level file I/O functions do their own buffer management whereas in low level file I/O, buffer management has to be done by the programmer A buffer is a block of memory where the data to be stored in a file is placed temporarily. For example, if a user enters a character through the keyboard to be written to a file. This character is not directly written to the file but temporarily stored in a buffer. When this buffer is full, the disk is accessed and the contents are transferred to the file on disk. Also when end of file is reached, the buffer contents are transferred to the disk file. If

disk files are to be accessed for each character, it will take a lot of time. Thus, buffer allows information to be read from and written to data files rapidly.

High level file I/O functions are further categorized as text and binary. Both are further sub-categorized into formatted and unformatted.



Text and Binary files

There are three major differences between text and binary files.

1. Handling newline character
2. Indicating End of File
3. Storage of numbers

Handling newline character

'C' considers new line as a single character `'\n'`. On the other hand, DOS converts it into two characters linefeed (`\r`) and new line (`\n`). Thus when a file is opened in the text mode and a 'C' program writes a 'C' text file to disk, DOS converts all new line into carriage return-linefeed combination. Also, when a 'C' program reads in a text file, the carriage return linefeed combination is converted to a single new line character. When the number of characters in a file are counted, then the count will be less than the number of characters (i.e. file size) counted by DOS.

This conversion does not take place when the file is opened in binary mode. The file size is the same as the DOS operating system.

Indicating End of File

Earlier, one of the ways to indicate end of a text file was to place a special character to mark the end of file. Operating systems such as MS-DOS used this method by placing a special character Ctrl+Z whose ASCII value is 26. Today's operating systems may use an embedded Ctrl+Z character to mark the end of a file. A file can be read till this mark.

But in case of binary mode, there is no special character to mark the end of a file. The operating system's file directory is used to get a number of characters present in the file and this count is used.

A file written in binary mode should be read in binary mode while a file written in text mode should be read in text mode. Consider an example where a file written in binary mode stores a number 26 (Hex 1A) in it. If text mode is used to read the file, the file will be terminated when it encounters the special character (Ctrl+z) before the actual end of file. Thus it is seen that binary and text modes are not compatible.

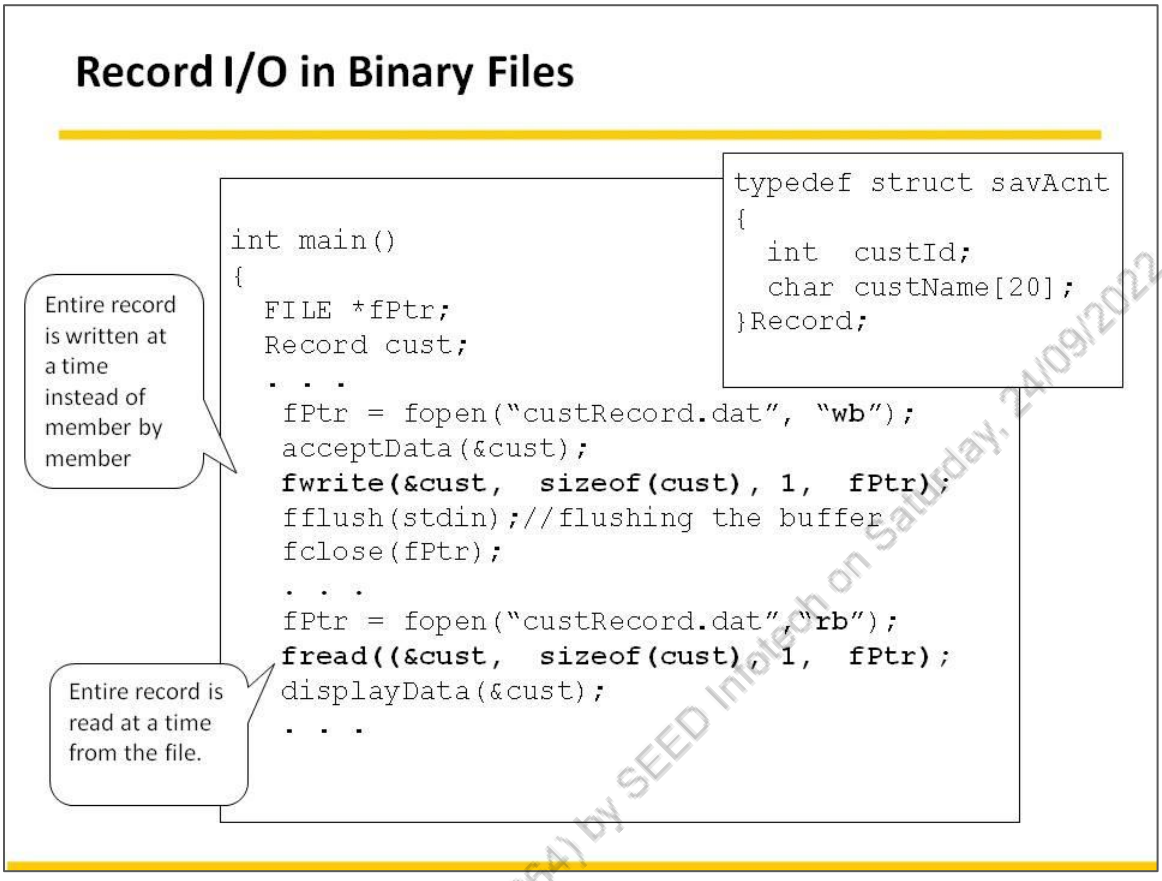
Storage of Numbers

A character occupies 1 byte. An integer occupies 2 bytes. A float occupies 4 bytes and so on. When numbers are written using text mode, each digit is stored as a character.

For example, if a floating point number 1234.56 is stored in memory it occupies 4 bytes but when written to file in text mode, it occupies 7 bytes.

Binary mode stores numbers in binary format. Binary format means each float will occupy 4 bytes on disk, thus saving memory.

What is the difference between text and binary files?



Direct file I/O is used most often when data that has been stored has to be read later by the same or a different C program. Direct I/O is used only with binary-mode. With direct output, blocks of data are written from memory to disk. Direct input reverses the process. A block of data is read from a disk file into memory. For example, a single direct-output function call can write an entire array of type double to disk, and a single direct-input function call can read the entire array from disk back into memory. Also when large numeric data needs to be handled, binary mode should be used. The direct I/O functions are `fread()` and `fwrite()`.

`fread()` and `fwrite()` functions are used to perform read and write operations. The whole record is written or read at a time instead of member by member.

```
size_t fwrite (const void *ptr, size_t size ,size_t n,
FILE *fPtr);
```

```
size_t fread (const void *ptr, size_t size ,size_t n,
FILE *fPtr);
```

- First argument is the storage location for data.
- Second argument is item size in bytes (`size_t` is defined as `unsigned int` in standard library).
- Third argument '`n`' is items of size bytes each
- Fourth argument is the `FILE` pointer
- Returns number of items actually read or written.

In the example mentioned on the slide, an entire structure variable `cust` is written to the disk using `fwrite()` function. The file is opened in `wb` mode where '`b`' stands for binary mode. Only '`b`' is appended to each of the modes described earlier if file I/O is in binary mode. The `sizeof` operator is used to provide the size of the structure variable. Many records can be written to the disk using a simple `while` loop in the above example.

The records are read back into a structure variable `sAcnt` using `fread()` function. The file is opened in read mode.

Random Access

- Required when the records are to be processed in random order.
- Operations that can be performed are
  - Adding a record anywhere in file
  - Deleting the record
  - Modifying the record
  - Rearranging the records , and so on.
- Functions used are `fseek()`, `ftell()`.

All the examples mentioned till now performed file I/O operations in a sequential manner. In a real time scenario, there could be thousands of records. If one of the records needs to be modified or deleted or an entirely new record needs to be added, then sequential file handling would be time consuming. Instead the operations can be performed by accessing the file at random.

Sequential Vs Random Access

Every open file has a file position indicator associated with it. The position indicator specifies where read and write operations take place in the file. The position is always given in terms of bytes from the beginning of the file. When a new file is opened, the position indicator is always at the beginning of the file, position 0. If an existing file is opened in append mode, the position indicator is at the end of the file.

The sequential file access functions make use of this position indicator. The reading and writing operations occur at the location of the position indicator and update the position indicator as well. The next read-write operation starts at the

updated position. Since the functions move this indicator automatically, programmer need not be concerned about it.

To get more control on different operations on a file, C library defines some functions which let the programmer determine and change the file position indicator. By controlling the position indicator, random file access is performed. Here, random means that the data can be read from or written to any position in a file without reading or writing all the preceding data.

Random file access functions are mentioned below.

rewind()

To set the position indicator to the beginning of the file `rewind()` function is used. Its prototype is present in stdio.h.

```
void rewind(FILE *fPtr);
```

- The argument `fPtr` is the `FILE` pointer associated with the stream.
- If some data is already read from a file and you want to start reading from the beginning of the file without closing and reopening it in a specific mode, use `rewind()` function.

ftell()

To determine the value of a file's position indicator, `ftell()` function is used. Its prototype is present in stdio.h.

```
long ftell(FILE *fPtr);
```

The argument `fPtr` is the `FILE` pointer returned by `fopen()` when the file is opened. The function `ftell()` returns a type `long` that gives the current file position in bytes from the start of the file (the first byte is at position 0). If an error occurs, `ftell()` returns -1L (a type `long` -1).

fseek()

More precise control over a stream's position indicator is possible with the `fseek()` library function. By using `fseek()`, the position indicator can be set anywhere in the file. The function prototype is present in stdio.h.

```
int fseek(FILE *fPtr, long offset, int origin);
```

The argument `fPtr` is the `FILE` pointer associated with the file. The distance that the position indicator is to be moved is given by offset in bytes. The argument origin specifies the relative starting point. There can be three values for origin, with symbolic constants defined in stdio.h, as shown in table given below:

Constant	Value	Description
`SEEK_SET`	0	Moves the indicator offset bytes from the beginning of the file.
`SEEK_CUR`	1	Moves the indicator offset bytes from its current position.
`SEEK_END`	2	Moves the indicator offset bytes from the end of the file.

The function `fseek()` returns 0 if the indicator is successfully moved or nonzero if an error occurs. The example given below uses `fseek()` for random file access.

The following code snippet demonstrates the use of `ftell()` and `fseek()` functions.

```
int calTotalRecords();
int main()
{
 . . .
  FILE *fPtr;
  Record sAcnt;
  int total=0;
  long offPos;
  int rec;
// code to accept the records goes here//
 . . .
total = calTotalRecords();
printf("Enter the number of record to be searched");
scanf("%d", &rec);
offPos = sizeof(Record)*rec;
if(rec > 0 && rec < total)
```

```
{
    fseek(fPtr, offPos, SEEK_SET);
    fread(&s, sizeof(rec),1,fPtr);
    printf("%d %s", sAcnt.custId, sAcnt.custName);
. . .
}
int calTotalRecords()
{
  FILE *fPtr;
  fPtr = fopen("savAcnt.dat", "rb");
  fseek(fPtr, 0L, SEEK_END);
  totRec = ftell(fPtr) / sizeof(Record);
  fclose(fPtr)
  return totRec;
}
```


Some other important functions commonly required in file I/O operations are explained below.

feof()

With a binary-mode stream, end-of-file cannot be detected by looking for -1. Instead, the macro `feof()` can be used.

```
int feof(FILE *fPtr);
```

The argument `fPtr` is the `FILE` pointer. The function `feof()` returns 0 if the end of file has not been reached or a nonzero value if end-of-file has been reached. If a call to `feof()` detects end-of-file, no further read operations are permitted until a call to `rewind()` or `fseek()` is called or the file is closed and reopened. The function can be used as shown in the following code snippet:

```
while ( !feof(fPtr) )
{
     fgets(buf, BUFFSIZE, fPtr);
     printf("%s", buf);
}
```

remove()

`remove()` function is used to delete the file. Its prototype is present in stdio.h, as follows:

```
int remove( const char *filename );
```

- The argument is a pointer to the name of the file to be deleted.
- The specified file must not be open.
- If the file exists, it is deleted (just as the `DEL` command in DOS or the `rm` command in UNIX). It returns 0 on success. If the file does not exist, is read-only or if you do not have sufficient access rights, or if some other error occurs, `remove()` returns -1.

```
if ( remove(filename) == 0)
  printf("The file %s has been deleted.\n",
filename);
else
  fprintf(stderr, "Error deleting the file %s.\n",
filename);
```

rename()

The `rename()` function is used to change the name of an existing disk file. The function prototype is present in stdio.h.

```
int rename( const char *oldname, const char *newname );
```

- The `oldname` and `newname` are the file names of the old and the new file respectively
- The only restriction is that both names must refer to the same disk drive
- The function `rename()` returns 0 on success, or -1 if an error occurs. Errors can be caused if file by name `'oldname'` does not exist or the file by name `'newname'` already exists or file from different disk drive is renamed.

```
if ( rename( oldname, newname ) == 0 )
   printf("%s has been renamed %s.\n", oldname,
newname);
else
   fprintf(stderr, "An error has occurred renaming
%s.\n", oldname);
```

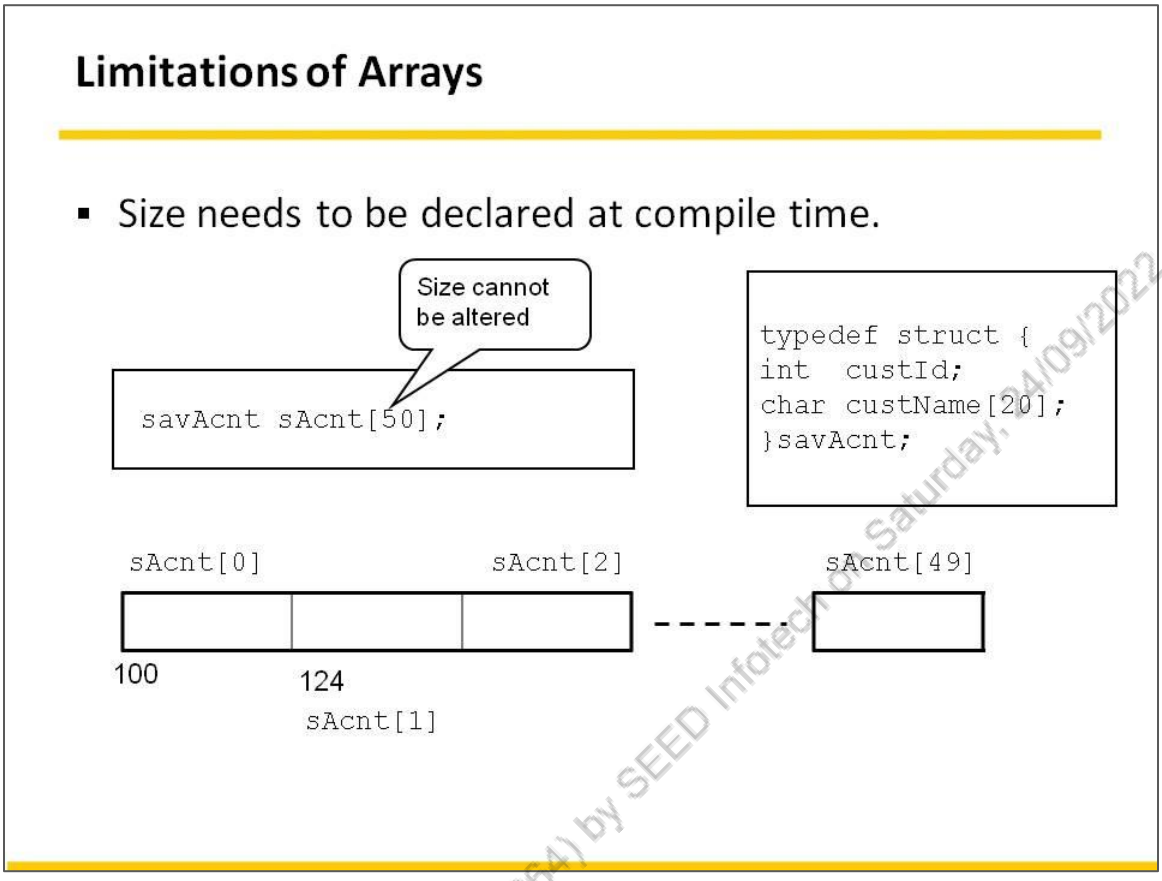
Write a function to search a record of a customer based on the customer ID entered by the user.

Though there are different types of lists, this chapter covers only singly linked list, its representation and different operations that can be performed on the linked list like creation, insertion, updation and deletion.

Objectives

At the end of this chapter you will be able to:

- State the limitations of an array and understand the need of Linked List as a data structure.
- Distinguish between representation of an array and linked list.
- List different types of Linked Lists.
- Declare a self-referential structure to represent the Linked List.
- Implement some of the operations on singly linked list like creating a node, appending a node, displaying all the nodes and deleting a node.
- Construct a menu driven program implementing singly linked list operations.



Arrays are a collection of data and are used to store a set of homogenous elements. There are certain limitations of an array. They are:

- Size

  While declaring an array, its size has to be specified, which is a constraint. To overcome this drawback, dynamic memory allocation (`malloc()`) is used where the memory is allocated at runtime depending on the number of elements required. But this too allocates a block of memory contiguously and if a single contiguous block is not available, `malloc()` fails.

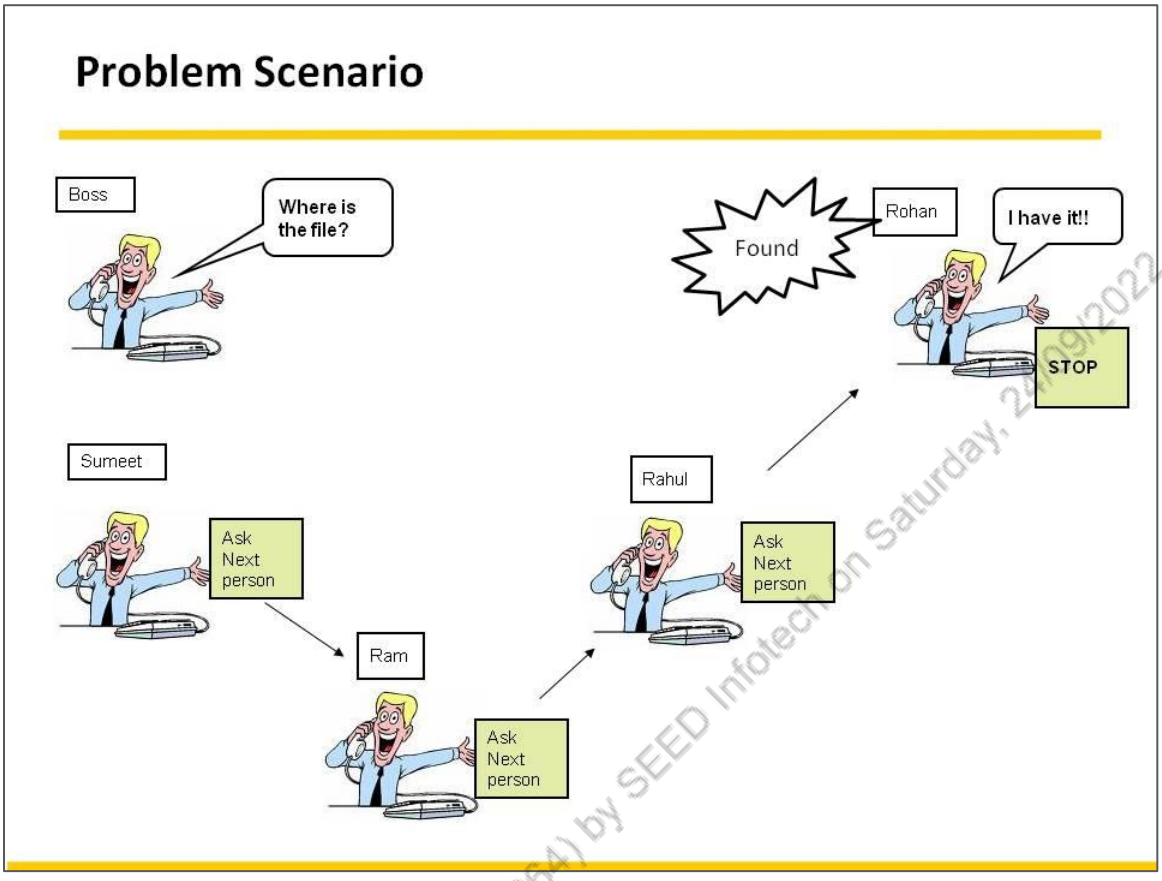
- Operations on the array

  Different operations like inserting an element into an array or deleting an element from an array is time consuming since all the elements before or after the specific position need to be shifted.

Both these limitations of arrays are overcome in linked list by using self referential structure and dynamic memory allocation in its implementation. Memory is also saved since only as much memory is allocated as required.

Consider an example wherein information of a number of employees needs to be stored.

4. An array of structures can be created. This method will have a restriction on the number of employees, because the size of the array needs to be fixed at compile time.
5. Memory can be allocated at runtime using functions like `malloc()` or `calloc()`. This too is a contiguous block of memory. If in the program, the size needs to be changed, memory has to be reallocated using `realloc()` function which would involve a lot of overheads.



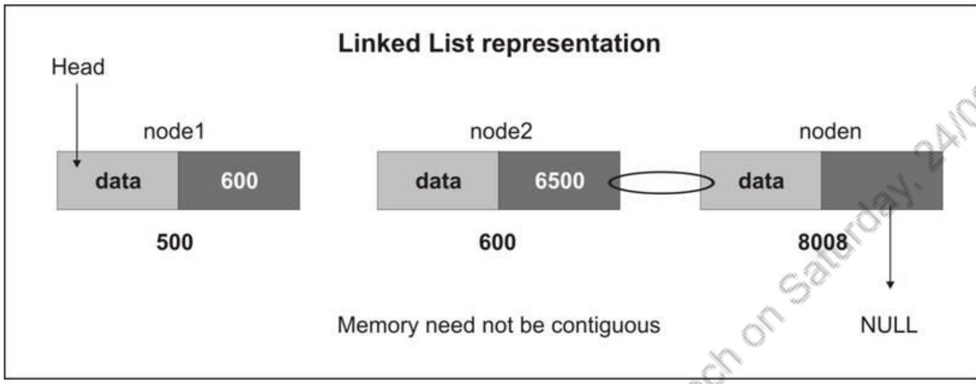
Imagine a real life situation where the Boss is looking for some parcel. He has to ask a few people before finally someone tells him, where the parcel can be found. When the file is found the search stops.

When information is stored in the memory in scattered locations, a similar situation arises. How can it be found? A record of all the locations needs to be kept. How can this be done?

A better solution for this problem is using a self referential structure. Each structure includes a pointer to the next structure so that each time a new structure is created; its address is stored in the preceding structure.

One of the members of a self referential structure is a pointer to the parent structure type.

Representation of Linked List



Refer to the diagram in the slide above. `node1`, `node2` and `node3` represent records in a linked list. Memory is allocated dynamically for each node and therefore is not contiguous. The address of `node2` is stored in `next` pointer of `node1`. Thus, `node1` points to the location of `node2`. `next` pointer of `node2` points to the location of `node3` and since there are no more nodes in the list, `next` pointer of `node3`, points to `NULL`.

To access a node in the list, one has to travel from `head` node to the target node. This is called traversing a list. Traversing is possible because each node contains a pointer pointing to the node of the same type.

A linked list consists of nodes. Some information about a linked list which should be remembered is as follows:

- Each node has two parts. An information or data field which holds the actual element in the list and the next address field which stores the address of the next node in the list. It is therefore also called a pointer or a link.

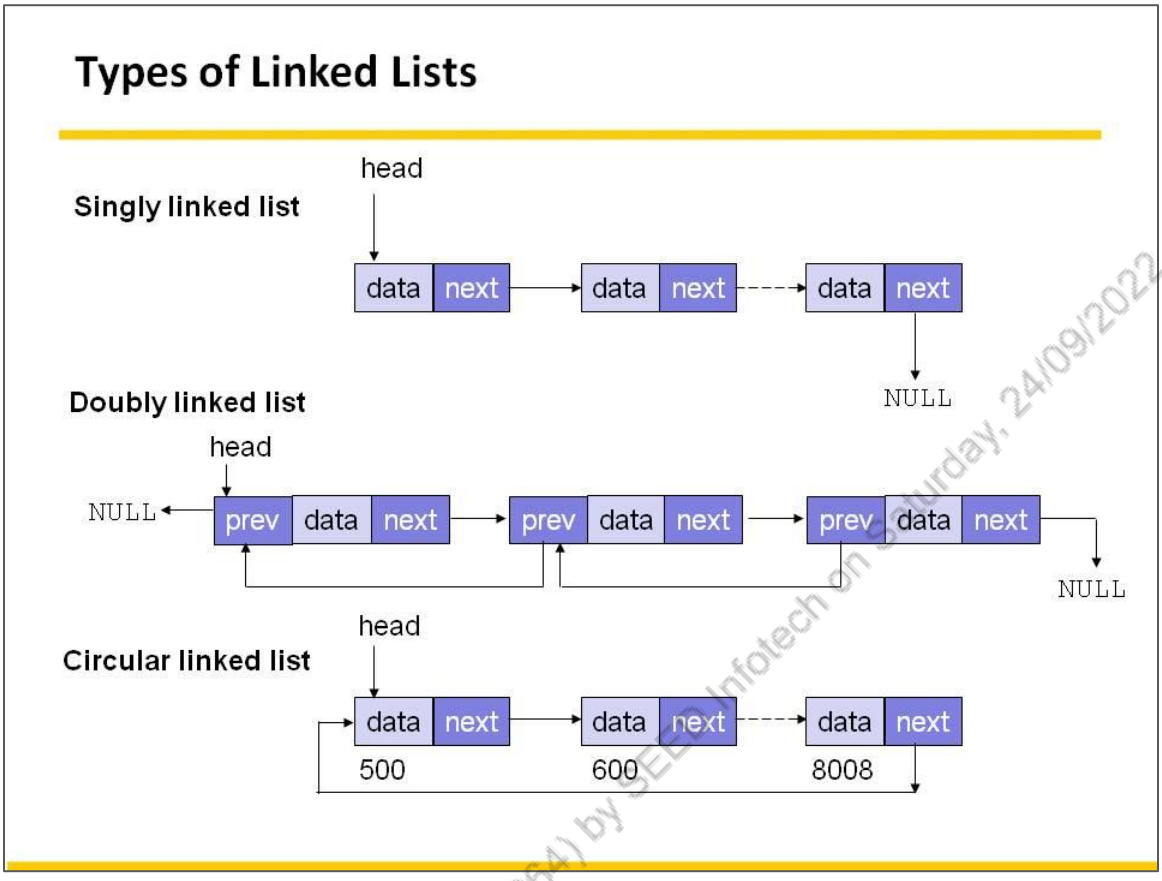
- Since memory is allocated to nodes individually when they are created, they may not be stored in adjacent memory locations. So although, one node points to the next node in the list, they may not be in contiguous memory locations. Thus, elements of a list are logically adjacent but may not be physically adjacent.
- Since one node of a list provides the link to the next node, it is called linked allocation.
- To indicate the end of list, the next part of the last node always contains a special value known as `NULL`.
- Although each node points to the next node how can one find the first node of the list? For this purpose, a special external pointer called `head` is used to point to the first node.
- A list which has all the above features is called a singly linked List.

Advantages of a linked list

- Grows dynamically.
- Unlike an array, memory is allocated as and when required. Hence nodes can be added to a linked list even at run time.
- Insertion and deletion operations are fast.
- In an array inserting or deleting a single element involves a lot of data movement. Whereas, in a linked list, it is comparatively faster because only the address of the next pointer needs to be changed.

Read about the applications of linked list.

You should know about the applications of linked list.



A linked list is a list in which each node contains information describing where to find the next node. Every node contains two parts – a data part and a link part which links to the next node.

There are several different kinds of linked data structures.

- Linear/Singly linked list

  Nodes are linked together in a sequential manner. One external pointer points to the first node in the list and the pointer of the last node points to `NULL`.

- Doubly linked list

  Multiple pointers permit forward and backward traversal within the list. One pointer points to the previous node in the list and one to the next node in the list.

- Circular linked list

  A circular linked list has no beginning and no end. The pointer in the last node points to the first node in the list making it circular.

Note: This chapter covers only singly linked list. Other types are beyond the scope of this course.

Self Referential Structure

- One of the members of the structure is a pointer to the parent structure type.
- General form

```
struct tag
{
  member 1;
  member 1;
  . . .
  struct tag *next;
};
```


A better representation for the linked list is – a self referential structure. Each structure should include a pointer to the next structure so that each time a new structure is created, its address is stored in the preceding structure.

One of the members of a self referential structure is a pointer to the parent structure type.

General form has been shown above. The structure of type `tag` contains a member, `next`, which is a pointer to structure of type `tag`. Thus `tag` is a self referential structure.

List as a data structure

Self referential data structures are useful in applications that involve linked data structures such as lists, trees and graphs. One structure will represent one component of a list. Each component of the list is called a node. The self-referential pointer indicates where the next node is located. Thus, it links one node of the list with another. Consider the following self referential structure

```
typedef struct LL       //declaring the structure for
linked list
{
    int custId;
    char custName[20];
    struct LL *next;

}node;
```

`struct node` identifies a structure consisting of two members, an integer and a pointer (`next`) to a structure of same type. '`head`' is an external pointer to `struct node`. It stores the address of the first node in the list. This is shown below.

Pointer to next node is assigned a value of `NULL` until next node is created. All operations on linked list, i.e. changing the order of nodes, adding nodes, deleting nodes etc., can be done easily since only the pointer value has to be changed.

Singly Linked List

The address of the first node is stored in a separate external pointer referred to as the `head` pointer. The `next` member pointer of the last node points to `NULL`.

- Need of head as an external pointer

  In a linear linked list, there is a starting node and an ending node. The address of every node except the first one is stored in a node. That is why an external pointer (`head`) is maintained to store the address of the first node of a list.

- Need of initializing pointer '`next`' to `NULL`

  Each node points to the next node in the list. But the last node has no node to point at. To mark the end of list, the '`next`' pointer in the last node is set to `NULL`.

Whatever be the application that uses linked list, the following are the basic operations:

- Creating a node

  Creating a node is the basic operation as initially the list would be empty. It is done by using the `malloc()` function.

- Displaying a list

  All the nodes in the list are displayed by traversing the entire list.

- Inserting a node

  A node can be added to a list anywhere – either at the beginning, at the end or anywhere in between. The logic is the same as used for creating a new node.

- Deleting a node

  A node can be deleted from a list whether it is at the beginning, or end or anywhere in between. The memory occupied by the deleted node should be freed.

- Reversing a list

  Reversing a list means changing the order of the nodes – the first node becomes the last and the last one becomes the first.

- Sorting a list

  When the data in all the nodes is not in any particular required order, it can be arranged by sorting the list in either an ascending or descending order.

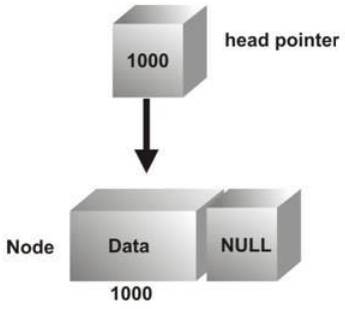
Note: Some of the functions are explained in the chapter.

Creating a Node

- Create the first (head) node by allocating memory using `malloc` function and storing data in it.

```
head = (node*)malloc(sizeof(node));
head→next = NULL;
```

- Make an external pointer point to the head node. Call the pointer 'head'

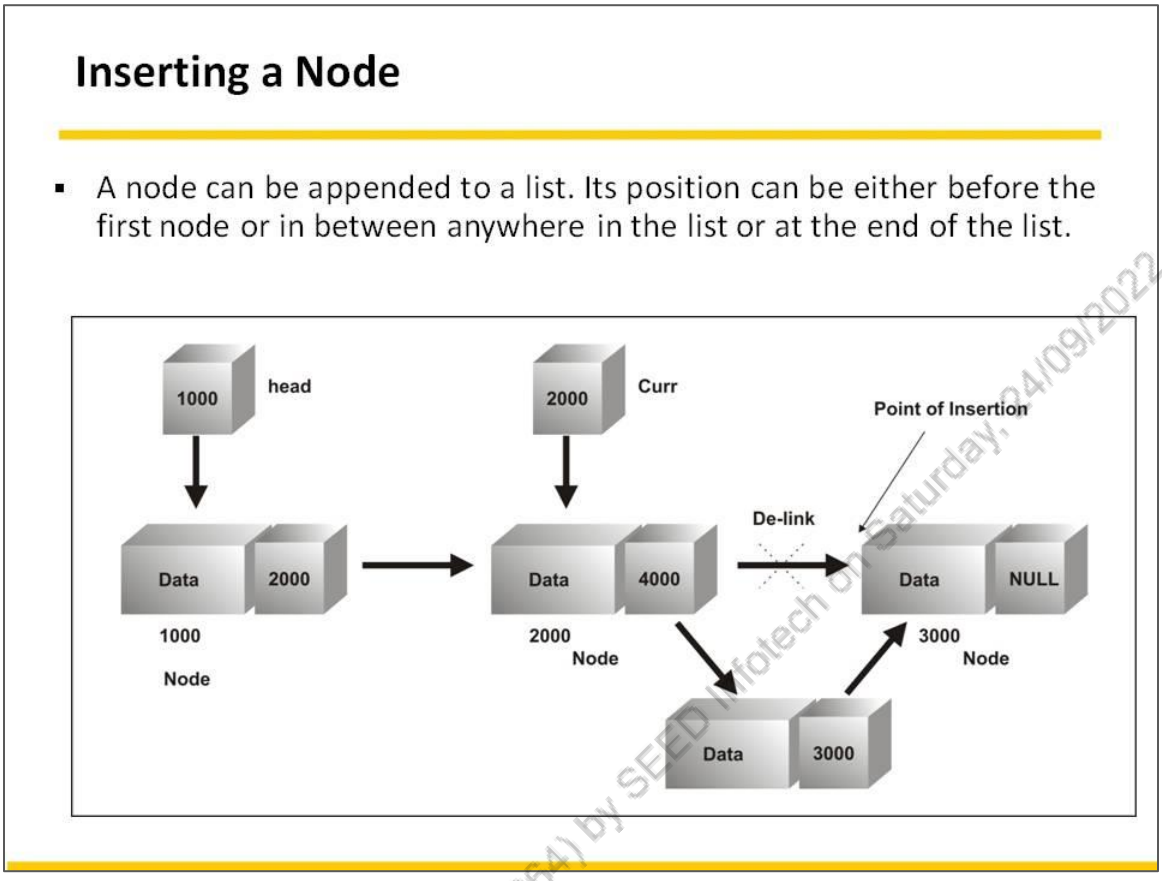


Creating a node is the basic operation as initially the list would be empty.

```
// create new node
node* createNode()
{
    node *newNode;
    int num;
    char nm[20];
    printf("\n\nCustomer ID:");
    scanf("%d",&num);
    fflush(stdin);
    printf("\nEnter customer name: ");
    gets(nm);
    newNode=(node*)malloc(sizeof(node));    //allocate
memory to the new node
    newNode->custId=num;
    strcpy(newNode->custName, nm);
    newNode->next=NULL;
```

```
    return newNode;
}
```

In the above function definition, `newNode` is a pointer of type `Node`. Memory is allocated to `newNode` using `malloc()`. If memory is allocated, values for `custId` and `custName` are accepted, and `next` pointer is set to `NULL`.



A node can be inserted into a list before the first node. In this case, the new node becomes the first node. A node can also be inserted anywhere in between the list. The following is the algorithm used to write a function to insert a node.

It is assumed that the list is already created. To insert a node, it will have to be first created. To create a new node, `createNode()` function is called. After creating the node, it has to be inserted in the list. For this purpose, the user is given an option to either insert the node in the beginning, middle or at the end of the list.

Case 1 - Beginning

1. If the user wants to insert the node in the beginning, and there are already nodes in the list, the address of the first node is assigned to the next pointer of the new node and the new node is made the first node by making the head pointer point to it.

Case 2 - Middle

1. To insert a node in the middle, the user is asked the position where the node is to be inserted. A temporary pointer, called `curr`, is used to traverse the list. Address of the first node is assigned to `curr`.
2. If the user selects the first position after selecting this option, and there are no nodes in the list, the new node is inserted in the beginning by pointing the head pointer (`hd`) to `newNode`.
3. If there are nodes in the list and the user selects this option, the list is traversed using the `curr` pointer until the node after which the node to be inserted is located. Variable `count` is used for this purpose. If the entire list is traversed but position is not found, then an error message is displayed.

Case 3 - End

1. If the list is empty and this option is selected, the node is inserted as the first node by pointing the head pointer (`hd`) to `newNode`.
2. To insert the node at the end, the entire list is traversed and the node is inserted as the last node in the list.

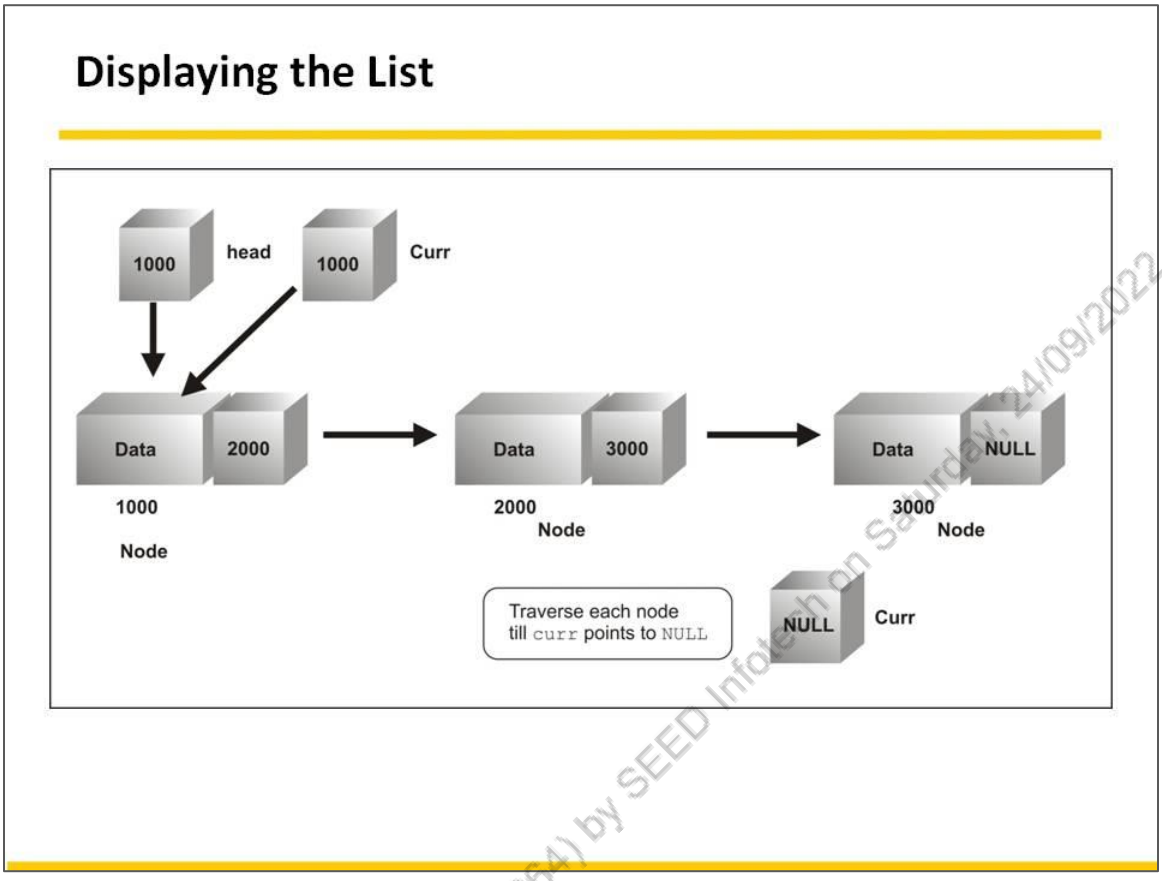
```
//add as the first node
node* addFront(node *head)
{
 node *newNode;
 newNode=createNode();            //create a node
 newNode->next=head;
 head=newNode;                   //change the head
address
}
```

```
// add after a node
node* insert(node *head, int after)
{
 node *newNode,*curr=head,*prev;
 if(curr==NULL)                  // check if link list
is empty.
 {
     printf("\n\nLinked List Empty");
     return head;
```

```
 }
 newNode=createNode();
 while(curr!= NULL)
 {
     if(curr->custId==after)
         break;
     curr=curr->next;             //traverse through
the linked list
 }
 if(curr==NULL)                 //if end of the linked
list
 {
     printf("\n\nNode not found");
     return head;
 }
 prev=curr->next;               //place the new
node
 curr->next=newNode;
 newNode->next=prev;
 return head;
}
```

```
//add as the last node
node* append(node *head)
{
 node *newNode,*curr=head,*prev;
 newNode=createNode();
 if(head==NULL)
     head=newNode;              //if head is null will
become first node
 else
 {
     while(curr!= NULL)
     {
         prev=curr;
         curr=curr->next;       //traverse through
the linked list
```

```
      }
      prev->next=newNode;
 }
 return head;
}
return head;
}
```



A list can be displayed by traversing the list, node by node, from the beginning till `NULL` is encountered. A temporary pointer will be required to point to one node at a time. As data in node is displayed, this temporary (`curr`) pointer is moved to the next node in the list. The figure in the above slide demonstrates this.

To display a list, it is assumed to be already created. First, the temporary pointer (`curr`) is pointed to the first node (`head`) pointer. The data of the head node is displayed. The `curr` pointer is then moved to the next node: `curr = curr→next`.

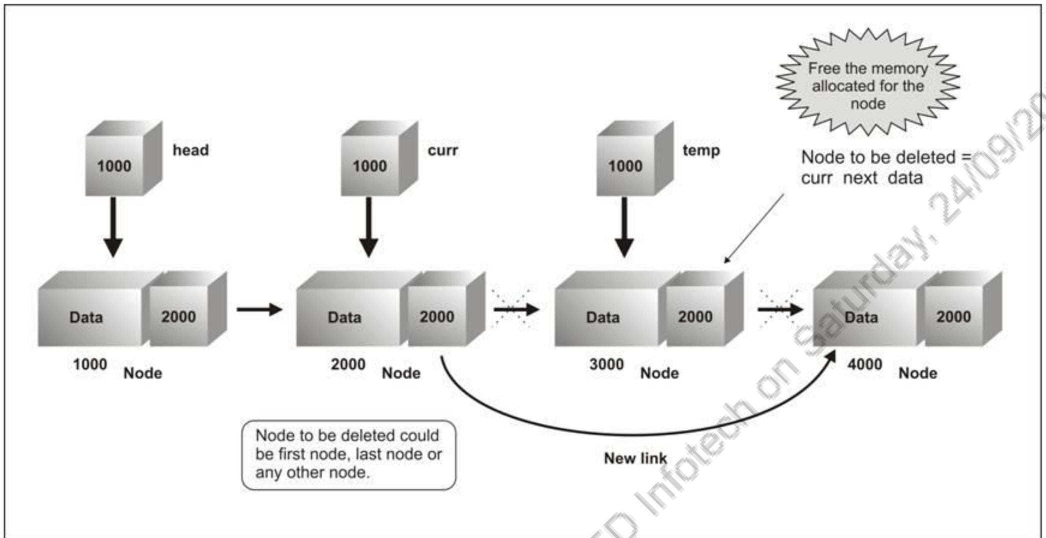
The above steps are repeated until (`curr!=NULL`). The display of the list stops when the condition (`curr==NULL`) is reached.

The function to display a list is as follows:

```
void display(node *head)
{
    node *curr=head;
```

```
    if(curr==NULL)
        printf("\n\nList is Empty\n");
    else
    {
        printf("\nLinked List :");
        while(curr != NULL)
        {
            printf("%5d%10s->", curr->custId, curr-
>custName );
            curr=curr->next;
        }
        printf("->NULL\n");
    }
}
```

Deleting a Node



When deleting a node, it is required to know the node after which the node is to be deleted. This is similar to the method of inserting a node.

It is assumed that the list is already created. A temporary pointer (`temp`) will have to be maintained to point to a node after which a node has to be deleted. The `temp` pointer is moved to one node before the node to be deleted. The node to be deleted may be the first node or a node anywhere in between the list. Following is the algorithm used to write a function to delete a node.

- There may be no nodes in the list and the user attempts to delete a node. An error message is displayed in this case.
- The `custId` which is unique is used to identify the node to be deleted. So the user is asked to input the `custId`.
- The node to be deleted may be the first node in the list. A `temp` pointer is used to store the `next` pointer and the memory is freed.
- The node to be deleted could be anywhere else in the list. The entire list is traversed up to the second last node to locate the `custId`. The `curr`

pointer is used to do so. When the `custId` matches, the node is deleted and memory is freed.

- If `custId` is not found in the list, a message is displayed to indicate that the `custId` was not found.

Complex applications like databases, store data in the form of advanced data structures which are based on linked list.

Following is the function to search and delete a node.

```
//search a node
node* search(node *head, int data)
{
    node *curr=head, *prev=NULL;

    while(curr != NULL)
    {
        if(curr->custId==data)
            break;
        curr=curr->next;        //traverse through the
linked list
    }
    if(curr==NULL)          //if end of the linked list
    {
        printf("\n\nNode not found");
        return head;
    }
    return curr;
}
```

```
node* deleteNode(node *head,int ele)
{
    node *curr=head,*prev=head;
    //search the node
    curr = search(head,ele);

    if(curr==NULL)  //if end of the linked list
    {
        printf("\n\nNode not found");
        return head;
    }
    if(curr==head)  //if delete the first node
    {
        head=curr->next;
        free(curr); //free memory
    }
    else
    {
        prev->next=curr->next;  //skip the address of
the node
        free(curr);             //free the skipped node
address
    }
    return head;
}
```


The following program is not full-fledged, but it is sufficient to understand how linked list operations can be implemented. All the above modules (functions) can be combined as follows –

```
char mainMenu();
char addMenu();

node* createNode();
node* append(node*);
node* addFront(node*);
node* insert(node*, int);
node* search(node*, int);
```

```
node* deleteNode(node *head,int ele);
node* modifyNode(node *head,int ele);
void display(node *head);
```

Other required code:

```
//display menu
char mainMenu()
{
    char choice;
    printf("\n\t\t\t:Menu:\n\t\t1:Add a
Node\n\t\t2:Delete a Node\
        \n\t\t3:Modify a Node\n\t\t4:Display Linked
            List\n\t\t5:Exit\n\t\tChoice:");
    fflush(stdin);
    choice=getche();
    return choice;
}
char addMenu()
{
    char ch;
    printf("\n1:Add at the start\n2:Add at end\n3:Add
after\nChoice:");
    fflush(stdin);
    ch=getche();
    return ch;
}
node* modifyNode(node *head,int ele)
{
    node *curr=head;
    char nm[20];
        //search the node
    curr = search(head,ele);

    if(curr==NULL)  //if end of linked list
    {
        printf("\n\nNode not found");
        return head;
```

```
    }
    else
    {
        fflush(stdin);
        printf("\n\nNew name of customer: ");
        gets(nm);
        strcpy(curr->custName, nm);       //assign the
new value to the node
    }
    return head;
}
```

Code in `main()`

```
int main()
{
    node *head=NULL;
    int after,ele;
    char ch,choice,cont;
    do
    {
        choice = mainMenu();
        switch(choice)
        {
            case '1'://Add
                ch = addMenu();
                switch(ch)//switch(getche())
                {
                    case'1'://Add at start
                            head=addFront(head);
                            break;
                    case'2'://Add at end
                            head=append(head);
                            break;
                    case'3'://Add After
                        if(head==NULL)
                        {
```

```
                                printf("\n\nLinked List
Empty");
                                break;
                            }
                            printf("\n\nAdd after which
element? ");
                            scanf("%d",&after);
                            head=insert(head,after);
                            break;
                        default:printf("\n\nWrong Choice");
                    }
                    break;
                case '2'://Delete
                        if(head==NULL)
                        {
                            printf("\n\nLinked List
Empty");
                            break;
                        }
                        printf("\nEnter Id of customer to
be deleted: ");
                        scanf("%d",&ele);
                        head=deleteNode(head,ele);
                        break;
                case '3'://Modify
                        if(head==NULL)
                        {
                            printf("\n\nLinked List
Empty");
                            break;
                        }
                        printf("\nEnter ID of customer to
be modified: ");
                        scanf("%d",&ele);
                        head=modifyNode(head,ele);
                        break;
                case '4'://Display
```

```
                    display(head);
                    break;
            case '5':exit(0);
            default :printf("\nWrong Choice");
        }
        printf("\n\nDo you want to continue?(y/n):
\n");
        cont=getche();
    }while(cont=='y'|| cont=='Y');
    return 0;
}
```

Appendix A

Function Pointer

A function also has an address like an integer variable, character variable, or an array. Thus there can be a pointer to function. The idea may seem strange but it exists. So, a function can be invoked using its pointer instead of the usual way. Declaration of function pointer purely depends on the function declaration. For example,

```
. . .
void disp(int x, int y)      //function 1
{
   printf("%d\t%d", x, y);
}
void add(int x, int y)  //function 2
{
   printf("%d", x + y);
}
void (*funPtr)(int, int);  //declaration - pointer to
function
. . .
```

The above code snippet shows two function definitions and a declaration of a pointer to a function. Now, if the function is to be invoked using pointer the address of the function will be required. The name of the function by itself (without the round brackets), gives the address of function.

The code snippet given below invokes functions `sub` and `add` using the same function pointer.

```
. . .
int num1, num2;
funPtr = disp;       /* assign address of function 1*/
printf("\nAddress  of function disp is %u", funPtr);
(*funPtr)(num1, num2);       //call to function 1
funPtr = add;        /* assign address of function 2*/
printf("\nAddress  of function add is %u", funPtr);
```

```
(*funPtr)(num1, num2);       // call to function 2
. . .
```

This can be understood quickly if the concept of how to define a pointer to a variable and an array of pointers. Consider the macro defined as `#define SIZE 10`

Pointer to integer	`int *p;`
Array of pointers to integer	`int *p[SIZE];`
Pointer to array of integers	`int (*p)[SIZE];`
Pointer to function returning integer	`int(*p)( );`
Function returning pointer to integer	`int *p( );`
Pointer to a function returning nothing (void)	`void (*p)( )`

Round brackets are used around the '*' operator to give a higher precedence than the function call parenthesis. `int *p(),` defines a function returning a pointer to integer. Here function brackets take precedence.

The address of function `sub` is assigned to `funPtr` in the second statement. First, the pointer to function is defined then an address is assigned to it. This is the same procedure when a pointer to an integer is defined and then an address is assigned to it.

`(*funPtr)()` invokes the function `sub` since `funPtr` now points to `sub`.

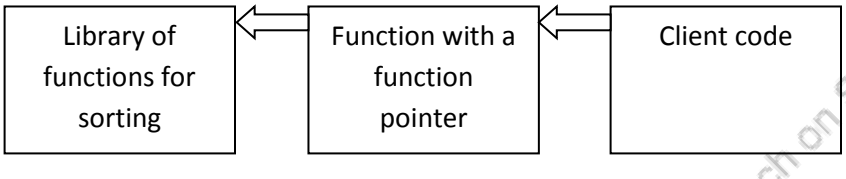
The same function pointer can be used to point to other functions, provided they have the same prototype. It is shown in the snippet by making `funPtr` point to the function `add` and then using the function pointer to invoke the function.

Function pointers are used in callback functions. Callback functions separate the calling function from the called function. The called function does not care who is calling it; all it knows is that there is a caller with a certain prototype and probably some restriction.

A classic practical application of callback functions is when writing a library that provides implementation for sorting algorithms such as bubble sort, shell sort,

quick sort, and others. The logic of sorting is same for any type of data. When a function pointer is used, there is no need of specifying the type of data; depending on the type of argument passed at the time of calling the function, the appropriate function will be called.

The advantage is that there is no need to write the sorting logic into the functions, making the library more general to use. The client will be responsible to that kind of logic. Or, it can be used for various data types (ints, floats, strings, and so on).



Bitwise Operators

Earlier, it was stated that C is a middle level language. It can be used to perform low-level operations too. This is possible because C allows manipulations of individual bits in a variable.

For example, a hardware device is often controlled by sending it a byte or two in which each bit has a particular meaning. Also, operating system information about files is often stored by using particular bits to indicate particular items. Many compression and encryption operations manipulate individual bits. Thus, C can be used for writing device drivers and embedded code.

Bitwise operators can be divided into 3 categories –

- Ones complement operator
- Logical bitwise operator and
- Shift operators

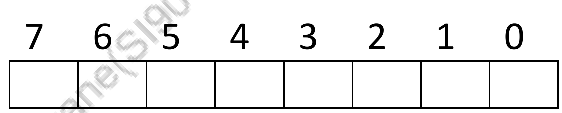
Byte has been considered as the smallest thing that can be operated upon. But each byte is further made up of bits. ‘C’ contains several operators that allow bitwise operations to be carried out easily. Direct interaction with the hardware is possible by using bitwise operations. Collection of bits is binary representation of numbers.

When bitwise operators are used, the binary representation of numbers is considered. These operations can be performed only on integers (short, long) and signed or unsigned characters

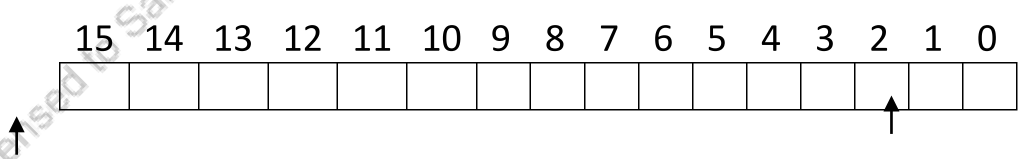
```
Decimal   Binary
   5→     00000101
   6→     1000001        bit
5225→     00010100 01101001

                           byte

                word
```

Bits are numbered from zero onwards, increasing from right to left.

Character



Integer



most significant bit   least significant bit

One’s complement operator

The one's complement operator (~), which is a unary operator, causes the bits of its operand to be inverted. Each 1 is converted to 0 and each 0 is converted to 1. The operator precedes the operand. For example,

```
~(5225) = 59286
```

$(5225)_2$ = 0001010001101001

On inversion it will result into 1110101110010110 = 59286

Number	$(5225)_2$	0001010001101001
One's complement	$\sim(5225)_2$	1110101110010110

Number	$(1)_2$	0000000000000001
One's complement	$\sim(1)_2$	1111111111111110

It can be seen from the above two examples, that one's complement of a number gives an entirely different number. Hence it can be used to encrypt a file.

Logical bitwise operators

There are three types of logical bitwise operators -

1. Bitwise AND operator (&)
2. Bitwise OR operator (|)
3. Bitwise XOR operator (^)

These operators use two operands. The operator has to be applied to each pair of bits – one from either operand, independent of other bits within the operand. The least significant bits within the two operands will be considered, and then the next least significant bit and so on, until operator has been applied to all bits. Both operands must be of the same type.

AND operator

The ANDing operation can be better understood by looking at the truth table.

Bit from operand 1	Bit from operand 2	Result
0	0	0
1	0	0
0	1	0
1	1	1

There is a difference between the logical `AND` operator (`&&`) and bitwise `AND` operator (`&`). For example, if `x = 1` and `y = 2` then, the expression (x && y) will consider x as one expression which on evaluation gives true (Since it has some value other than 0) and y as another expression which on evaluation gives true. Thus, the result is true.

Therefore (`x && y`) is 1.

For (`x & y`), binary equivalent of `x` will be ANDed with binary equivalent of `y`, bit by bit.

```
x  0000
   0001

y  0000
   0010

   0000      After ANDing (refer
   0000      truth table)
```

Therefore (x & y) is `0`.

The ANDing operation can be used to

1. Check whether particular bits are `ON` or `OFF`.
2. To turn off any bits.

In either case the second operand has to be constructed according to the bits to be tested or to be turned off. It is called mask.

- Check whether particular bits are ON or OFF

  Suppose, in the bit pattern 1010 1101 it is required to check if 3rd bit is ON or OFF. A bit pattern (mask) such that 3rd bit in the pattern is 1 will have mask as 0000 1000

  ANDing these two patterns,

  ```
  1010      Original bit
  1101      pattern

  0000      AND Mask
  1000

  0000
  1000
  ```

  It can be seen that the resulting bit pattern has the 3rd bit ON. It will be ON, when the 3rd bit of both operands is ON. A Mask has been constructed such that 3rd bit is ON. Thus 3rd bit in the original bit pattern must be ON.

- To turn off any bits.

  If it is required that 5th bit of the given bit pattern 1011 1101 be turned OFF. A bit pattern can be constructed such that its 5th bit is OFF and all other bits ON. So that whatever is the 5th bit of the given pattern, result will have 5th bit set to OFF, while all other bits remain unchanged.

  ```
  1011      Original bit pattern
  1101

  1101      AND mask
  1111

  1001      Resulting bit pattern (5th bit
  1101      turned OFF)
  ```

OR operator

The truth table for OR operation can be given as follows –

Bit from operand 1	Bit from operand 2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR operator is used to turn ON particular bits in a number. The bits to be turned ON in the original pattern should be made 1 in the OR mask. All other bits in the mask are kept 0 to maintain the remaining original bits unchanged. To turn off the 3rd bit,

```
1101        Original bit pattern
0000

0000        OR mask
0100

1101        Resulting bit pattern
0111
```

XOR operator

It is called bitwise exclusive OR operator. It excludes the last condition in the OR truth table. The bitwise exclusive OR operator sets a 1 in each bit position where its operands have different bits, and 0 where they are same.

Bit from operand one	Bit from operand two	Result
0	0	0
0	1	1

1	0	1
1	1	0

`XOR` operation can be used to toggle (invert) bits. If you want to toggle last 8 bits of a pattern,

a = 0110 1101 1011 0111 Original bit pattern

0000 0000 1111 1111 `XOR` mask

0110 1101 0100 1000 Result

When each of the rightmost 8 bits in 'a' is XORed with corresponding 1 in mask, the resulting bit is opposite of bit originally in 'a'. On the other hand, when each of the leftmost 8 bits in 'a' is XORed with corresponding 0 in mask, the resulting bit will be the same as the bit originally in 'a'.

Shift operators

There are two types of shift operators –

1. Right shift operator.
2. Left shift operator.

These operators take two operands. The first operand is the bit pattern to be shifted and the second is an unsigned integer, a number that indicates the number of places the bits are shifted.

Right shift operator

>> is the right shift operator. For example, `x >> 3` shifts all bits in x three places to the right. If x has a bit pattern `1101 0111`, then `x >> 3` will give `0001 1010`

When bits are shifted to the right, the blanks created at the left are filled with zeroes if the operand is unsigned. If the operand is *signed, blanks are filled with either zeroes or sign-bit. It depends on the machine.

If the operand is a multiple of 2, then shifting one bit to the right causes division by 2.

`64 >> 1 = (0100 0000) >> 1 = (0010 0000)` which is the binary equivalent of 32.

`64 >> 2 = (0100 0000) >> 2 = (0001 0000)` which is the binary equivalent of 16.

* Signed quantity: The leftmost bit is used to indicate sign of a signed number. It is called the sign-bit. The leftmost bit is 1 if the number is negative. The leftmost bit is 0 if the number is positive.

Left shift operator

<< is the left shift operator. For example,

`y << 5` shifts all bits in y five places to the left.

If `y` contains the bit pattern `0110 1001`, `y << 5` gives `0010 0000`. For each bit shifted to left, a zero is added to the right of the number.

If the operand is multiple of two then shifting one bit to the left causes multiplication by 2.

Bitfields

When the question of saving memory is at priority, variables can be grouped and this requires a few bits as a single word (i.e. single integer), instead of defining each variable as an integer or character. This can be done using bitfields.

1-bit can give two values (true / false). Value of 2-bits can range from 0 to 4. Value of 3-bits can range from 0 through 7. Several such data items can be combined into an individual word of memory and are called bit-fields. They are defined as members of structure. For example,

```
struct tag{
    member 1 : size;
    member 2 : size;
    .
    member m : size;
};
```

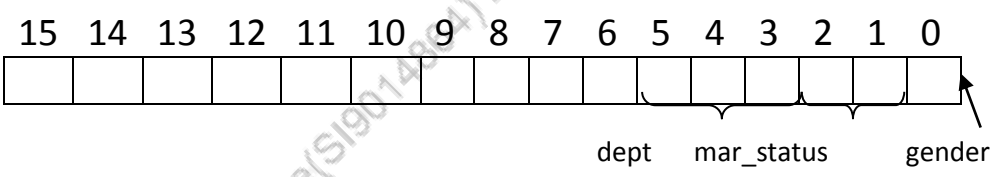
Each member name is followed by a semicolon and an unsigned integer indicating field size. Variable definition and accessing each member is the same as ordinary structures. If you have to store the information of an employee:

1. He/she can be male or female. 1- bit will be sufficient for the purpose.
2. Can be single, married, divorced or widowed. 2-bits are sufficient.
3. Can belong one of 5 different departments. 3-bits will do.

Thus the structure can be declared as:

```
struct emp
{
  unsigned gender: 1;
  unsigned mar_status: 2;
  unsigned dept: 3;
};
```

The number after the colon specifies the size of each bit-field. It is defined as a structure, which is subdivided into 3 bit fields. They have a width of 1, 2 and 3. Hence they occupy 6 bits within a word of memory.

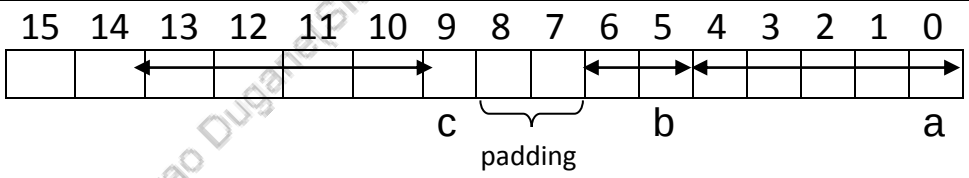



```
/* program demonstrating use of bitfields */
#include<stdio.h>
#define MALE 0
#define FEMALE 1
#define MARRIED 0
#define DIVORCED 1
#define WIDOWED 2
#define RND 0
main()
{
  struct emp
  {
    unsigned gender: 1;
    unsigned mar_status: 2;
    unsigned dept: 3;
```

```
  };
  struct emp e;
  e.gender = MALE;
  e.mar_status = DIVORCED;
  e.dept = RND;
  printf("\nGender=%d \t Marital Status=%d \t
Department=%d",e.gender,  e.mar_status, e.dept);
  printf("\n Requires : %d bytes", sizeof(e));
}
```

Values are assigned to bitfields using `#define`. Bitfields can also be initialized. They can appear in arithmetic expressions. Arrays of bitfields are not allowed. The (&) address operator cannot be applied to a bitfield, pointer cannot access a bitfield and a function cannot return a bitfield. A bitfield without a name can be used for padding.

```
struct xyz{
  unsigned a : 5;
  unsigned b : 2;
  unsigned   : 2;
  unsigned c : 5;
};
```



A bitfield can be forced to start at the beginning of a new word by specifying an unnamed bitfield of width 0.

```
struct xyz{
    unsigned a : 1;
    unsigned b : 2;
    unsigned   : 0;
    unsigned c : 3;
};
```

Advanced ‘C’ Programming

Lab Manual and Appendix

Contents

Before you proceed

This 'C' lab manual aims at giving a complete understanding of the various aspects of 'C' programming language. It encourages exploring different features of 'C' as one proceeds with these sessions.

This manual is divided into lab exercises. Each lab is based on an individual topic.

You are expected to complete each lab session within the allotted time. Every lab is further divided into three sections.

Objective

States what you will achieve after completing a particular lab exercise. At the end of each lab exercise you should keep a track whether the objectives of that exercise have been achieved.

Lab Exercise

Gives a list of the assignments based on that topic. These assignments are compulsory and you are expected to complete all of them.

Advanced Exercise

This section gives more assignments, which you should try after completing the assignments. Completion of these extra assignments ensures that you get the expected clarity of the topic. These assignments are optional but it is advised that you try to complete them.

Note that your evaluation will be based on the following factors:

- Your performance in the classroom. This includes your regularity, initiative and interaction.
- Your performance in the labs. This includes timely completion of assignments and the coding practices followed.
- The test conducted at the end of the course.
- At the beginning of each session the assignments of the previous session will be checked.
- Now, if you have finished reading this step over to the GOOD CODING PRACTICES before you begin.

Good Coding Practices

Guidelines to complete all the 'C' lab assignments

Sequence to be followed to complete the assignments has been discussed below.

Whenever a programmer has to transform a problem statement into a program that a computer can execute, the activity should be split in the following manner:

- *Analysis Phase:* The problem statement should be understood clearly. (In real life this involves study of problem domain).
- *Design Phase:* Think of right algorithm or steps to be followed to achieve the goal. If required, flow charts should be drawn to understand flow of control of a program. Function prototypes should be decided in such a way that these functions can be used as and when required. (In real life this might involve top to bottom approach which leads to deciding the functions/their responsibilities/their interface, user interface, communication with other intelligent controllers if any, etc.)
- *Formulating of testing strategies:* Before you start writing actual code, one should always think how it is going to be tested? How it can be assured that the program specifications are met?
- *Writing of actual code:* Write the code by following good coding practices as given below.
- *Testing Phase:* Test the written code as per your strategies. Confirm user interface.
- *Backup activity:* Keep proper backup of tested code so that if required, already written functions can be reused. Use a properly defined directory structure for storing the data.

Follow the above tips meticulously while building up huge software for any application. Same can be followed for the assignments, because this is the first step towards climbing up the mountain.

Why one should follow good coding practices?

In a software life cycle around 80% of time of completion of cycle, the software is in maintenance/upgrade mode. In this case, people involved at various stages are different. One might have to refer to actual code even after years. Therefore, anybody should be in a position to make the required changes and finish off the

activity. This is possible only if the code is well documented. Actually, writing a right kind of well-documented software is an art along with your technical skills. Necessary documentation should be made.

What are these coding practices?

- Use logical names for variables, functions, file.
- Use uniform notation throughout the code. For example, `float balAmt` – float type variable `balAmt`. This is called camel case notation.
- Create user defined header file to include commonly required variable declarations, function prototypes.
- Code should be properly indented.
- While documenting the code
  - Write the purpose of the piece of code.
  - For functions:
    - Write the task assigned to the function, arguments passed to it, etc.
    - Write a clear comment if any statements have been added for testing purpose.
  - While writing actual code distribute it in following sections :
    - main function, local variable declarations, local function prototypes, user inputs, algorithm processing, function calls, user outputs, end of main, function definitions.

Revision - Arrays

Lab Exercise 1

Objective

- To declare and initialize arrays.

Problem Statement

Write a program to calculate and display the average marks of 5 subjects obtained by a student.

Lab Exercise 2

Objective

- Use pointers to pass the entire array to a function

Problem Statement

Modify Lab Exercise 1 by writing separate functions to accept and display their average.

Lab Exercise 3

Objective

- Pass entire array to a function

Problem Statement

Accept five integers in an array and use separate functions to:

a. Find the maximum and minimum of the integers. Do not sort the array.
b. Multiply each element of the array by 5 and store it in another array. Then display the new array.

Advanced Exercises

1. Write a program that defines a float array and accepts elements from the user. Interchange these elements at all consecutive even and odd positions and display the original and the modified array. 1^{st} with 2^{nd} and 3^{rd} with 4^{th} and so on.
2. Write a program to evaluate the expression $z = x^2 + y^2$, where x and y are two arrays. Each array holds 10 user entered elements. Store this result in another array, z and display it.
   For example, if x[0]=3 and y[0]=4 then, display the result z[0]=25 That is 9 + 16.
3. Write a menu driven program for
   a. Deleting an element from an array. (Position of the element should be considered).
   b. Inserting element into an array. (Position of the element should be considered).

Chapter 1 - Sorting

Lab Exercise 4

Objective

- Use selection sort

Problem Statement

Write a function `sort()`, to sort an array using selection sort and then display the maximum and minimum of the integers. Call the function from `main()`.

Lab Exercise 5

Objective

- Use insertion sort

Problem Statement

Sort the following list of integers using insertion sort.

23 56 4 76 21 7 1 98

Advanced Exercise

Write a program to accept 8 integers in an array. Sort the array using bubble sort. Insert one more integer into the array keeping the array sorted.

Chapter 2 - Strings

Note: Write your own functions for all the exercises. Do not use library functions.

Lab Exercise 6

Objectives

- To declare and initialize strings.
- To pass strings to functions.

Problem Statement

Write a program to accept a string from the user. Write separate functions to perform the following:

a. Find the number of occurrences in the string.
b. Find the number of blank spaces in the string.
c. Find the number of words in the string.
d. Find the total number of all the vowels in the string

Lab Exercise 7

Objective

- To pass strings to functions.

Problem Statement 1

Write a menu driven program that provides user with options to:

a. Compare two strings.
b. Concatenate two strings.
c. Copy one string into another**.**

Problem Statement 2

Write a program to check whether the user entered string is palindrome or not. A string is called a palindrome if the reverse of that string is exactly similar to it. For example, `LEVEL`

Problem Statement 3

Write a function `xStrChr()` which scans a string in search of a character. If the character is found it should return a pointer to the first occurrence of the given character. If the given character is not found, the function should return `NULL`.

Advanced Exercises

1. Write a program to print all possible combinations of a user entered string. For example, if the string is `abcde`, its combinations are: `abcde, bcdea, cdeab, deabc, eabcd`.
2. Write a program that accepts two strings as command line arguments. Concatenate second string with first string and display the concatenated string.
3. Write a program to replace every occurrence of letter ‘a’ with ‘e’ in a user entered string. Ignore the case of the characters.
4. Write a program, which scrolls “Hi” on the screen from left to right and “Hello” on the screen from right to left and when both meet in the center it should display “Bye”.

Chapter 3 - 2D Arrays and Dynamic Memory Allocation

Lab Exercise 8

Objectives

- To declare and initialize 2D arrays.
- To access the elements of a 2D array using array and pointer notations.
- To pass 2D array to a function.

Problem Statement 1

1. Write a program to accept and display 3 x 3 matrix. Write `accept()` and `display()` functions to perform the tasks.
   a. Find the transpose of a matrix and print the transpose using `display()` function..
   b. Accept another matrix of same dimensions. Find the addition of two matrices and print the resultant matrix.

Problem Statement 2

Write a program that calculates the average marks of all the subjects. The number of subjects 'n' is accepted from the user. Allocate memory dynamically for 'n' integers. Free the memory when not in use.

Lab Exercise 9

Objective

- Allocate memory dynamically to store 2D array

Problem Statement

Write a program that accepts the number of subjects and number of students from the user. Allocate memory dynamically to store the marks of the subjects of each student. Display the average marks of each subject. Free the memory when not in use.

Lab Exercise 10

Objectives

- Use pointer to pointer for a 2D character array
- Implement sorting for strings

Problem Statement

Write a program that accepts the number of students from the user. Allocate memory dynamically to store the names of the students entered by the user. Search a name of a student in the list and display appropriate message “Found” or “Not Found”. Use function to perform searching. Free the memory when not in use.

Advanced Exercises

1. Write a program to print multiplication of a 3 x 3 matrix.
2. Write a program to ask the user the type of elements he wants to enter. For example, `int`, `char`, or `float`. Then ask him how many elements he wants to enter. Depending on the data type and number of elements he wants to enter,
    a. allocate the space
    b. accept elements
    c. display elements

    After entering the elements, if the user wants to enter more data, reallocate the space to accept more elements and display them. Free the allocated space if the user does not want to enter more elements.
3. Write a program to replace all the hard coded elements in the upper triangle of a 4 x 4 matrix above diagonal, by zero.
4. Modify the assignment 2 by adding a function that sorts the names in the ascending order of their length. Call the function from `main()`. Use bubble sort technique. Free the memory when not in use

Chapter 4 - Structure and Union

Lab Exercise 11

Objectives

- To define a structure.
- To declare and use structure type variables.
- To use `typedef`.

Problem Statement

Define a structure "`Book`" having members – `bookId` , `title`, `price`. Use `typedef` to name this structure as "`BOOK`". Accept the data for a book and display the record.

Lab Exercise 12

Objective

- Pass a structure to a function

Problem Statement

Modify Lab Exercise 1 to accept and display the data of a particular book using functions.

Lab Exercise 13

Objective

- Use an array of structures

Problem Statement

Modify the above assignment to hold records of 5 books. Display the records of all the books using a function.

Lab Exercise 14

Objective

- Create a copy of a structure variable

Problem Statement

Write a program to copy one structure to another –

a. On element by element basis.
b. Copying an entire structure to another.

Lab Exercise 15

Objectives

- To implement nested structure.
- To implement `enum`.

Problem Statement

Define an enumerated type "`Category`". It should have values like `MAGAZINES`, `NOVELS`, `ENCYCLOPEDEA`, `COOKING`, etc. Also declare a structure "`publisher`" that has `pubName` and `pubAddress` as members. Modify the structure "`book`" to include variables of type `category` and `publisher`. Perform the following operations on the structure:

a. Accept the data for a book
b. Display the data of the book (the category should be displayed as a string on the screen along with other data)

Lab Exercise 16

Objective

- To allocate memory dynamically for a structure

Problem Statement

Considering the same structure defined above, write a program to accept the number of records from the user at runtime. Accordingly allocate memory for the

number of books at runtime. The program should also display a menu to perform following operations:

a. Add record
b. Delete record (based on `bookId`)
c. Modify record (based on `bookId`)
d. Display records

Lab Exercise 17

Objective

- Implement `enum` and `union`

Problem Statement

Define a structure `Student` having members `rollNo`, `name`, `section` and `performance`. The section is either `primary` or `secondary`. Performance for students in primary section is stated as grades and for the students in secondary section is stated as percentage. Declare an array for ten students. Accept information and display division wise list of students and their performance.

Advanced Exercises

1. Modify assignment 3 above, by defining a function `changePrice()`. This function checks the price of the book and reduces it by 10% of the original price is greater than 200, for the purpose of sale being announced by the book shop. Note that, the function should not return any value.
2. Write a program that prompts user to enter today's date and prints tomorrow's date.
3. Make a digital clock that displays time in the hh:mm:ss format. Write a function called `getTime()` to accept the current time from the user. This clock will keep on showing the right time as long as the program is in the running mode. You can use built-in function such as `kbhit()`. You will need to include `conio.h`

4. Define following structures containing the members given below them :

bookMast	**bookInfo**	**member**	**memberBook**
bookId	bookId	memberId	bookId
title	copyNo	memberName	memberId
price	noOfCopies	issueDt	
category			
publisher			

A book can be kept only for 30 days after its date of issue. In case it is kept after 30 days from the date of issue, the fine amount charged per extra day is Rs. 2/-.

5. Write a program that gives user a choice to:
    a. Add, Delete, Modify: the book information and member information
    b. Display: member wise overdue, a list of books, their authors and number of copies of that book, list of all members and the books they have been taken.

Chapter 5 – File I/O

Lab Exercise 18

Objective

- Use functions related to file handling.

Problem Statement

1. Create a text file containing string data. Write a program to count spaces, tabs, and newline characters contained in this file.

Lab Exercise 19

Objective

- Use command prompt to execute a program

Problem Statement

Write a program that uses command line arguments to simulate the copy command in DOS.

Lab Exercise 20

Objective

- Use text files for storing and retrieving data.

Problem Statement

Accept name and address of a person from the user and write it to a text file. The program should accept the input as long as the user wishes to enter the data. Write a program that performs following operations on the text file:

a. If that file already contains data, then add to the existing data.
b. Display the information from this file on the console.
c. If the file to be read does not exist, display an error message to the user using standard library functions for error handling.

Lab Exercise 21

Objectives

- Use binary files
- Perform record I/O using functions like `fread()` and `fwrite()`.
- Access records randomly.

Problem Statement

Write a program to make a menu based Telephone Directory. Provide the user options to:

a. Create new entries.
b. Delete existing records (based on telephone number)
c. Modify the records (based on telephone number)
d. Search and display records.
e. Each record contains full name of a person, his address and telephone number. The data should be alphabetically sorted according to the names in binary files.
f. Flash appropriate error messages for appropriate situations. E.g. in case user is searching for a name that does not exist, flash an error message NAME NOT LISTED.

Advanced Exercises

1) Write a program to split an existing file into two files. Ask the user to specify the size of the first file.
2) Write a menu driven program to:
   a. Count occurrences of a user-entered character in a file.
   b. Search for a particular word in a file and replace it with a user-entered word. The word to be replaced may be hard coded or accepted from the user.
   c. Undo changes made to a file.
   d. Rename a file.

3) Write a program that will encode and decode multiple lines of text, using the encoding/decoding procedure. Store encoded text within data file, so that it can be retrieved and decoded at any time. The program should include the following features:
   a. Enter text from keyboard, encode the text and store the encoded text in data file
   b. Retrieve the encoded text and display it in its encoded form
   c. Retrieve the encoded text, decode it and then display the decoded text
   d. End the computation
   e. Test the program using several lines of text of your choice.

Chapter 6 - Linked List

Lab Exercise 22

Objective

- Implement a linked list.

Problem Statement 1

Declare a self referential structure to represent an employee in ABC organization. You should store the details of an employee like `empId`, `empName` and `empSalary`.

Problem Statement 2

4. Create a singly linked list to store the details of different employees in the organization. To do this, implement a menu driven program that has following options:
   a. Create a node
   b. Append a node
   c. Delete a node based on `empId`
   d. Modify data based on `empId`
   e. Display details of all the employees
   f. Count the number of nodes in the list

Advanced Exercises

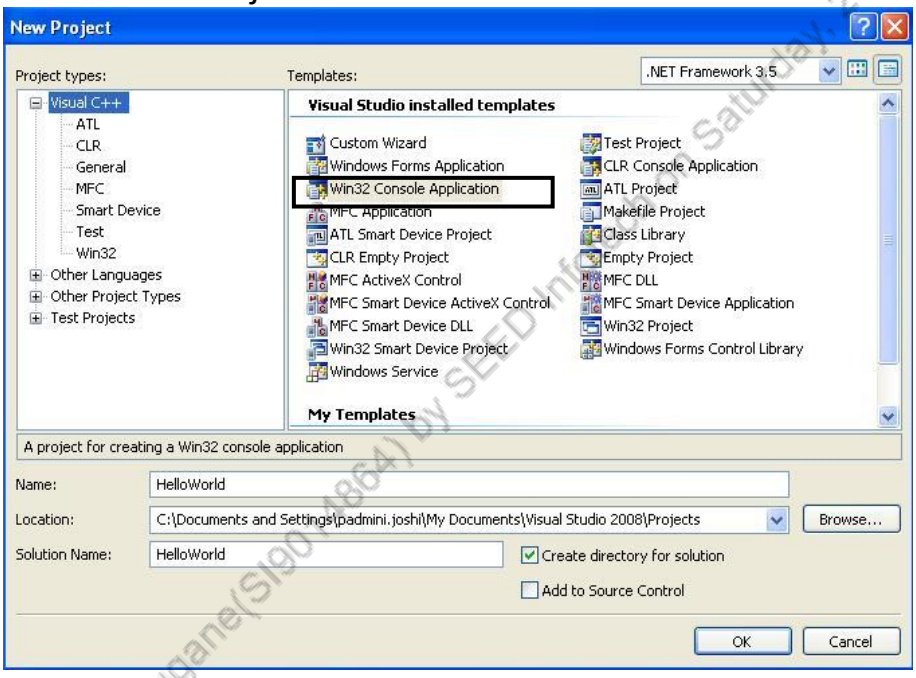
1. Modify the already created linked list to insert an employee either in the beginning, middle or end of the list.
2. Create a new linked list of Employees where new employee information would be inserted in sorted order.

Appendix B

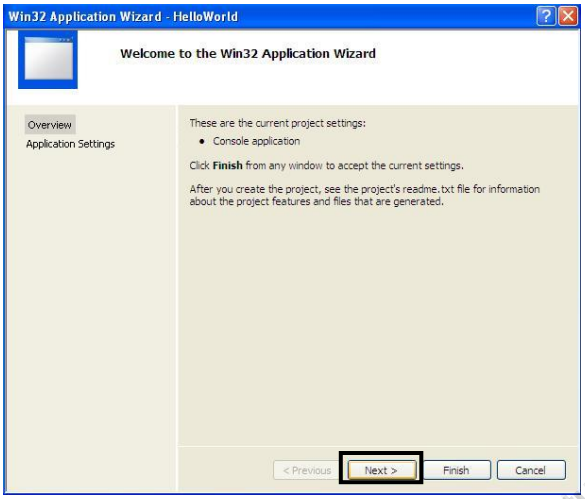
Introduction to Visual Studio IDE

Starting Visual Studio

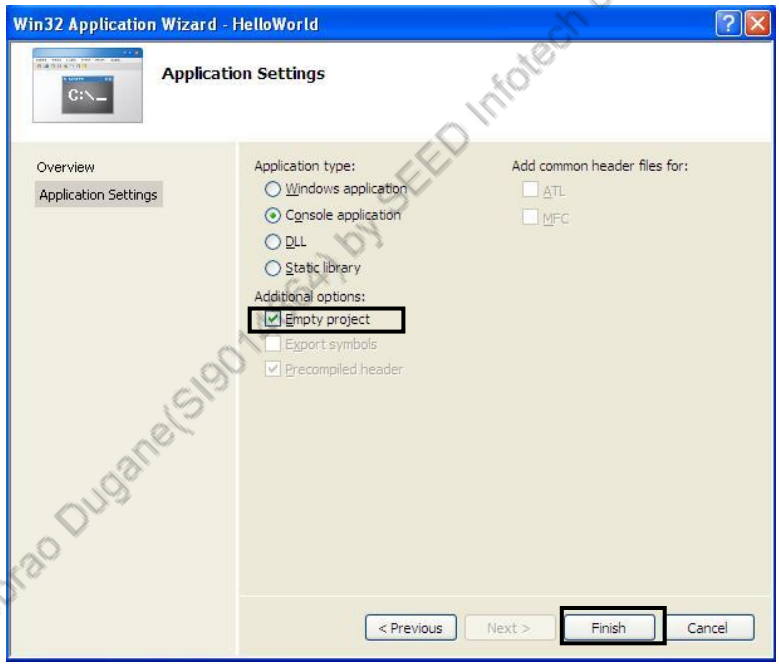
1. Start → All Programs → Microsoft Visual Studio 2008 → Microsoft Visual Studio 2008
2. Click on File →New →Project



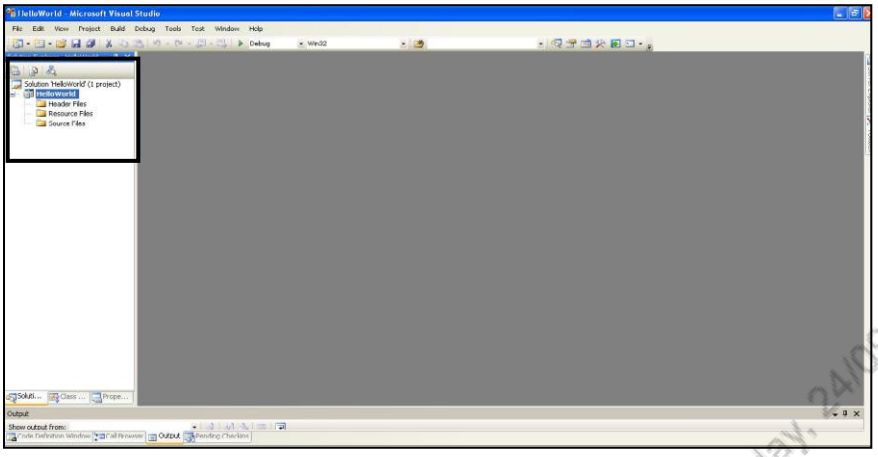
3. Select Visual C++ as Project Type; in the templates section select Win32 Console Application as your template.
4. Give a Suitable name to your project; if required, browse to the specific location to store the project.
5. Click on OK button.
6. As a result of step 5, a new Win32 Application wizard window will appear.
7. Click the Next button on the first screen.



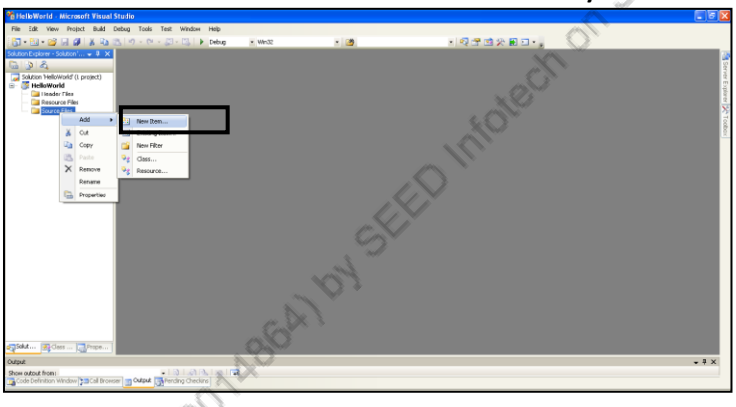
8. On the second screen check the Empty project additional option.
9. Then click on the Finish button.



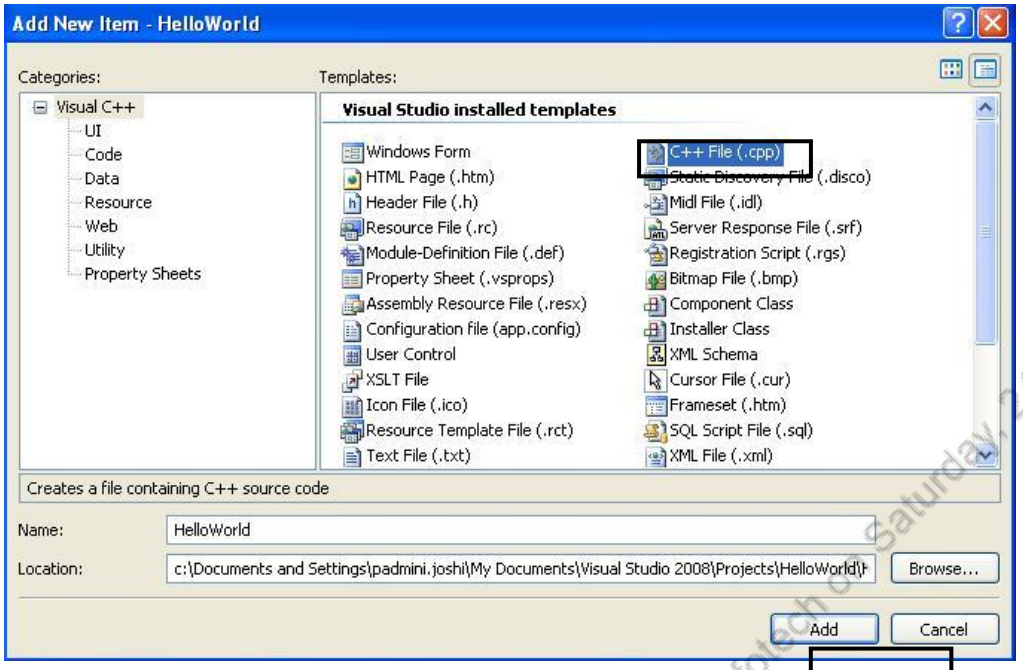
10. As a result of step 9, project window will appear.



11. Now in the solution explorer panel, right click on the Source Files folder, hover on Add menu and then click on New item entry in the submenu.

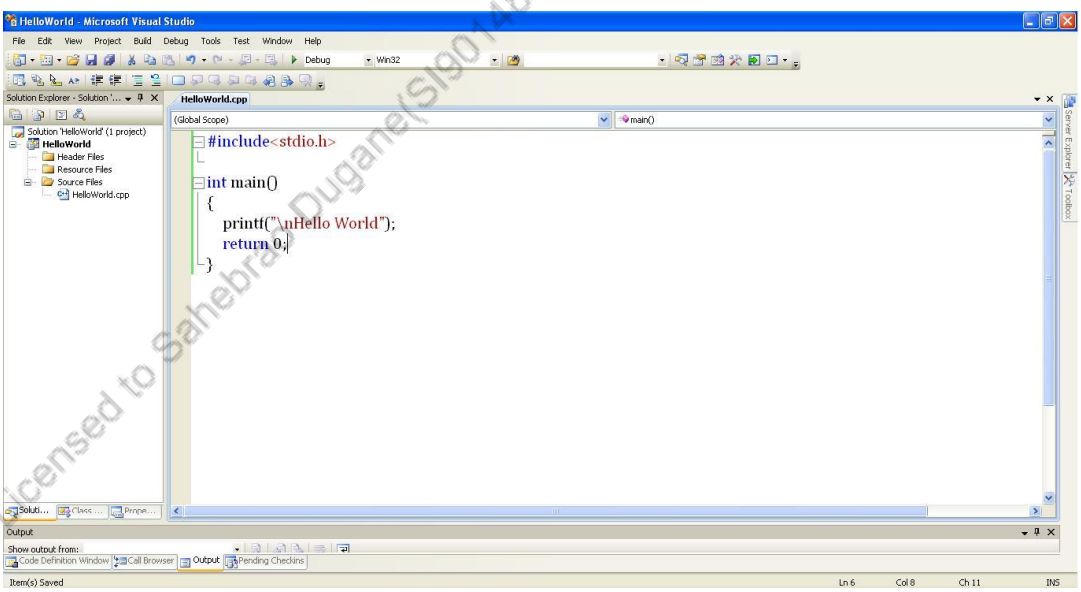


Add New Item window will appear. Now, select Visual C++ as category, C++ File (.cpp) as template and give a suitable name to your program. You may browse and go to the desired storage location if you want.

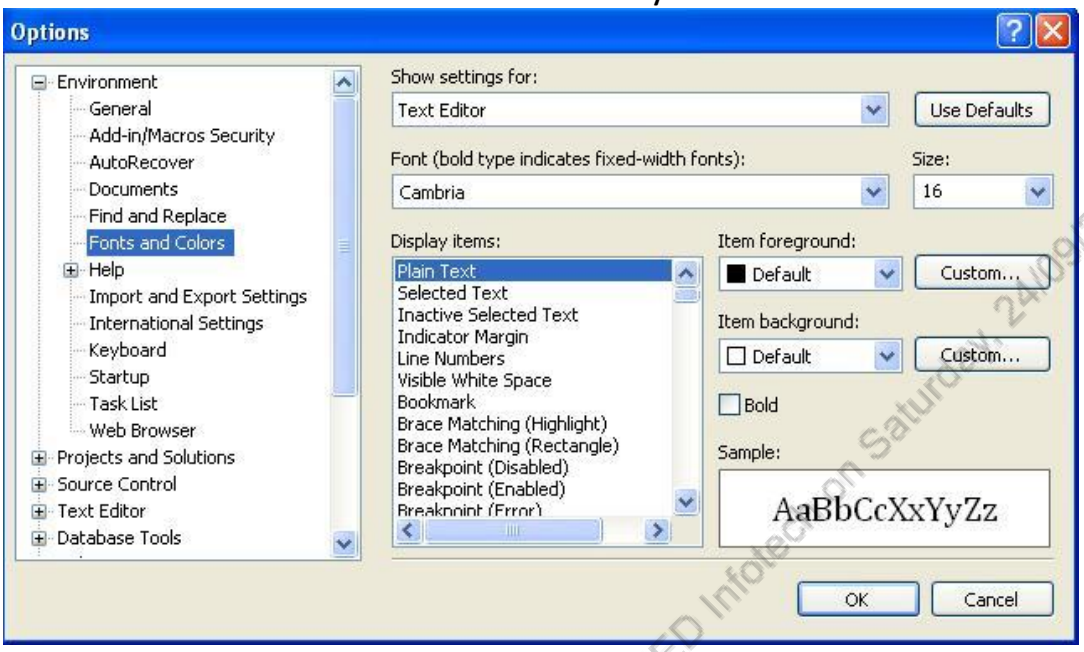


12. Click on the Add button.
13. As a result of step 13, code editor window will take the center stage.

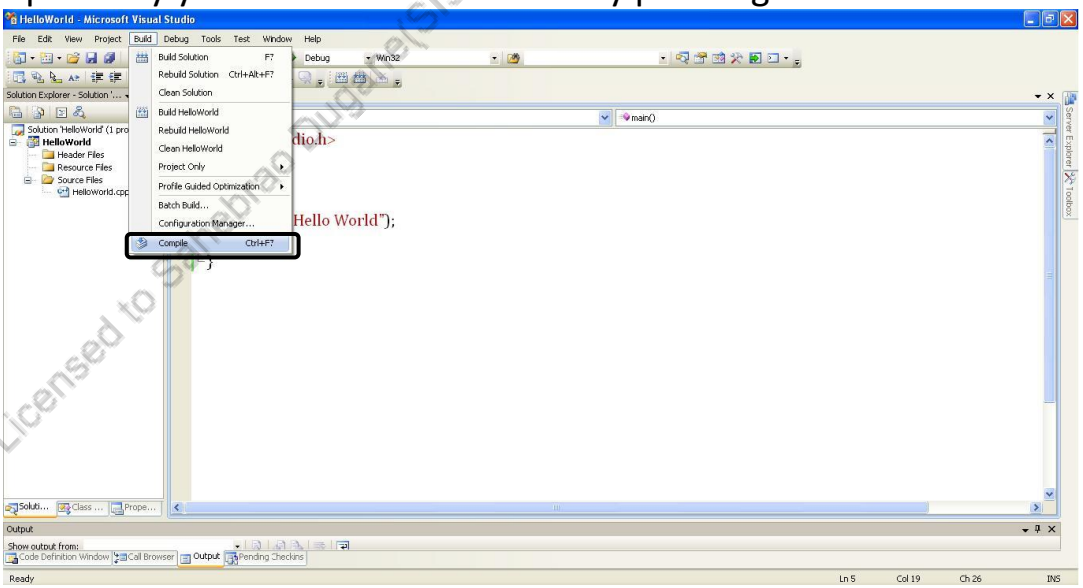
Type your code in the editor window space.



14. If you want to change the font size, go to Tools →> Options →Fonts and Colors. And select font and color of your choice.

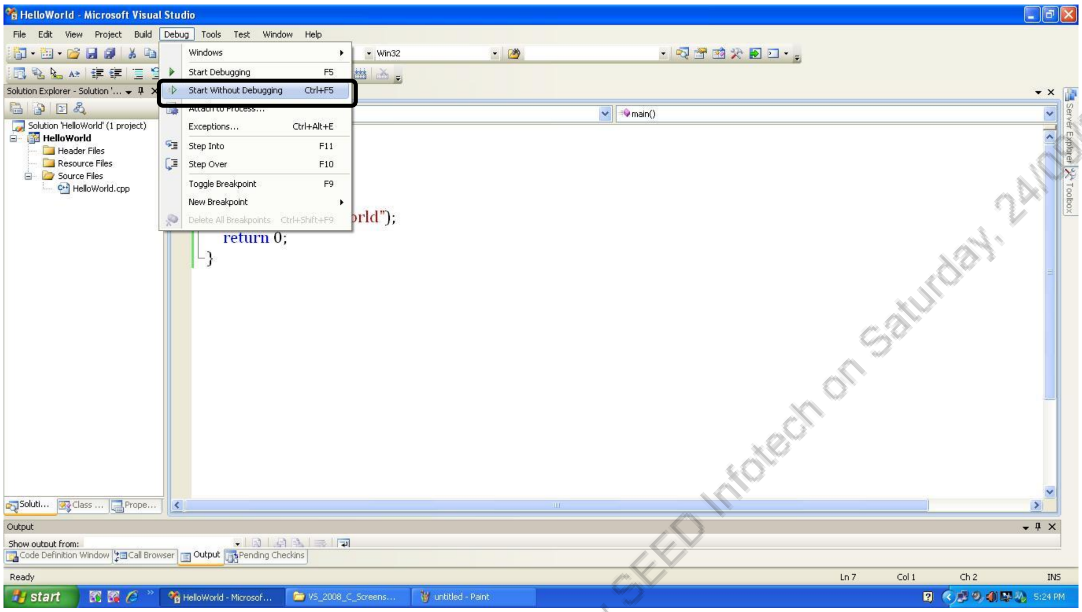


15. The program is ready. You have to compile, build and execute it.
16. To compile the program, click on Build menu option, and then click on compile menu entry.
Optionally you can also achieve this by pressing Ctrl + F7.

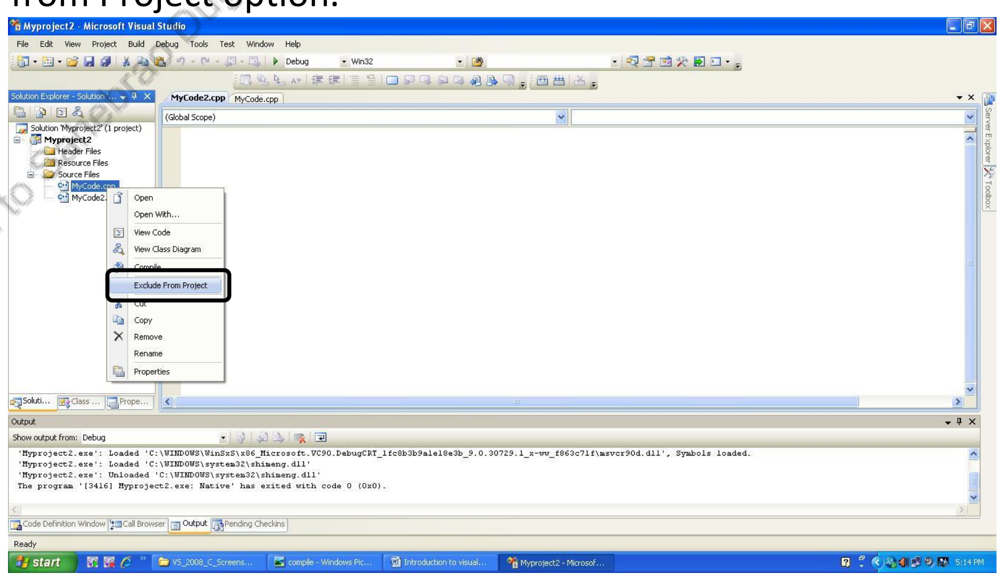


17. After successful compilation you need to build the solution. Go to build menu again and click on Build Solution menu option. Optionally you can also achieve this by pressing F7.

18. In order to execute your program, go to Debug menu option and click on start without debugging menu option. Optionally you can also achieve this by pressing Ctrl+ F5.

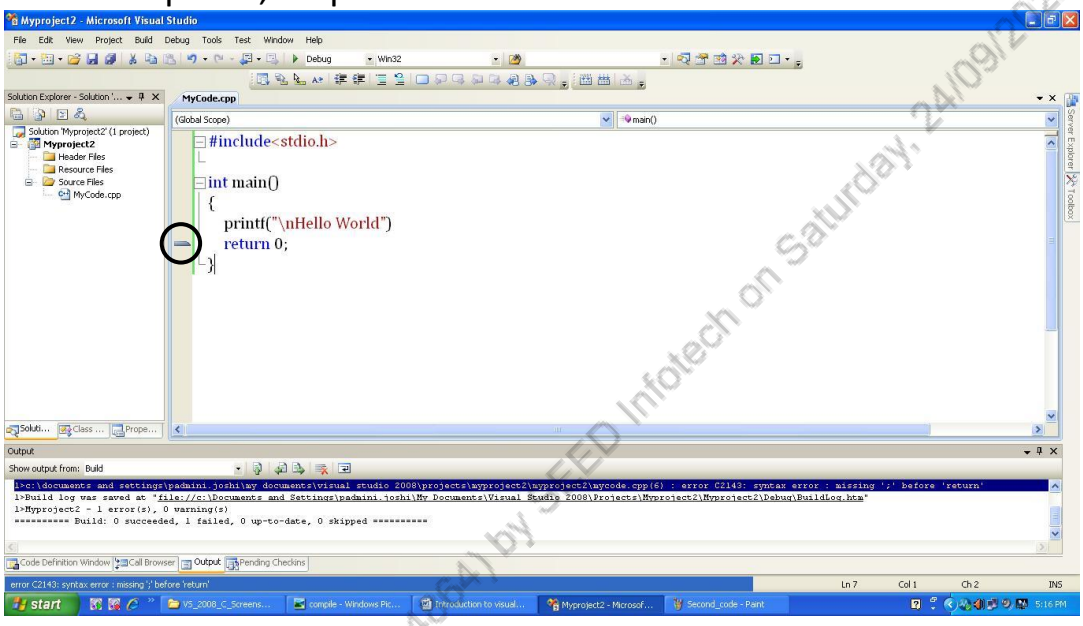


19. If you want you can add another source code (.cpp) file to your project; but as only one main() function can act as the starting point of execution. Make sure to exclude the earlier source code from the project.
20. To do so, right click on the source file to be excluded, and select Exclude from Project option.

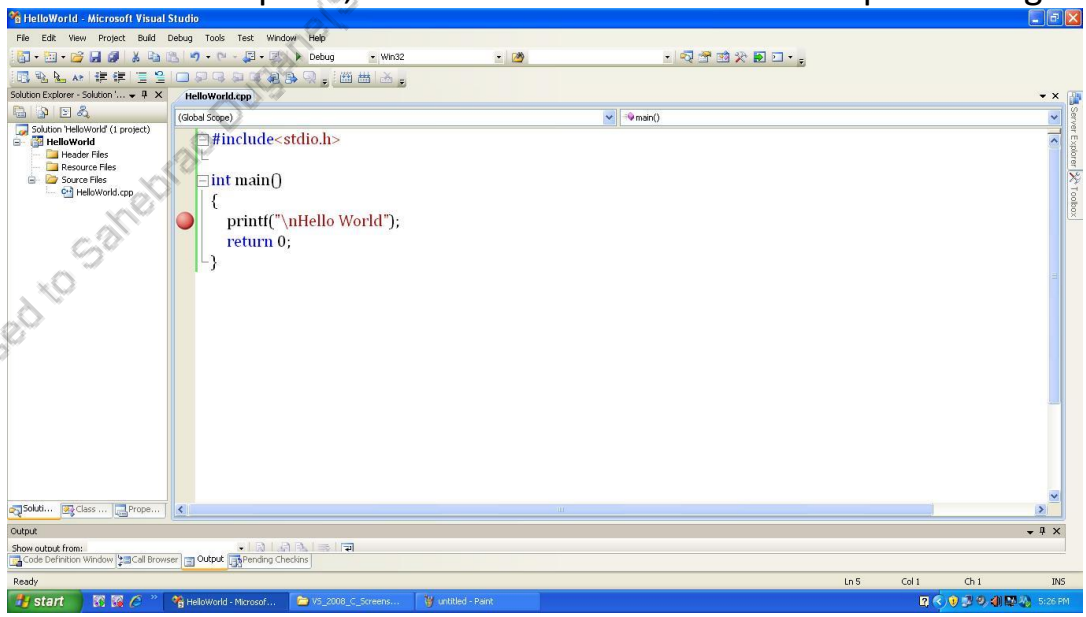


Debugging

When there are compiler errors, the line in the code, in which they occur, can be found out by pressing the F4 key. The error will be pointed by a blue arrow in the editor and in the message window the error message is highlighted. Explore the debug menu further to find the different debugging options like Step into, Step over etc.



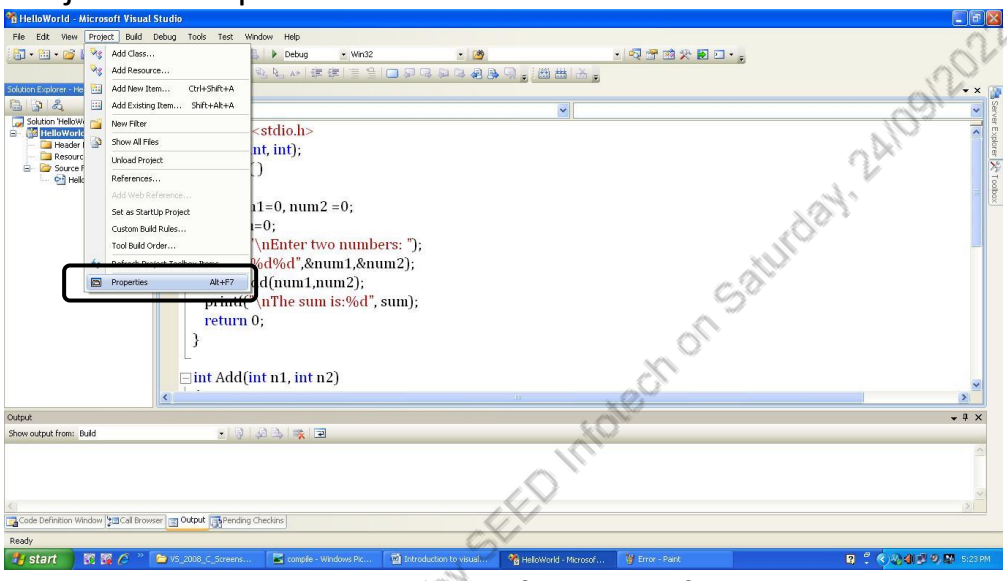
To insert breakpoint, take the cursor to the required line and press F9 key. To remove the breakpoint, take the cursor to the line and press F9 again.



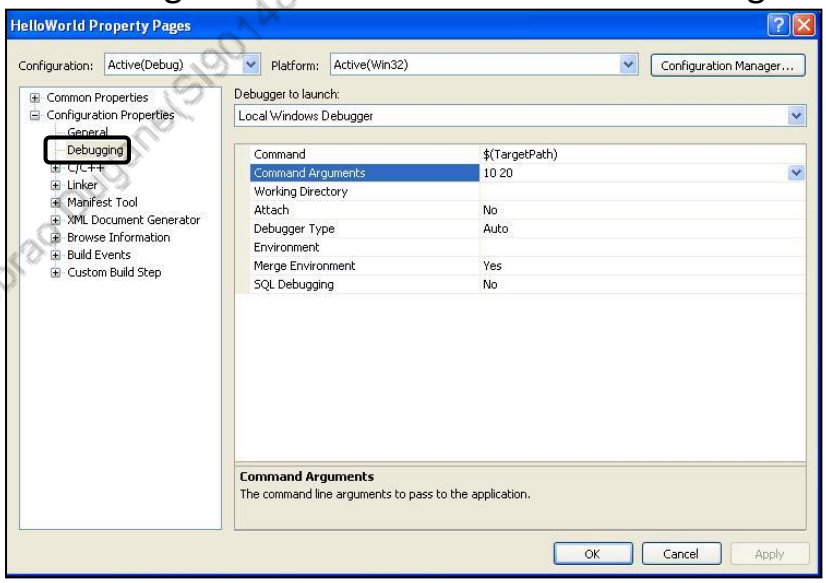
Using Command Line Arguments

There is another way to execute a program that needs some input – by command prompt and passing the input using the command prompt.
To do so in Visual Studio:

1. Select Project → Properties

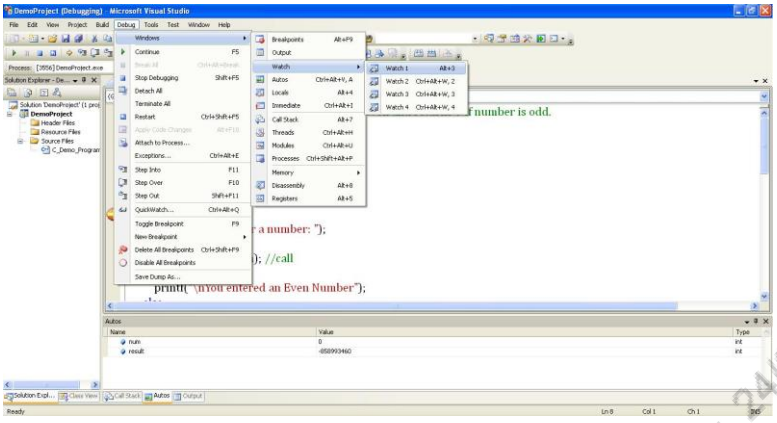


2. Then, select the Debugging tab. For further reference see the snapshot below. Here we have given 10 and 20 as command line arguments.

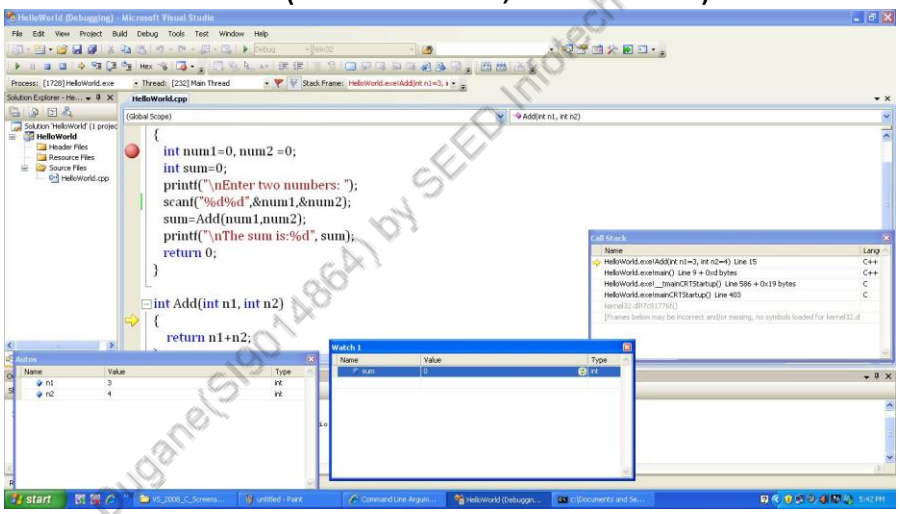


Using the Watch Window

When you start debugging, in the window below you can type the names of variables whose values you want to watch during the program execution. This can be seen in the snap shot below:



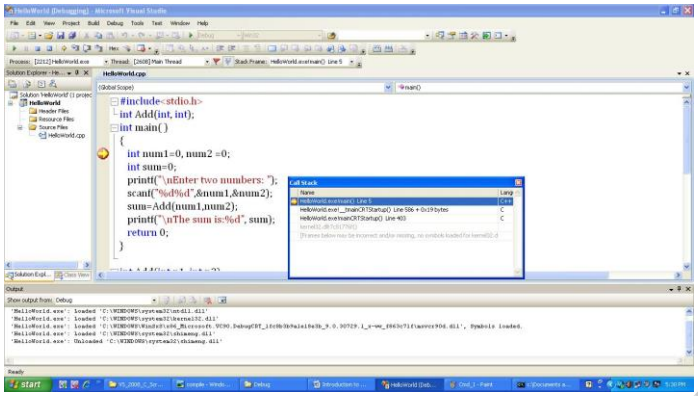
To open the watch window, first start debugging. Then go to Debug menu option, and then hover on windows option and in the subsequent submenu, hover on watch menu item. In the last sub menu you can select any of the watch windows (from watch1, watch2 etc).



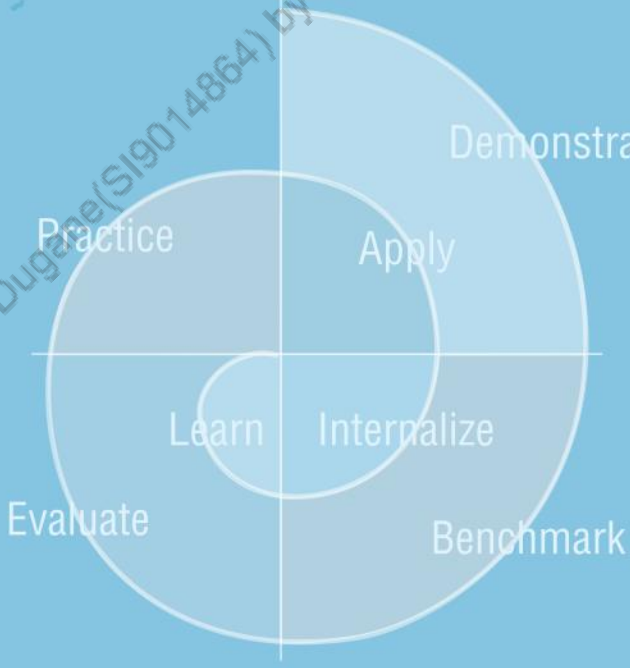
In the Autos window all the variables in the function under execution are listed whereas in the watch window are the variables whose values you want to watch.

Using the Call Stack

The call stack (for various function calls) can be seen in the run mode. To do so, debug the code by pressing F5 and then go to debug menu, hover on windows and select Call Stack menu entry in sub menu. Optionally you can achieve this by pressing Alt+7.



Here the various function calls can be seen on the stack. When a function is called, it is placed on stack.



Avani publications

N·S·D·C National Skill Development Corporation
Transforming the skill landscape
Training Partner

seed
Official Curriculum