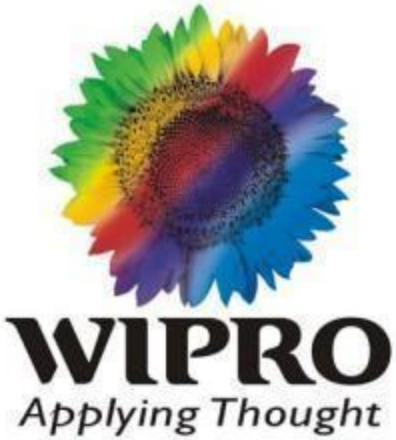


C Programming and Data Structures

Agenda

1. Introduction to C
2. Operators & Control Structures
3. Arrays and Pointers
4. Functions and Recursion
5. User-Defined Data Types
6. File Input / Output

Agenda

7	Introduction to Data Structures
8	Lists
9	Stacks & Queues
10	Binary Trees

Introduction to C

Module 1

Objectives

At the end of this module, you will be able to:

- Explain genesis of C programming language
- List features of C programming language
- State the data types in C
- Perform basic Input and output operations

Duration: 3 hrs

Computer Languages

- Machine Language: A Binary Language composed of 0s and 1s – that is specific to each computer.

- Assembly Language: Binary instructions are given abbreviated names called *mnemonics* which form the Assembly Language. Assembly Language is specific to a given machine.

- High Level Language: Programming language intended to be *machine-independent* is called High level language. Instructions are called as *statements*.

  Eg:FORTRON,PASCAL,C ..etc.

Assemblers and Compilers

- Assembler is a program that translates the mnemonics into the corresponding binary machine codes of the microprocessor

  Eg: KEIL's **a51**

- Compiler is a program that translates the source code (statements) into the machine language (object code) compatible with the microprocessor used in the system

  Eg: TURBO C,
  BORLAND C
  GCC ..etc.

The C Language

- C is a robust language whose rich set of *built-in functions* and *operators* can be used to write any complex program.

- C compiler combines the capabilities of an *assembly language* with features of *high-level language* and therefore it is well suited for writing both system software and business packages.

- Programs written in C are efficient and fast .This is due to its variety of *data types* and powerful operators.

C History

- Developed between 1969 and 1973 along with Unix
- Ken Thompson created the B language in 1969 from Martin Richard's BCPL
- Dennis Ritchie of Bell Laboratories later converted B into C by retaining most of B's syntax in 1972 and wrote the first compiler.
- Designed for systems programming
  - Operating systems
  - Utility programs
  - Compilers
  - Filters

Ken Thompson & Dennis Ritchie

Welcome to C Programming

```
#include <stdio.h>

void main()
{
  printf("Welcome to C Programming!\n");
}
```

Preprocessor used to share information among source files

Program mostly a collection of functions

"main" function special: the entry point

"void" qualifier indicates function does not return anything

I/O performed by a library function: not included in the language

Pieces of C

- Types and Variables
  - Definitions of data in memory
- Expressions
  - Arithmetic, logical, and assignment operators in an infix notation
- Statements
  - Sequences of conditional, iteration, and branching instructions
- Functions
  - Groups of statements and variables invoked recursively

Constants, Variables keywords and Identifiers

- Constants are the quantities which do not change during the execution of a program.
- Variables are the quantities which may vary during the execution of a program
- Keywords are the words whose meaning has already been explained to the C compiler.
- Identifiers are the user defined names given to variables,functions and arrays.

Rules for constructing Constants

- Rules for constructing Integer Constants
- Rules for Constructing Real Constants
- Rules for Constructing Character Constants

C variables

- Variable names are names given to the memory locations of a computer where different constants are stored.

- These locations can contain integer,real or character constants.

- The rules of constructing variable names of all types are same.
  - A variable name is any combination of 1 to 8 alphabets, digits or underscores.
  - The first character must be alphabet.
  - No commas and blanks are allowed within the variable name.
  - No special symbol other than an underscore can be used in a variable name.
  - Eg: si_int, pop_e_89

C Data Type

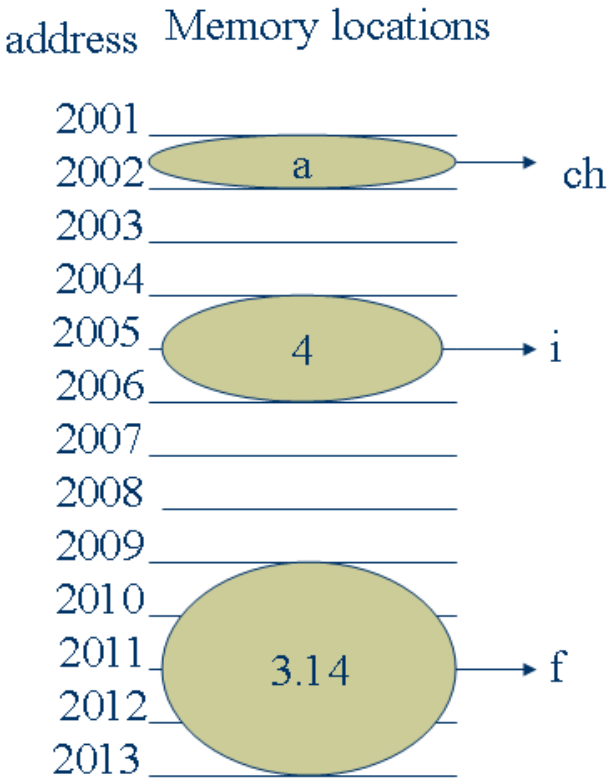
Data Type	Bytes	Format
char	1	%c
Unsigned int	2	%u
Signed int	2	%d
Long signed int	4	%ld
Long unsigned int	4	%ul
Float	4	%f
double	8	%lf
Long double	10	%Lf

Memory allocation is done by OS.

int i=4;

float f=3.14;

char ch='a';



C Keywords

- Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names .

- Doing so is trying to assign a new name to the keyword which is not allowed by the *compiler.* Keywords are also called as Reserved words.

- There are only 32 keywords available in C.

Keywords			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C Instructions

- Type declaration : To declare the type of variables used in the program.
- Input/Output instructions: To perform the function of supplying input data to a program and obtaining the output results from it.
- Arithmetic instructions: To perform arithmetic operations b/n constants and variables.
- Control instructions: To control the sequence of execution of various statements.

C program

- Any C program is basically a collection of *functions* that are supported by the C library.*User defined functions* can be added to the C library.
- The **main( )** is a special function used by the C system to tell the computer where the program starts.
- The set of statements belonging to a function are enclosed within a pair of braces,**{ }**.
- Any *Variable* used in the program must be declared before using it.
- C program statements are written in *lowercase* letters and end with *semicolon*.
- *Uppercase* letters are used only for symbolic constants.

Basic Structure of C program

Link Section
Definition Section
Global Declaration Section

```
main( )
{
      Declaration Part(local)
      Execution Part
       function call
 }
function()
 {
     Declaration Part(local)
      Execution Part
  }
```

printf() statement

printf( ) is predefined ,standard C function for printing output.

The general form of printf() is,

printf("<format string>",<list of variables>);

<format string> could be,

%f for printing real values.

%d for printing integer values.

%c for printing character values.

Eg: printf("%f",si);

printf("%d",i);

scanf() statement

scanf( ) is a predefined,standard C function used to input data from keyboard.

The general form of scanf() is

scanf("<format string>",<address of variables>);

Eg: scanf("%d",&n);

scanf("%f",&interest);

& is pointer operator which specifies the address of that variable.

Example Program

```
1  /*   - - -- - - -*/
2     Addition program */
3  #include <stdio.h>
4
5  int main()
6  {
7     int integer1, integer2, sum;         /* declaration */
8
9     printf( "Enter first integer\n" );   /* prompt */
10    scanf( "%d", &integer1 );            /* read an integer */
11    printf( "Enter second integer\n" );  /* prompt */
12    scanf( "%d", &integer2 );            /* read an integer */
13    sum = integer1 + integer2;           /* assignment of sum */
14    printf( "Sum is %d\n", sum );        /* print sum */
15
16    return 0;  /* indicate that program ended successfully */
17 }
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Outline

1. Initialize variables
2. Input
   2.1 Sum
3. Print

Program Output

Hands-on: 1hour

Purpose

- Exposure to Program development environment, compilation, debugging and running a "Welcome program" program

- Working with *printf* & *scanf* function for integer data type.

Summary

In this module, we discussed:

- Features of C programming language
- Data types in C
- Basic Input and output operations

Operators & Control Structures

Module 2

Objectives

At the end of this module, you will be able to:

- Classify and use operators
- Apply the unary, binary, ternary, compound assignment operators, increment/decrement operators
- Use character arithmetic and understand the rules of conversion between different data types
- Write simple programs with conditional constructs and iterative constructs

Duration: 4hrs

C Expressions & Operators

- Traditional mathematical expressions

  y = a*x*x + b*x + c;

- An Operator is a symbol that tells the computer to perform certain mathematical or logic manipulations. Operators are used in programs to manipulate data and variables.

- C Operators can be classified as
  - Arithmetic Operators. + – * / %
  - Logical Operators. && || !
  - Relational Operators. == != < <= > >=
  - Bit wise Operators & | ^ ~
  - Assignment Operators. = += -=
  - Increment & Decrement Operators. ++ --
  - Conditional Operators. ? :

Arithmetic Operators

- Arithmetic operators:

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	*f + 7*	f + 7
Subtraction	-	*p – c*	p - c
Multiplication	*	*bm*	b * m
Division	/	*x / y*	x / y
Modulus	%	*r mod s*	r % s

- Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication,Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or –	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Equality and Relational Operators

WIPRO
Applying Thought

Decision Making operators

Standard algebraic equality operator or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Equality Operators			
=	==	`x == y`	`x` is equal to `y`
not =	!=	`x != y`	`x` is not equal to `y`
Relational Operators			
>	>	`x > y`	`x` is greater than `y`
<	<	`x < y`	`x` is less than `y`
>=	>=	`x >= y`	`x` is greater than or equal to `y`
<=	<=	`x <= y`	`x` is less than or equal to `y`

Increment & Decrement Operators

- The increment and decrement operators are ***unary operators.***
- The operator **++** adds 1 to the operand while **- -**subtracts 1.They take the following form:

  ++m(pre increment); or m++(post increment);

  --m(pre decrement); or m--(post decrement);

Prefix operator *first does the operation* and then result is assigned to the variable.

Postfix operator *first assigns the value to the variable* and then does the operation.

Operator	Operation
++	Increment
- -	Decrement

Conditional/ Ternary Operator

- A Ternary operator pair " **? :**"is available in C to construct conditional expression of the form

expression1 ? expression2 : expression3

Eg:

```
a=10;
b=15;
x=(a>b) ? a : b;
```

x will be assigned to the value of b.

Ternary Operator: Example

Write a program to display the greatest of two numbers.

```
#include <stdio.h>
main()
{
      int number1, number2, greater;
      printf("\nEnter the first number ");
      scanf("%d",&number1);
      printf("\nEnter the second number");
      scanf("%d",&number2);
      greater=number1 > number2?number1 : number2;
      printf("\nThe greater number is %d",greater);
```

Logical Operators

- The Logical AND Operator

  a && b;

  (a < b) && (c < d)

- The Logical OR Operator

  a || b

  (a < b) || (c < d)

- The Logical NOT Operator

  !a

  !(x + 7)

Operator	Operation
&&	Logical AND
||	Logical OR
!	Logical NOT

Values of Logical Expressions			
a	b	a && b	a || b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Assignment Operators

Operator	Operation
a = 1	Assignment
+ =1	a=a+1
a- =1	a=a-1
a* = b	a=a*b
a/ = b	a=a/b
a%= b	a=a%b

Type Conversions

- When an operator has operands of different types, they are converted to a common type according to a small number of rules.

- The following informal set of rules will suffice:
  - If either operand is long double, convert the other to long double.
  - Otherwise, if either operand is double, convert the other to double.
  - Otherwise, if either operand is float, convert the other to float.
  - Otherwise, convert char and short to int.
  - Then, if either operand is long, convert the other to long.

- Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

Type Conversions (Contd.).

- Consider the following assignments:

```
int i;
char c;
i = c;
```

- In the aforesaid assignment, the data type on the right (char) is converted to the data type on the left (int) which is the type of the result.

- If x is float and i is int, then x = i and i = x both cause conversions; float to int causes truncation of any fractional part. When a double is converted to float, whether the value is rounded or truncated is implementation dependent.

Explicit Type Conversion

- In the construction (type name) expression, the expression is converted to the named type by the conversion rules above.

- Consider the following example:

```
int i, j;
double d;
d = i / j; /* double assigned the result of the division of two integers */
```

- The problem with the above assignment is that the fractional portion of the above division is lost, and d is effectively assigned the integer quotient of the two integers i and j.

Explicit Type Conversion (Contd.).

- To resolve this, the variable i can be type cast into a double as in:

  d = (double)i / j;

- int i is converted to a double using the explicit cast operator. Since one of the operands is a double, the other operand (j) is also converted to a double, and the result is a double.

- The result is assigned to a double with no side effects. The cast operator can be done only on the right-hand side of an assignment.

Bit-wise Operators

- Bit-wise operators allow direct manipulation of individual bits within a word.
- These Bit-manipulations are used in setting a particular bit or group of bits to 1 or 0.
- All these operators work only on integer type operands.
  - Bit-Wise logical operators:
  - Bit-Wise AND (&)
  - Bit-Wise OR (|)
  - Bit-Wise exclusive OR (^)
- These are binary operators and require two integer-type operands.
  - These operators work on their operands bit by bit starting from the least significant bit

Operator	Operation
&	BitwiseAND
|	BitwiseOR
~	One's Compliment
^	Exclusive OR
<<	Left shift
>>	Right shift

Size of operator

- The sizeof is a compile time operator and,when used with an operator, it returns the number of bytes the operand occupies

- The operand may be a variable,a constant or a data type qualifier

```
Eg:   int m,n, sum;
      m=sizeof(sum);
      n=sizeof(long int);
```

- The sizeof operator is normally used to determine the lengths of arrays and structures

- It is also used to allocate memory space dynamically to variables during execution of a program

Control Structures

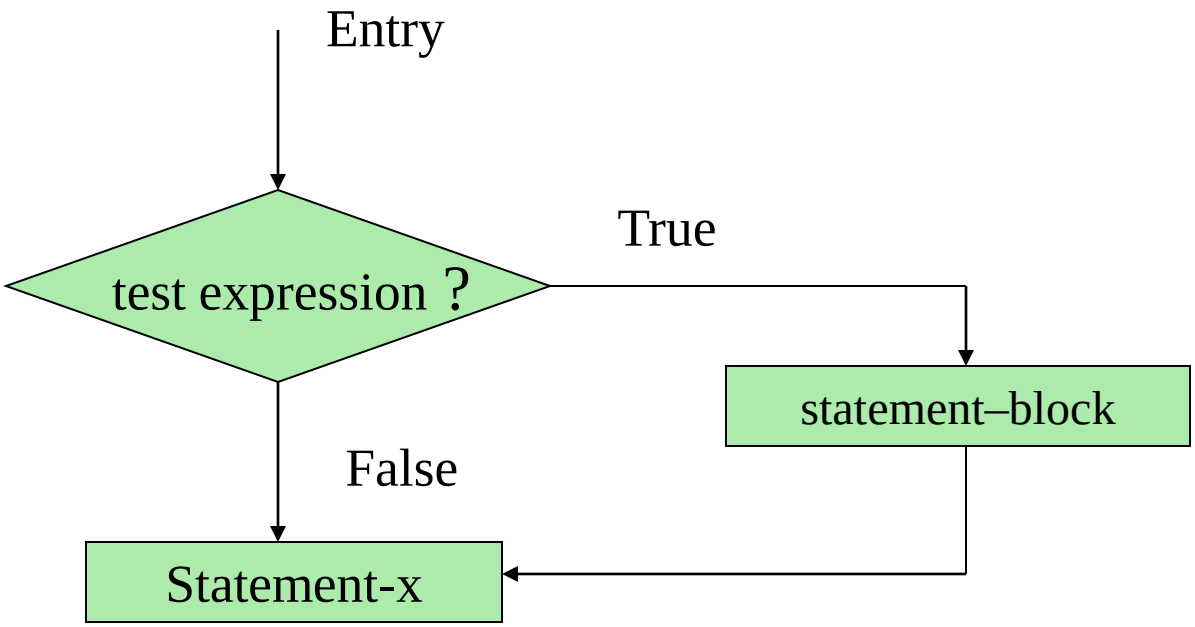
- These are the statements which alter the normal (sequential)execution flow of a program.
- There are three types of Control Statements in C

Decision Making	Looping	Others
If	for	break
If - else	while	continue
switch	Do-while	goto

if statement

The general form :

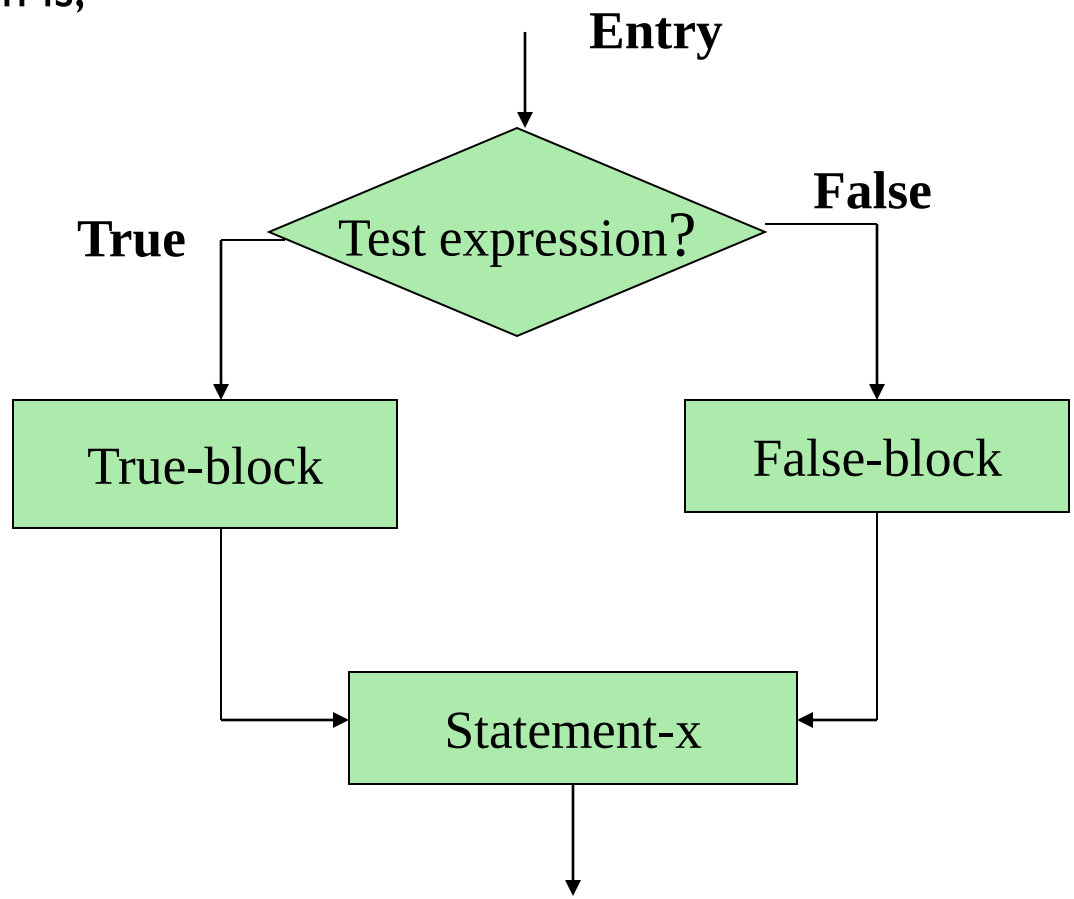
```
if(test expression)
{
    statement–block;
 }
 statement-x;
```



if….else statement

The if….else statement is an extension of simple if statement. The general form is,

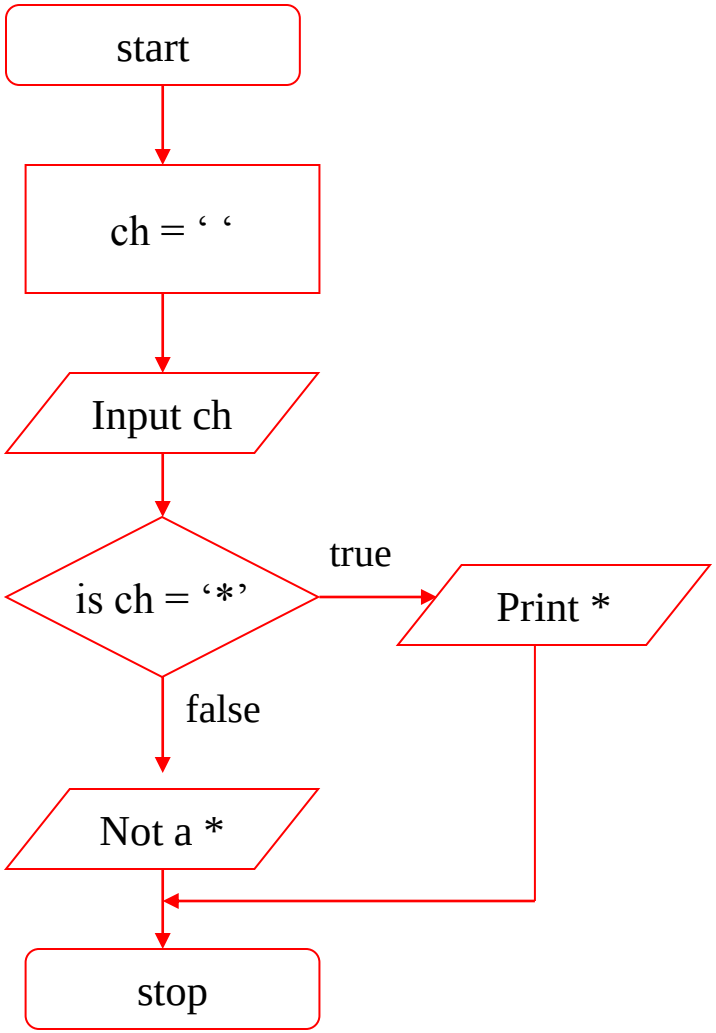
```
if(test expression)
    {
        true-block statements;
     }
        else
     {
        False-block statements;
     }
      statement-x;
```



Example - *if- else* Construct

The following function checks whether the character entered by the user is a star (*) character.

```
#include <stdio.h>
main( )
{
   char ch;
   ch = getchar( );
   fflush(stdin);
   if (ch = = '*')
     puts ("You have entered the
   star character");
   else
      puts ("You have not entered
   the star character");
}
```



Example - *if- else* Construct (Contd.).

The earlier function can also be alternately written as:

```
#include <stdio.h>
main( )
{
  char ch;
  ch = getchar( );
  fflush(stdin);
  if (ch ! = '*')
    puts ("You have not entered the star
  character");
  else
     puts ("You have entered the star character");
}
```

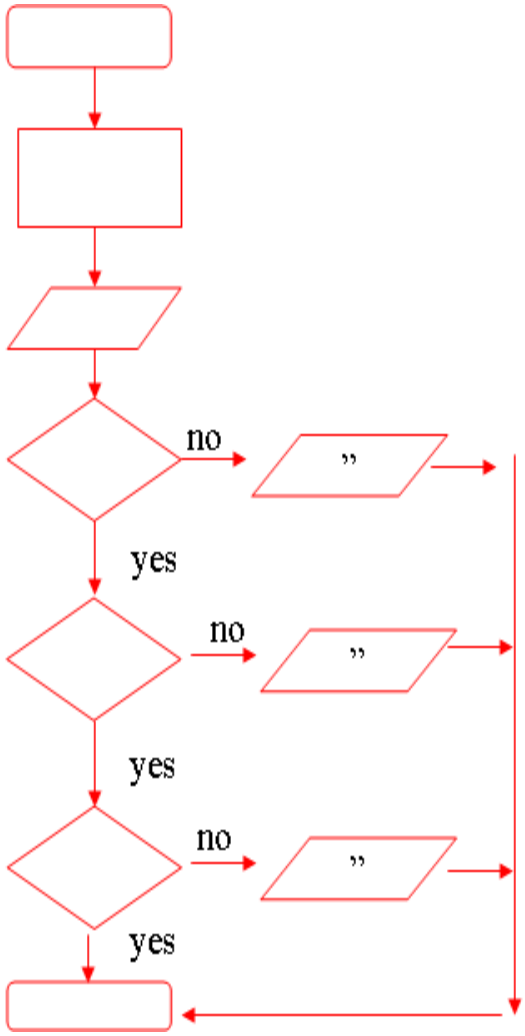
Cascading *if- else* Construct

- The cascading if-else construct is also known as the multiple if-else construct.
- On the first condition evaluating to false, the else part of the first condition consists of another if statement that is evaluated for a condition.
- If this condition evaluates to false, the else part of this if condition consists of another if statement that is evaluated for a condition, and so on.

Example - Cascading *if - else*

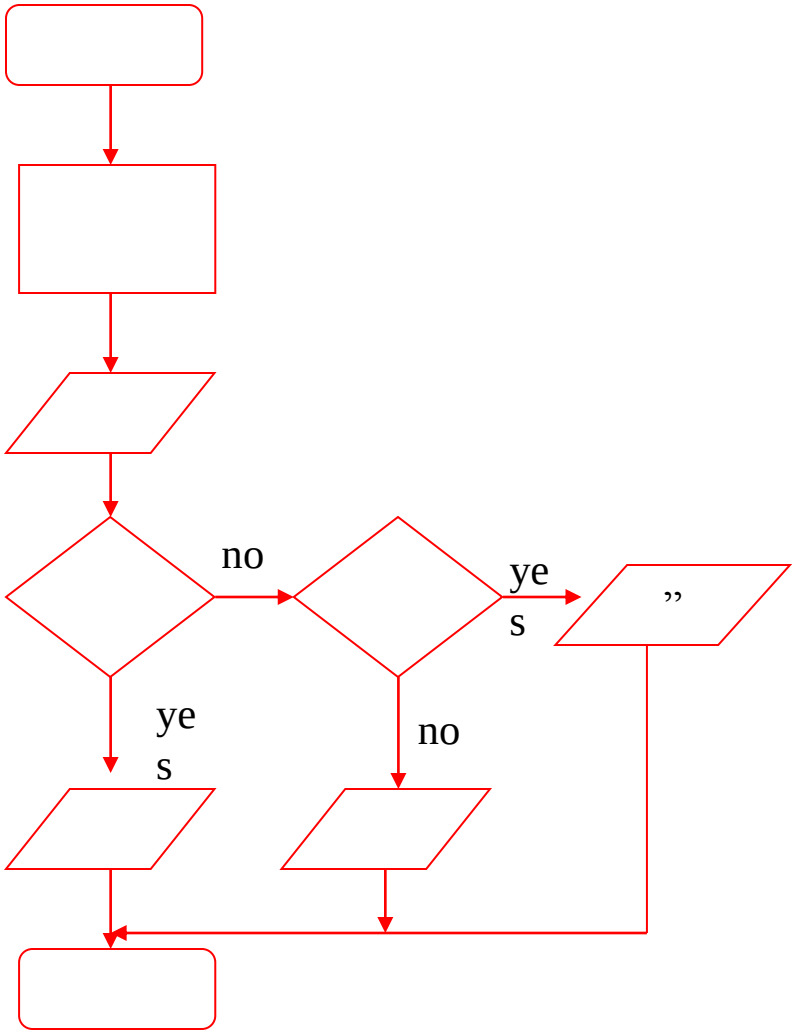
/* Function to determine for input of numbers 0 through 9 */

```
#include <stdio.h>
main( )
 { char chr;
    chr = getchar( );
    if (chr = = '0')
      puts("You entered the number 0");
    else if (chr = = '1')
      puts("You entered the number 1");
   else if (chr = = '2')
      puts("You entered the number 2");
..
    else if (chr = = '9')
         puts("You entered the number 9");
   else
     puts("You did not enter a number");
}
```



Nested *if* Construct

- A nested if statement is encountered if the statement to be executed after a condition evaluates to true is another if statement.
- Both the outer if statement and the inner if statement have to evaluate to true for the statement following the inner if condition to be executed.



Nested *if* Construct: Example

The following program determines whether the character entered is an uppercase or a lowercase alphabet.

```
#include< stdio.h>
main( )
{  char ch;
  ch = getchar( );
  fflush(stdin);
  if (ch >= 'A')
    if (ch <= 'Z')
      puts (Its an Uppercase alphabet");
    else if (ch >= 'a')
       if (inp <= 'z')
        puts ("It's a lowercase alphabet");
       else
        puts ("Input character > z");
    else
       puts ("Input character greater than Z but less than a");
  else
       puts ("Input character less than A");
}
```

switch statement

- The switch statement tests the value of a given variable(or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.
- The break statement at the end of each block signals the end of particular case and causes an exit from the switch statement, transferring the control to the statement-x.
- The default is an optional case. When present, it will be executed if the value of the expression does not match any of the case values.
- If default not present, no action takes place if all matches fail and control goes to statement-x.

```
switch (expression)
      {
          case value-1 : case-1 block
                         break;
         case value-2 : case-2 block
                         break;
              default  : default block
                         break;
       }
statement-x;
```

Example – *switch* statement

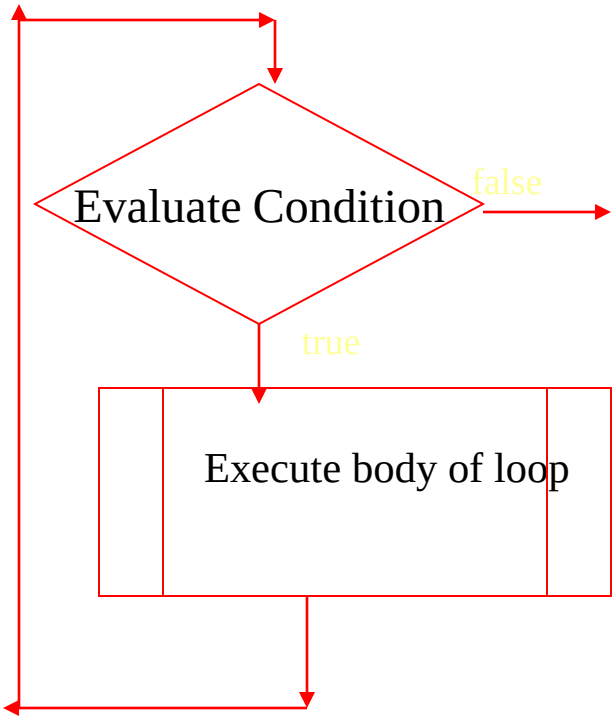
- Example :Write a program to accept a number from 1 to 7 and display the appropriate week day with 1 for Sunday.

```
#include <stdio.h>
      main()
      {
            int weekday;
            printf("\nEnter a number between 1 and 7 : ");
            scanf("%d",&weekday);
            switch(weekday)
            {  case 1 : printf("Sunday"); break;
               case 2 : printf("Monday"); break;
               case 3 : printf("Tuesday"); break;
               case 4 : printf("Wednesday");break;
        case 5 : printf("Thursday ");break;
              case 6 : printf("Friday ");break;
              case 7 : printf("Saturday");break;
              default : printf("\n Invalid option ");
          }  // end of switch case
 }   //  end of main
```

Iterative Constructs - The *while* Loop

- In a while loop, the loop condition is written at the top followed by the body of the loop.
- Therefore, the loop condition is evaluated first, and if it is true, the loop body is executed.
- After the execution of the loop body, the condition in the while statement is evaluated again. This repeats until the condition becomes false.
- General form

```
while(test-condition)
{
    body of the loop
}
```



Example - The *while* Loop

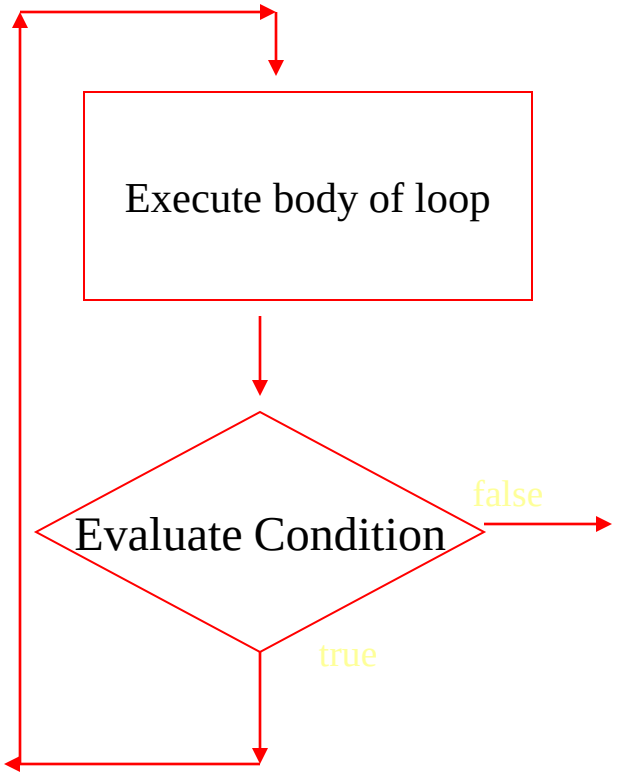
```
#include <stdio.h>
/* function to accept a string and display it 10 times
main( )
{
  int counter=0;
  char message[10];
  gets( message);
  fflush( stdin);
  while (counter <= 9)
   {
      puts( message);
      counter = counter + 1;
      gets( message);
      fflush (stdin)
    }
}
```

do…while loop

- The general form of do..while loop is

```
do
{
body of the loop
} while (test-condition)  ;
```

- The do..while construct provides an exit-controlled loop as the test-condition is evaluated at the bottom of the loop and therefore the body of the loop is always executed at least once.



Example – *do…while* loop

```
/* This is an example of a do-while loop */
main()
{
 int i;
 i = 0;
 do {
      printf("The value of i is now %d\n",i);
      i = i + 1;
     } while (i < 5);
}
```

for loop

- The for loop is another entry-controlled loop that provides a more concise loop structure

```
for(initialization;test-condition;increment)
  {
    body of the loop
  }
```

- Initialization of the control variable is done first,using assignment statements

Example – *for* loop

```
#include <stdio.h>
/* this function displays a message 10 times */
main( )
 {
   int i;
   for( i = 0; i <= 9; i = i + 1)
    {
       puts( "message");
    }
 }
```

Features of *for* loop

- Initializing, testing and incrementing are placed in the for loop itself, thus making them visible to the programmers and users, in one place.

- The for statement allows for negative increments. More than one variable can be initialized at a time.

  for (p=1,n=0;n<17;++n)

- The increment section can also have more than one part.

  for (n=1,m=50;n<=m;m++,n--)

- One or more sections can be omitted, if necessary.

  for(;m!=0;)

  ** The semicolons separating the sections must remain.

- If the test-condition is not present, the for statement sets up an infinite loop.

  for(i=0; ;i++)

Control of Loop Execution

- A loop construct, whether *while*, or *do-while*, or a *for* loop continues to iteratively execute until the loop condition evaluates to false.

- The *break* statement is used to exit early from all loop constructs (while, do-while, and for)

- The *continue* statement used in a loop causes all subsequent instructions in the loop body (coming after the continue statement) to be skipped.

break and *continue* statements

- When a break statement is encountered in a loop,the loop is exited and the program continues with the statements immediately following the loop.
- When the loops are nested,the break would only exit from the loop containing it.
- continue statement causes the loop to be continued with the next iteration after skipping any statements in between.
- In while and do loops,continue causes the control to go directly to the test-condition and then continue the iteration process.
- In case of for loop, the increment section of the loop is executed before the test-condition is evaluated

goto statement

- C supports goto statement to branch unconditionally from one point to another in a program.
- The goto requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name , and must be followed by a colon.

```
        int no,sqroot;
read:scanf("%d",&no);
        if(no<0)
        goto read;
        sqroot=sqrt(no);
```

C Statements

- Expression
- Conditional
  - if (expr) { … } else {…}
  - switch (expr) { case c1: case c2: … }
- Iteration
  - while (expr) { … } — zero or more iterations
  - do … while (expr) — at least one iteration
  - for (init ; valid ; next) { … }
- Jump
  - goto label
  - continue; — go to start of loop
  - break; — exit loop or switch
  - return expr; — return from function

Hands-on: 2 hours

Purpose

- Using all types operators
- Implementing various data types
- Using all types of conditional & iterative constructs
- Using control statement
- Performing data type conversions

Summary

In this module, we discussed:

- Classifying and using operators
- Unary, binary, ternary, compound assignment operators, increment/ decrement operators
- Rules of conversion between different data types
- Conditional constructs and iterative constructs

Arrays and Pointers

Module 3

Objectives

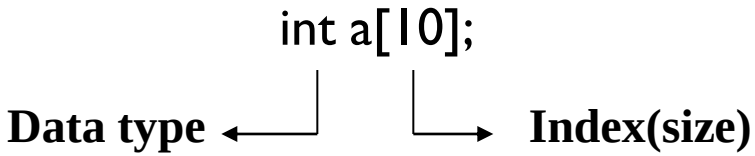
At the end of this module, you will be able to:

- Declare, initialize, manipulate, and address one-dimensional arrays
- Declare, initialize, manipulate and print from a two-dimensional array
- Declare, initialize, and manipulate pointers
- Use pointers for manipulating one-dimensional and two-dimensional arrays
- Distinguish between arrays and pointers
- Use pointer arithmetic
  - a

Duration: 4 hrs

Arrays

- An *array* is a group of related data items that share a common name.

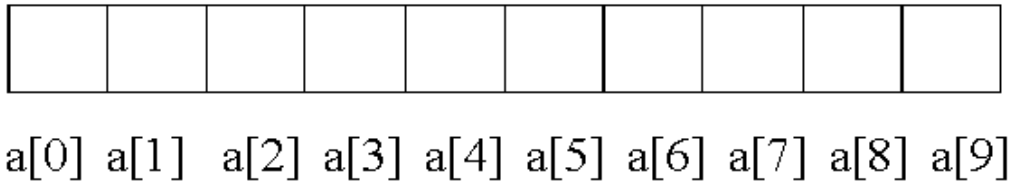


- The individual values are called *elements.*
- The elements are a[0] ,a[1] , a[2]……a[9].
- The elements are stored in *continuous* memory locations during compilation.
- The *name of the array*(a)contains the address of the first location i.e a[0].
- The elements of the array are *physically and logically adjacent.*
- When an array is passed as an argument to a function,its address is actually passed.

Array Representation

- `int a[10];`

  /*defines an array a of size 10, as a block of 10 contiguous elements in memory */

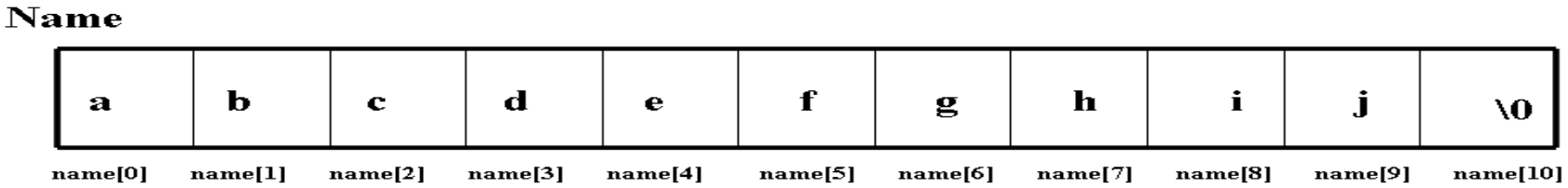


- Character Arrays

  `Char name[11];`

  – To define a character array, need to define a array of size n+1 characters. This is because all character arrays are terminated by a NULL character ('\0')

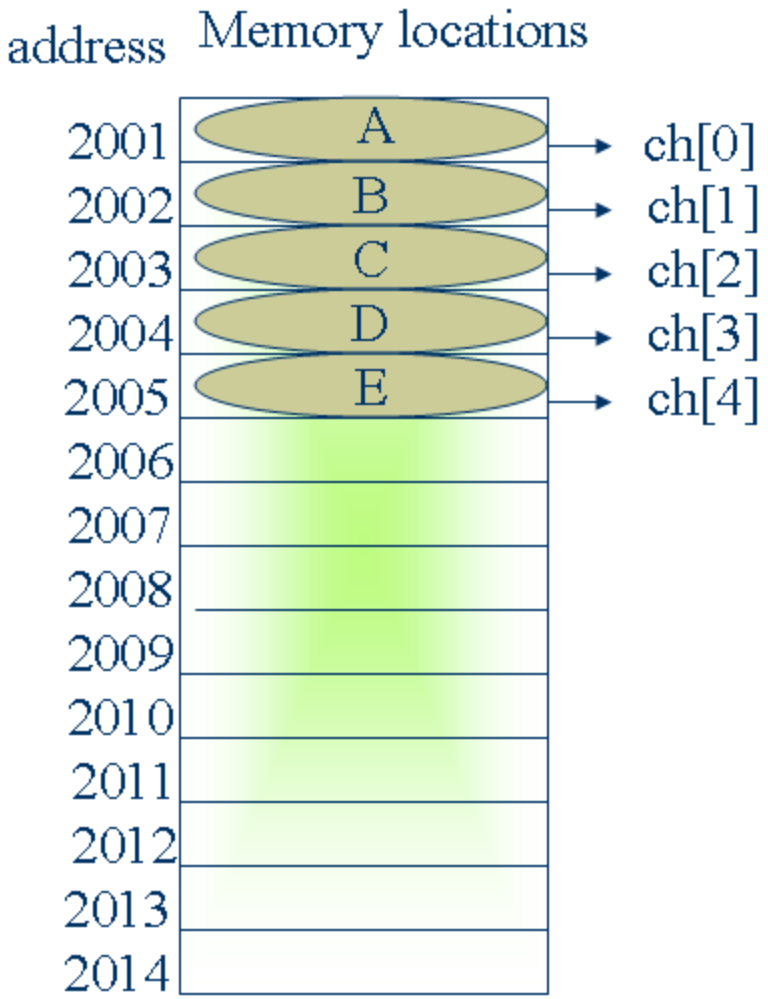
  – where name[0] through name[9] will contain the characters comprising the name, and name[10] will store the NULL character



Array - Memory allocation

```
char ch[5]={ 'A','B','C','D','E'};
      or
ch[0]='A';
ch[1]='B';
ch[2]='C';
ch[3]='D';
ch[4]='E';
```

Note: Memory allocation is done by OS.



Array Initialization

- Since an array is a set of elements located at contiguous memory locations, its initialization involves moving element by element, or one data at a time into the array.

```
#include<stdio.h>
main( )
{
  char array1[ ] = {'A', 'R', 'R', 'A', 'Y', '\0'};
  char array2[ ] = {"ARRAY"};
   char dayofweek[ ] = {'M', 'T', 'W', 'T', 'F', 'S', 'S', '\0'};
  float values[ ] = {100.56, 200.33, 220.44,0};
}
```

Array Processing

```
#include<stdio.h>
main( )
{  char array1[ ] = {'A', 'R', 'R', 'A', 'Y', '\0'};
   char array2[ ] = {"ARRAY"};
   char dayofweek[ ] = {'M', 'T', 'W', 'T', 'F', 'S', 'S', '\0'};
   float values[ ] = {100.56, 200.33, 220.44, 400.22, 0};
   int i = 0;

    printf( "String 1 is %s\n", array1);
    printf( "String 2 is %s\n", array2);

    for( i = 0; dayofweek[i] != '\0'; i = i +1)
   printf ( "The Day %d in a week is %c\n", i + 1, dayofweek[i];

    for( i = 0; values[i] != 0; i = i +1)
   printf ( "The amount %d in a week is %f\n", i + 1, values[i];
}
```

Array Manipulation Using Subscripts

```
/* this function finds the length of a character string */
#include <stdio.h>
main( )
{
   int i = 0;
   char string[11];

   printf("Enter a string of maximum ten characters\n");
   gets(string);
   fflush( stdin);

   for(i =0; string[i] != '\0'; i = i + 1);
   printf("The length of the string is %d \n", i);
}
```

Example - Array Manipulation Using Subscripts

```
/* this function converts a string to upper case */
main( )
{
  char string[51];
  int i = 0;
  printf("Enter a string of maximum 50 characters\n");
  gets(string);
  fflush(stdin);
  while (string[i] != '\0')
   {
     if(string[i] >= 'a' && string[i] <= 'z')
      {
        string[i] = string[i] - 32;
        i = i + 1;
      }
   printf("The converted string is %s\n", string)
 }
```

Example - Array Manipulation Using Subscripts (Contd.).

- /* function to extract a substring from a main string starting at a specified position, and containing a specified number of characters */

```
main( )
{
  int i, start_pos, no_of_chars;
  char string[101], substring[101];
  printf("Enter a string of upto 100 characters\n");
  gets(string);
  fflush( stdin);
  printf("Enter start position, and no. of characters to
   extract\n");
  scanf("%d,%d", &start_pos, &no_of_chars);
  fflush(stdin);
  start_pos = start_pos -1;
  for (i = 0; i < no_of_chars; i = i + 1, start_pos =
   start_pos + 1)
   {
     substring[i] = string[sp];
    }
   substring[i] = '\0';
   printf("The substring is %s\n", substring);
 }
```

Example - Array Manipulation Using Subscripts

/* the following function accepts numbers as long as 0 is not entered as an array element, and displays the sum of the numbers in the array along with the array elements */

```
main( )
{
  int total, int_array[20], i = -1;
  do
   {
     i = i + 1;
     printf("Enter number %d (0 to terminate)\n");
     scanf("%d", &int_array[i]);
    }while (int_array[i] != 0);
   i = 0;
   while (int_array != 0)
   {
     printf("Element number  %d is %d\n", i + 1, int_array[i]);
     total = total + int_array[i];
    }
  printf("The sum of the numbers in the array is %d \n", total);
 }
```

Two-Dimensional Arrays

- **Declaration:**

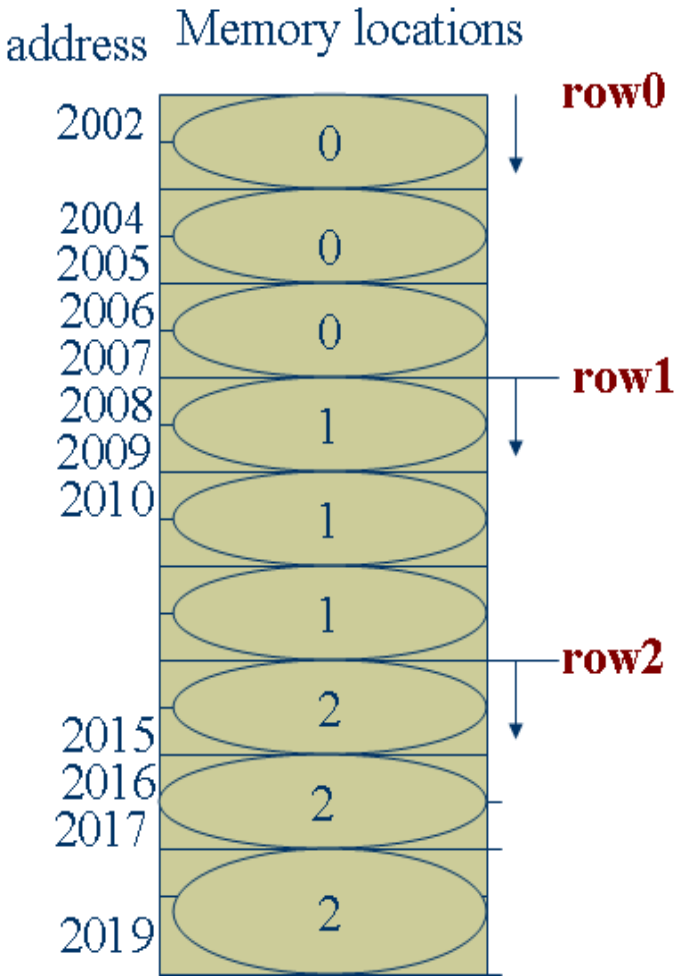
type array_name[rowsize][columnsize];

Eg:

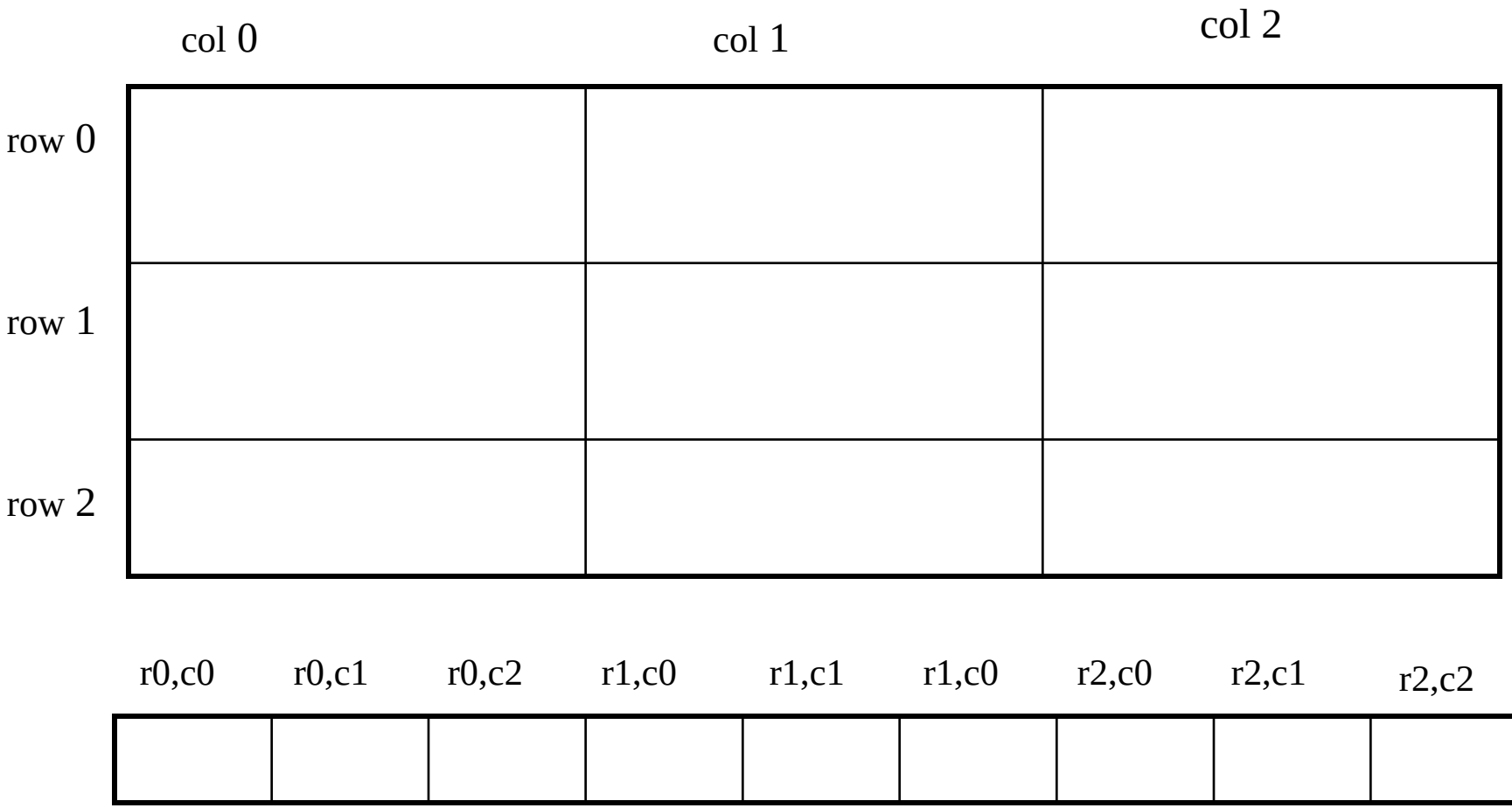
```
int m[3][3]={0,0,0,1,1,1,2,2,2};
```

m[0][0] m[0][1] m[0][2]
m[1][0] m[1][1] m[1][2]
m[2][0] m[2][1] m[2][2]

In multi-dimensional arrays,elements are stored *sequentially row wise* in memory.



Two-dimensional Arrays (Contd.).



Two-Dimensional Array – Declaration, Initialization

- Declaring a two-dimensional array involves two indices, or two subscripts. There will be an extra set of [ ] to indicate the second subscript, or the second index.

```
int rp_array[3][3] = { 0,0,0,
                       0,0,0,
                       0,0,0
                     };
```

- To improve the legibility of the initialization, it can be written as:

```
int rp_array[3][3] = { {0,0, 0},
                       {0, 0, 0},
                       {0, 0, 0}
                     };
```

Processing Two-Dimensional Arrays

/* Program for converting these sales figures into percentages of total sales. */

```
main( )
{  int r_counter, p_counter, rp_array[3][3], total_sales = 0;
    float rp_array_perc[3][3];

/* input sales into rp_array using the for loop */
  for (r_counter = 0; r_counter < 3; r_counter ++)
    {  for (p_counter = 0; p_counter < 3; p_counter ++)
        { printf( "\nEnter sales data for Region %d and Product %d, r_counter + 1, p_counter + 1);
          scanf("%d", &rp_array[r_counter][p_counter]);
          fflush( stdin);
        }
     }

/* Determine total  sales using the for loop */
for (r_counter = 0; r_counter < 3; r_counter ++)
 {  for (p_counter = 0; p_counter < 3; p_counter ++)
          total_sales += rp_array[r_counter][p_counter];
  }
```

Processing Two-Dimensional Arrays (Contd.).

```
/* Determine percentage of individual sales data against total sales */
for (r_counter = 0; r_counter < 3; r_counter ++)
 { for (p_counter = 0; p_counter < 3; p_counter ++)
     rp_array_perc[r_counter][p_counter] =
        (( float)100 * rp_array[r_counter][p_counter] ) / total_sales ;
 }
} /* end of main( ) */
```

The Preprocessor Phase

- The measure of the flexibility of any program is the ease with which changes can be implemented.
- The preceding exercises involving two-dimensional arrays is inflexible because the number of regions and products is hard coded into the program, causing the size of the array to be also hard coded.
- If changes in terms of the number of regions and products were to be incorporated with the least amount of code maintenance, the best solution would be the use of macros referred to in C as #define.

The Preprocessor Phase (Contd.).

- The #define constitutes the preprocessor phase, wherein the value/s specified as part of the macro or #define is substituted into the code prior to compilation. This is also known as macro substitution.

```
#include <stdio.h>
#define RG 3
#define PR 3
main( )
{  int r_counter, p_counter, rp_array[RG][PR], total_sales = 0;
   float rp_array_perc[RG][PR];

 /* initialization of rp_array using the for loop */
  for (r_counter = 0; r_counter < RG; r_counter ++)
  { for (p_counter = 0; p_counter < PR; p_counter ++)
       rp_array[r_counter][p_counter] = 0;
  }
```

Two-Dimensional Character Arrays

- If you were to store an array of 11 names, with each name containing up to a maximum of 30 characters, you would declare it as a two-dimensional character array such as: char name[11][31];

- Here, name[0] through name[9] would store character strings representing names of 30 characters each.

- The first index represents the number of names, and the second index represents the maximum size of each name.

```
main( )
{ char team_india [11][30] = {
                "Akash Chopra",
                "Virendra Sehwag",
                "Rahul Dravid"
                "Sachin Tendulkar",
                "V.V.S. Laxman",
                "Yuvraj Singh",
                "Ajit Agarkar",
                "Parthiv Patel",
                "Anil Kumble",
                "L. Balaji",
                "Irfan Pathan"
                };
  int i;
  for( i = 0; i < 11; i ++)
    printf("%s", team_india[i]);
  }
```

Pointers

- A pointer is a variable which holds the *address of another variable* in memory.The actual location of a variable in the memory is system dependent.
- Since *pointer is a variable* ,its value is also stored in the memory in another location.
- **&** operator(address of) is used to assign the address of a variable to a pointer variable.
- The declaration of a pointer takes the following form:

```
    data type *ptr;
Eg: int *pa;
    int a;
    pa=&a;
```

  - ✓ The asterisk(*)tells that pa is a pointer variable.
  - ✓ pa needs a memory location.
  - ✓ pa points to a variable a of type int

Accessing a variable through its pointer

- A pointer contains garbage until it is initialized
- The value of a variable can be accessed using pointer by *(asterisk) known as *indirection* operator.

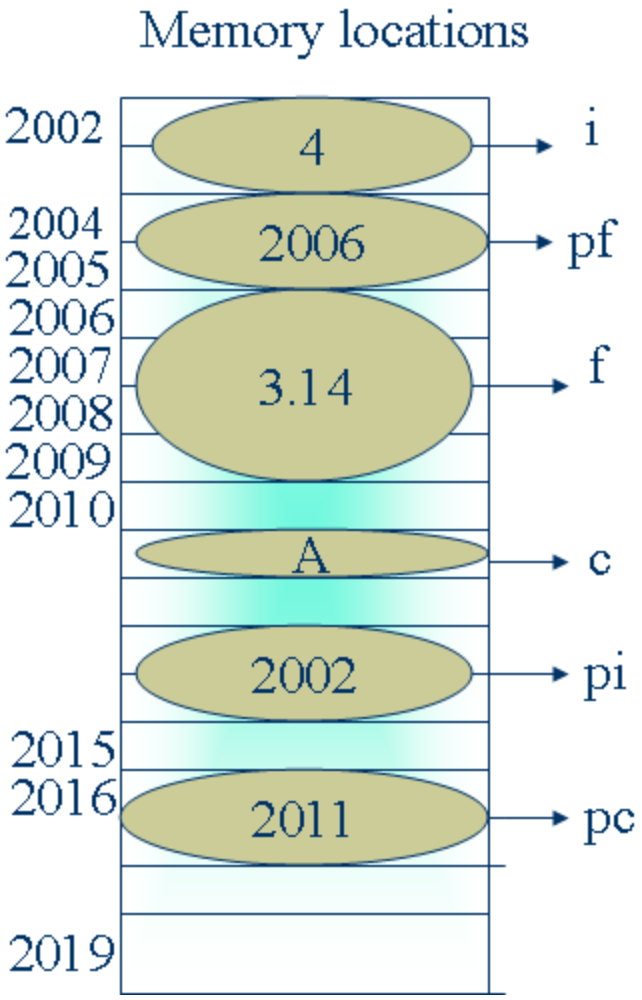
Eg:
```
int a,n;
  int *pa;
  a=4;
  n=*pa;
```

The value of **a** i.e 4 is assigned to n through its pointer **pa**.

*When the operator * is placed before a pointer variable in an expression,the pointer returns the value of the variable of which the pointer value is the address*.

Pointer - Memory Allocation

```
int i,*pi;
float f,*pf ;
char c,*pc;
i=4;
f=3.14;
c='A'
pi=&i;
pf=&f;
pc=&c;
```



The actual location of a variable in the memory is OS dependent.

*Two bytes(in **DOS**)of memory is allocated to any type of pointer.*

Pointer – Declaration & Initialization

- **It is in this sense that a pointer is referred to as a derived data type, as the type of the pointer is based on the type of the data type it is pointing to.**

- For the earlier declaration of the integer variable i, its pointer can be declared as follows:

```
int i, *pointer_to_an_integer;
i =  22;
pointer_to_an_integer = &i;
```

/* initialize pointer variable (pointer_to_an_integer) with address of the integer variable i. */

Dereferencing a Pointer

- Returning the value pointed to by a pointer is known as pointer dereferencing. To deference the contents of a pointer, the * operator is used.

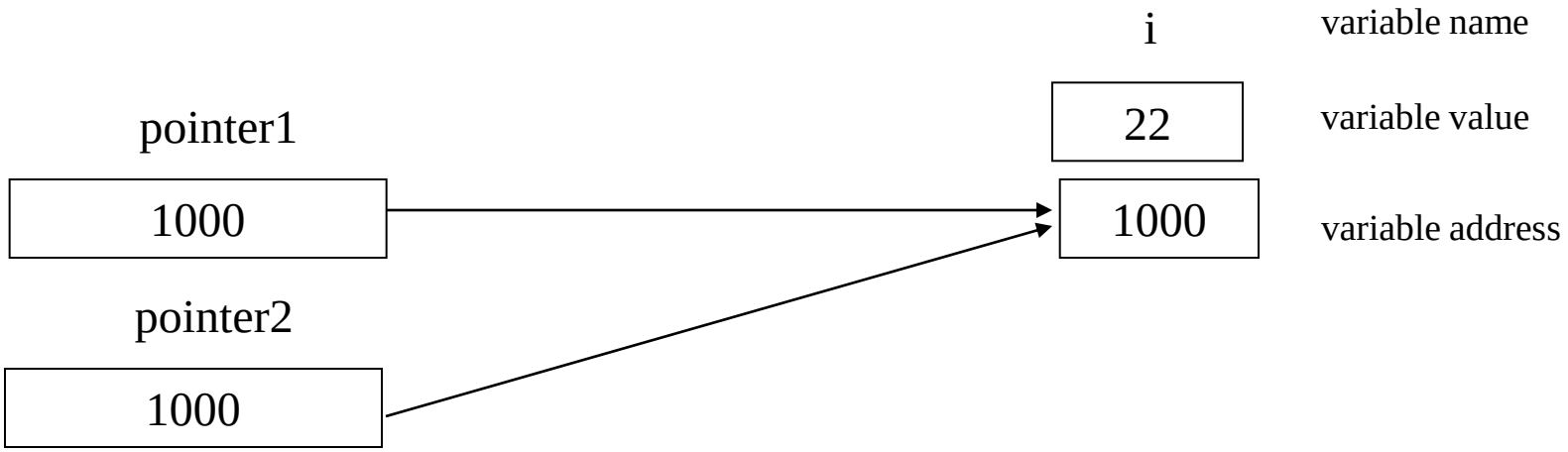
```
#include<stdio.h>
main( )
{ int x, y, *pointer;
  x = 22;
  pointer = &x;
  y = *pointer; /* obtain whatever pointer is pointing to */
}
```

Pointers - Variations on a Theme

- Consider the following code snippet:

```
int x, y, *pointer1, *pointer2;
pointer1 = &x   /* return the address of x into pointer1 */
pointer2 = pointer1;
```

/* assign the address in pointer1 to pointer2. Hence, both the pointer variables, pointer1 and pointer2, point to the variable x. */



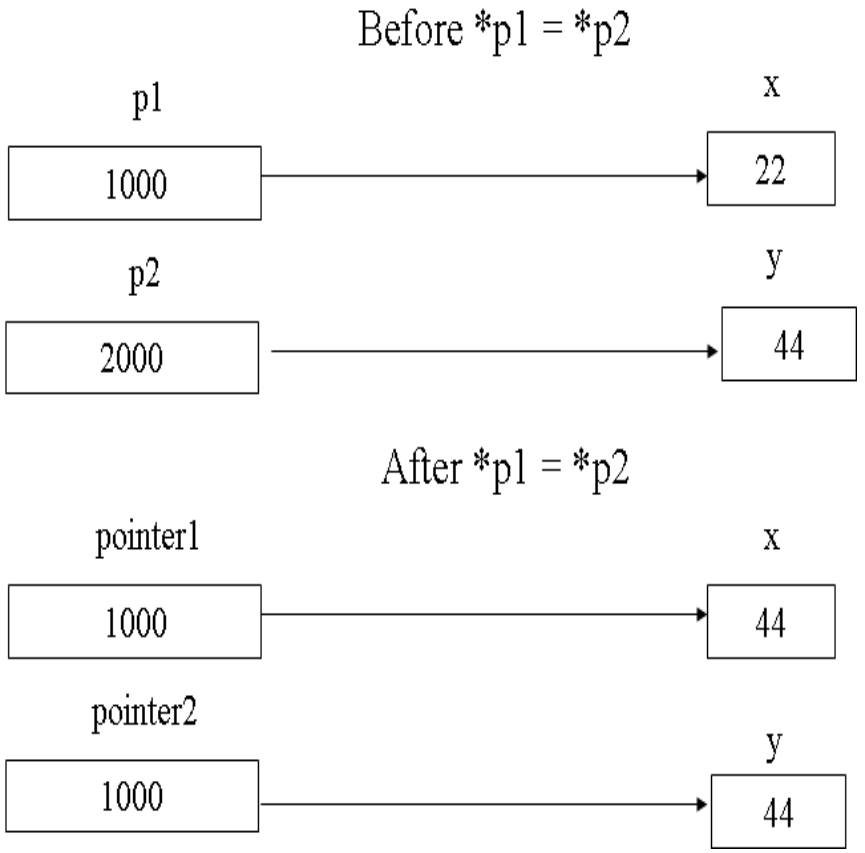
Pointers - Variations on a Theme (Contd.).

- Consider the following code snippet:

```
int x, y, *p1, *p2;
x = 22;
y = 44;
p1 = &x;  /* p1 points to x */
p2 = &y ;  /* p2 points to y */
*p1 = *p2 ;
```

/* make whatever p1 was pointing to (variable x and hence the value 22) equivalent to whatever p2 is pointing to. */

- Hence, both x and y will now have the value 44.



Printing Using Pointers

- Consider the following code snippet:

```
#include <stdio.h>
main( )
{
  int x, *p;
  x = 26;
  p = &x;
  printf("The value of x is %d\n", x);
  printf("The value of x is %d\n", *p);
 }
```

Pointer increments and scale factor

```
int *pa;
float *pf;
char *pc;
pi++;
pf++;
pc++;
```

When a pointer is incremented ,its value is increased by the length of the data type that its points to.This length is called *scale factor.*

The no of bytes used to store various data types can be found by using **sizeof** operator.

Pointers and Arrays

- Consider the following declaration:
  - char string[10], *p;
    - Both **string** and **p** are pointers to **char**

- However, string[10] has 10 bytes of contiguous storage allocated for it.

- Thus, during execution, string is effectively a pointer with a constant address, namely, the address &string[0]; and this address cannot be changed during the life of the program

Character Arrays Using Pointers

One-dimensional

- The one-dimensional character array declared earlier (char string[11]) can be alternately declared as:

```
char *string; /* string is now a explicit pointer variable that can point to a character */
char *string = "Hello";
printf("%s", string);
```

Two-dimensional

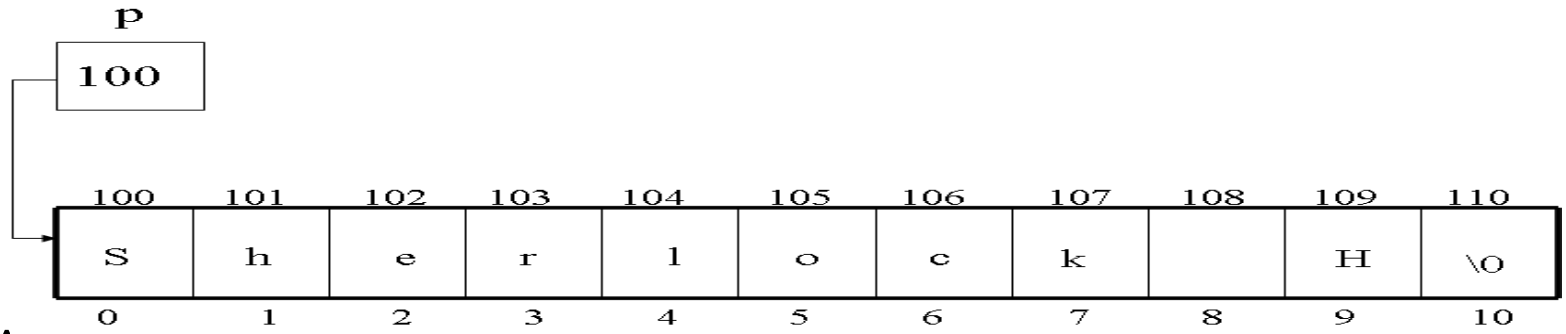
- Since a pointer to a character can be a pointer to a string, it follows that a two-dimensional character array can be declared as an **array of character pointers.**

- This could be alternately declared as:
  char *team_india[11];

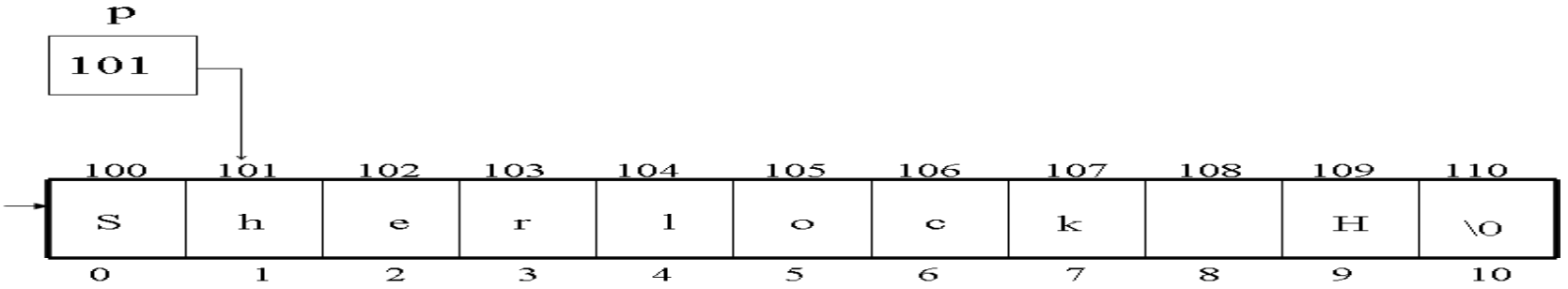
Pointer Arithmetic

```
char *p = "Sherlock H";
printf("%s\n", p);
```

- The initialization of the character pointer with the string "Sherlock H" can be visualized as shown in the following slide.



- After incrementing the pointer p by 1, it points to the next element in the string or the character array, i.e., character 'h' after 'S'. p now contains the address of the element 'h', i.e., 101



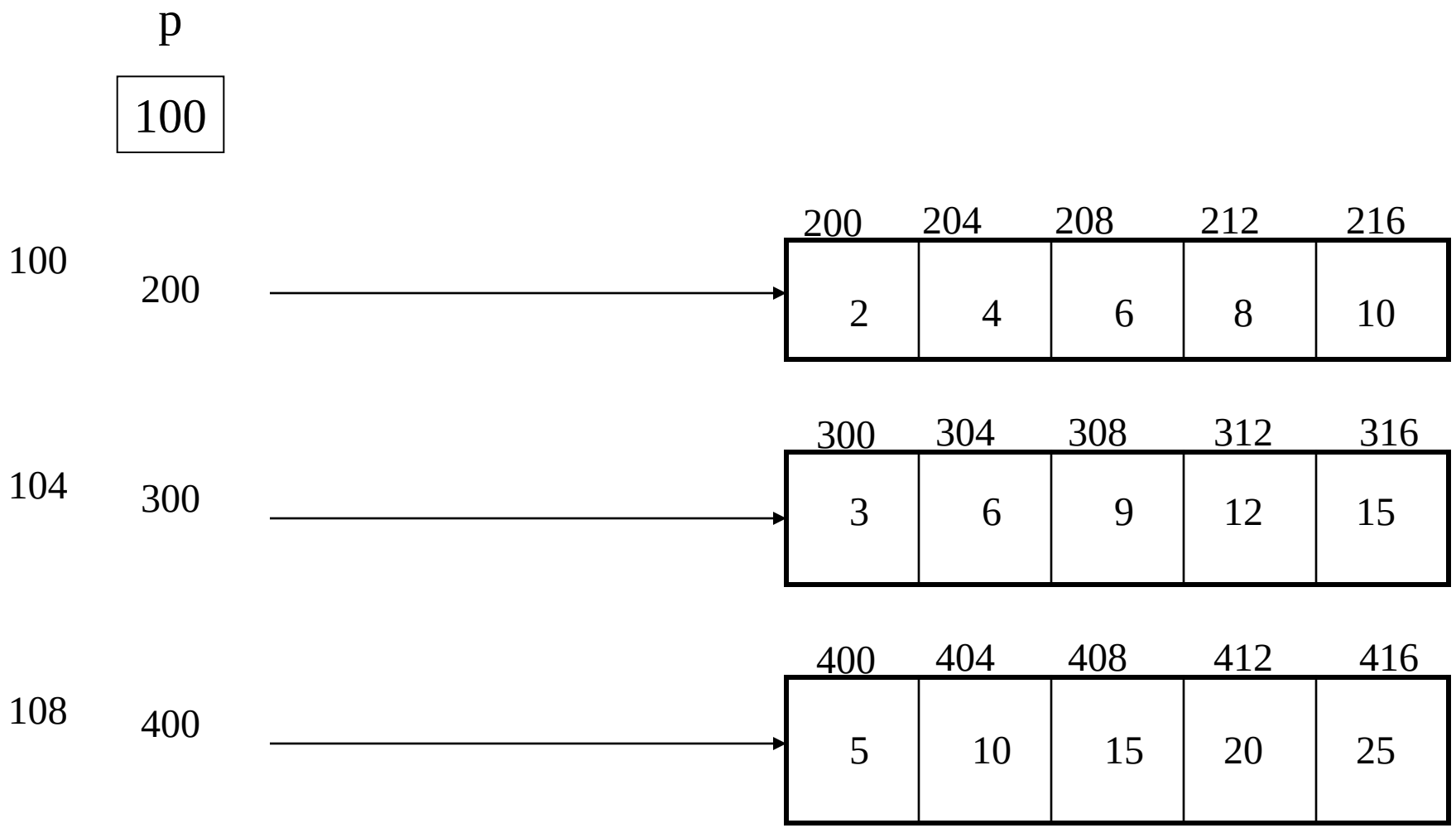
Pointer Arithmetic (Contd.).

- Consider the following declaration of a two-dimensional integer array:

```
int p[3][5] = {{ 2, 4, 6, 8, 10},
               { 3, 6, 9, 12, 15},
               { 5, 10, 15, 20, 25}
              };
```

- The aforesaid declaration declares an array of three integer pointers, each pointing to the first element of an array of 5 integers. This can be visualized as follows:

Pointer Arithmetic (Contd.).



Pointer Arithmetic (Contd.).

- Hence, *(p + 1) returns the address 300.
- Therefore, * (*(p + 1)) returns the value at this address, i.e., the element with the value 3. In other words, the element at the offset [1,0].
- The following table gives various pointer expressions, and their values:

Pointer Expression	Resulting Address	Variable	Value
*(*p)	200	p[0][0]	2
*(*p+1)	204	p[0][1]	4
((p + 1))	300	p[1][0]	3
((p+1)+1)	304	p[1][1]	6
((p+1)+1)+1	304	p[1][1] + 1	6 + 1 = 7

String Handling Functions Using Pointers

/* The following function determines the length of a string */

```
#include <stdio.h>
main( )
{
  char *message = “Virtue Alone Ennobles”;
  char *p;
  int count;
  for (p =  message, count = 0, p != ‘\0’, p++)
   count++;
 printf(The length of the string is %d\n”, count);
}
```

String Handling Functions Using Pointers (Contd.).

/* The following functions compares two strings */

```
#include<stdio.h>
main( )
{
 char *message1 = "This is a test";
 char *message2 = "This is not a test";
 char *p1, *p2;
 for(p1=message1, p2=message2; (*p1 = = *p2) && (*p1
  != '\0') && (*p2 != '\0'); p1++, p2++)
if ((*p1 = = '\0') && (*p2 = = '\0'))
  printf("The two strings are identical\n");
else
  printf("The two strings are not identical\n");
}
```

Hands-on: 2 hours

Purpose

- Construct single and 2 dimensional Arrays
- Access & computing array elements
- Use pointers for any variable

Summary

In this module, we discussed:

- One-dimensional arrays - Declare, initialize and manipulate
- Two-dimensional array - - Declare, initialize and manipulate
- Pointers - Declare, initialize, and manipulate
- pointers for manipulating one-dimensional and two-dimensional arrays
- Use pointer arithmetic

Functions & Recursion

Module 4

Objectives

At the end of this module, you will be able to:

- Write programs that invoke functions through a:
  - Call by value
  - Call by reference
- Define function prototypes
- Describe the function call mechanism
- Use the Auto, Static, and Extern storage qualifiers
- Use string handling functions, conversion functions, and functions for formatting strings in memory
- Describe how to write a recursive function
- Describe recursive calls to a function using the runtime stack
- Define, Declare, and Use Function Pointers

Duration: 4 hrs

Function - Introduction

- Functions facilitate the factoring of code.

- Functions facilitate:
  - Reusability
  - Procedural abstraction

- Function Parameters
  - Function parameters are defined as part of a function header, and are specified inside the parentheses of the function.
  - The reason that functions are designed to accept parameters is that you can embody generic code inside the function.
  - All that would be required by the function would be the values that it would need to apply the generic code on the values received by it through its parameters

Function Parameters

- Consider the following printf( ) statement:

  printf("%d", i);

- This is actually a call to the printf( ) function with two pieces of information passed to it, namely, the format string that controls the output of the data, and the variable that is to be output.

- The format string, and the variable name can be called the parameters to the function printf( ) in this example.

Invoking Functions

- In C, functions that have parameters are invoked in one of two ways:
  - Call by value
  - Call by reference

Call by Value

```
void swap(int,int );
 main()
     { int a=10, b=20;
       swap(a, b);
       printf(" %d %d \n",a,b);
     }

void swap (int x, int y)
    {   int temp = x;
        x=  y;
        y=temp;
    }
```

Call by reference

```
void swap( int *, int *);
 main()
     { int a=10, b=20;
       swap(&a, &b);
       printf(" %d %d \n",a,b);
     }

void swap (int *x, int *y)
    {    int temp=*x;
         *x=*y;
         *y=temp;
    }
```

Passing Arrays to Functions

- Arrays are inherently passed to functions through a call by reference
  - For example, if an integer array named **int_array** of 10 elements is to be passed to a function called **fn( )**, then it would be passed as:

```
fn( int_array);
```

- Recall that int_array is actually the address of the first element of the array, i.e., &int_array[0]. Therefore, this would be a call by reference.

- The parameter of the called function **fn( )** would can be defined in one of three ways:

```
fn( int num_list[ ]); or
fn(int num_list[10]); or
fn(int *num_list)
```

Returning a Value From a Function

- Just as data can be passed to a function through the function's parameters at the time of call, the function can return a value back to the caller of the function.

- In C, functions can return a value through the return statement.

```
#include<stdio.h>
main( )
{ int i,j, value;
   scanf("%d %d", &i, &j);
   fflush(stdin);
  value = add(i, j);
  printf( "the total is %d\n", value);
}
add( int a, int b)
{ return (a + b);
}
```

Function Prototype

- C assumes that a function returns a default value of **int** if the function does not specify a return type
  - In case a function has to return a value that is not an integer, then the function itself has to be defined of the specific data type it returns.

- Functions should be declared before they are used.

- Consider the situation where you want to use the **pow( )** function, called the power function, one of many functions in the mathematics library available for use by the programmer.
  - A function call such as **pow(x, y)** returns the value of x raised to the power y.

Function Prototype (Contd.).

- To elucidate further, pow(2.0, 3.0) yields the value 8.0
  - The declaration of the function is given by:

    double pow(double x, double y);

- Function declarations of this type are called function prototypes.
  - An equal function prototype is given by:

    double pow(double, double);

- A function prototype tells the compiler the number and data types of arguments to be passed to the function and the data type of the value that is to be returned by the function.

- ANSI C has added the concept of function prototypes to the C language.

Function Prototype (Contd.).

```
#include <stdio.h>
float add(float a, float b);
main()
{  float i, j, value;
   scanf("%f %f", &i, &j);
   fflush(stdin);

   value = add(i, j);
   printf( "the total is %d\n", value);
}

float add(float a, float b)
{
  return (a + b);
}
```

```
#include <stdio.h>
main( )
{  void add( float, float);
   float i, j, value;

   scanf("%f %f", &i, &j);
   fflush(stdin);

   add(i, j);
   printf( "the total is %d\n", value);
}

void add(float a, float b)
{
  printf("%f", a + b);
  return;
}
```

Function Calls

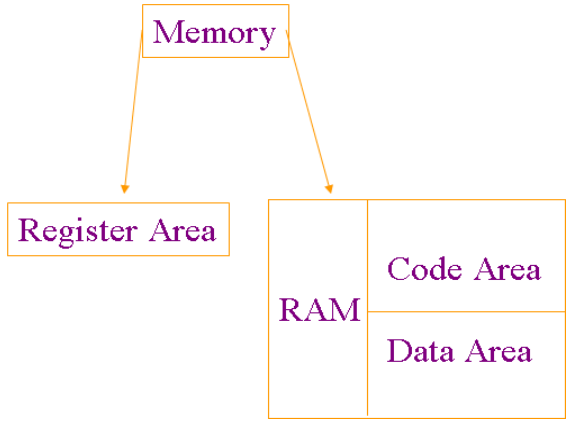
- It is important for us to know what happens under the hood when a function executes.
- A logical extension of the aforesaid point is the situation that arises when a function calls another function.
- It is important to know how the CPU manages all this, i.e., knowing where to look for when a function call is encountered, and having executed the function, to also know where it has to return to.
- In short, we need to know the call mechanism, and the return mechanism.

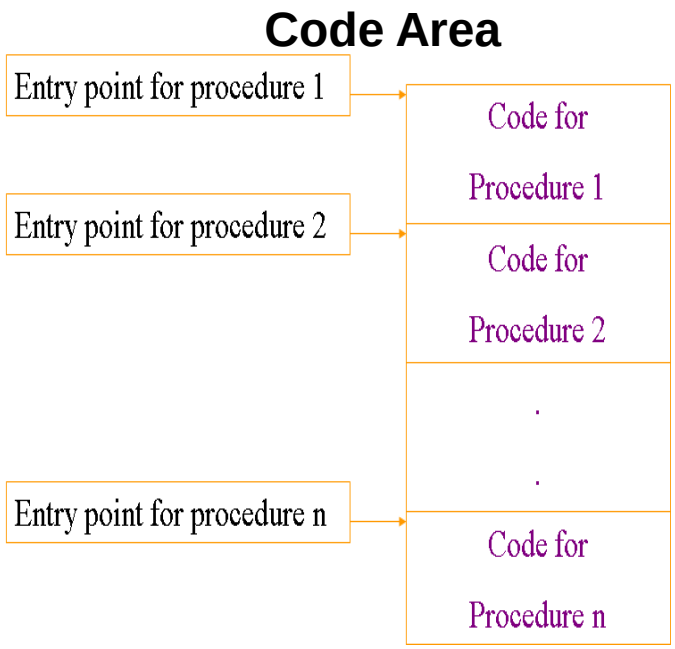
Function Calls & The Runtime Stack

- **Runtime Environment:** Runtime Environment is the structure of the target computer's registers and memory that serves to manage memory and maintain the information needed to guide the execution process.

- The C language uses a stack-based runtime environment, which is also referred to as a runtime stack, or a call stack.

- Let us begin by understanding the internal memory organization that comes into the picture whenever a program needs to be executed.





Data Area

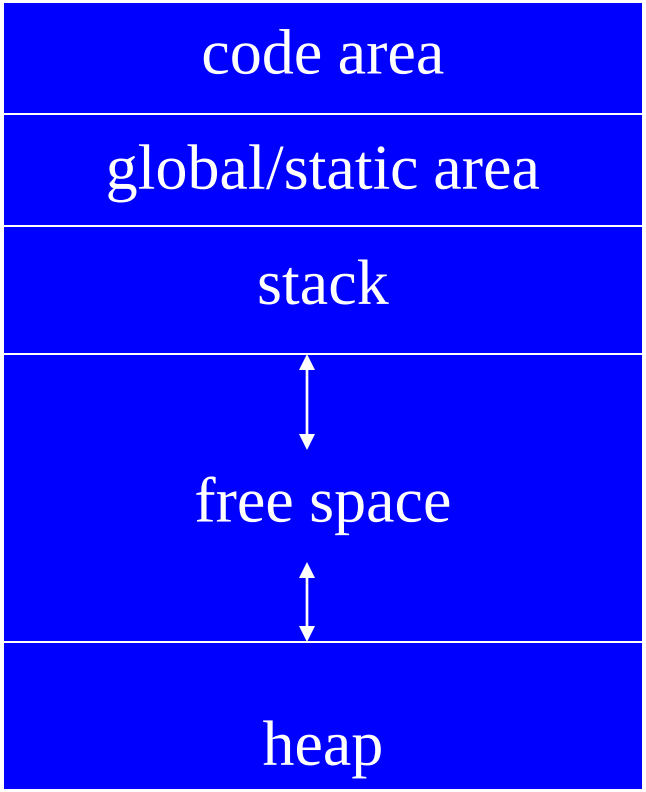
- small part of data can be assigned fixed locations before execution begins
  - Global and/or static data
  - Compile-time constants
    - Large integer values
    - Floating-point values
  - Literal strings

Dynamic Memory

- The memory area for the allocation of dynamic data can be organized in many different ways.

- A typical organization divides the dynamic memory into
  - stack area (LIFO protocol)
  - heap area

Memory Organization

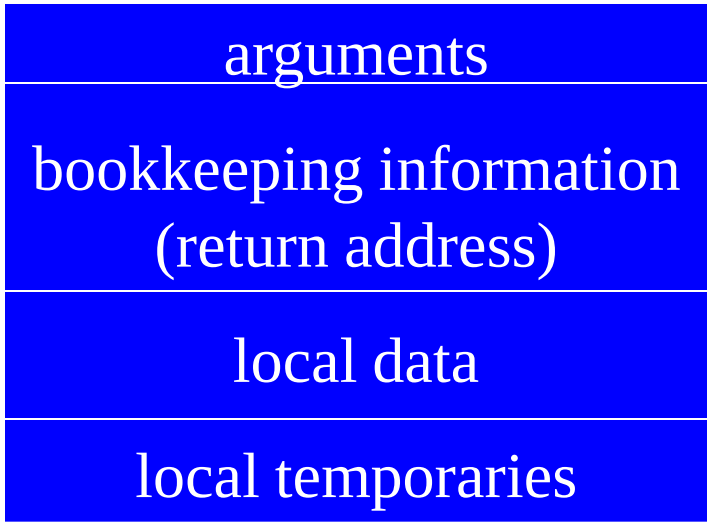


Procedure Activation Record

- Procedure activation record contains memory allocated for the local data of a procedure or function when it is called, or activated.

- When activation records are kept on stack, they are called stack frames. Details depend on the architecture of target machine and properties of the language.

A Procedure Activation Record or a Stack Frame



Registers

- Registers may be used to store temporaries, local variables, or even global variables.
- When a processor has many registers, the entire static area and whole activation records may be kept in the registers.
- Special purpose registers:
  - Program counter (PC)
  - Stack pointer (SP)

Calling Sequence

- The calling sequence is the sequence of operations that must occur when a procedure or function is called.
  - Allocation of memory for the activation record
  - The computation and storing the arguments
  - Storing and setting registers

1. Compute the arguments and store them in their correct positions in the new activation record (pushing them in order onto the runtime stack)
2. Store (push) the fp as the control link in the new activation record.
3. Change the fp so that it points to the beginning of the new activation record (fp=sp)
4. Store the return address in the new activation record.
5. Jump to the code of the procedure to be called

Return Sequence

- The return sequence is the sequence of operations needed when a procedure or function returns.
  - The placing of the return value where it can be accessed by the caller
  - Readjustment of registers
  - Releasing of activation record memory

1. Copy the fp to the sp
2. Load the control link into the fp
3. Jump to the return address
4. Change the sp to pop the arguments.

Fully Static Runtime Environment

- The simplest kind of a runtime environment.
- All data are static, remaining in memory for the duration of the program.
  - All variables can be accessed directly via fixed addresses
- Each procedure has only a single activation record, which is allocated statically prior to execution.
- Such environment can be used to implement a language in which:
  - There are no pointers or dynamic allocation,
  - Procedures cannot be called recursively.
- Example: COBOL & FORTRAN

Stack-based Runtime Environment

- In a language in which recursive calls are allowed, activation records cannot be allocated statically.
- Activation records are allocated in a stack-based fashion.
- This stack is called the stack of activation records (runtime stack, call stack).
- Each procedure may have several different activation records at one time.

Global Procedures

- In a language where all procedures are global (the C language), a stack-based environment requires two things:
  - A pointer to the current activation record to allow access to local variables.
    - This pointer is called the frame pointer (fp) and is usually kept in a register.
  - The position or size of the caller's activation record
    - This information is commonly kept in the current activation record as a pointer to the previous activation record and referred as the control link or dynamic link.
    - Sometimes, the pointer is called the old fp
  - Additionally, there may be a stack pointer (sp)
    - It always points to the top of the stack

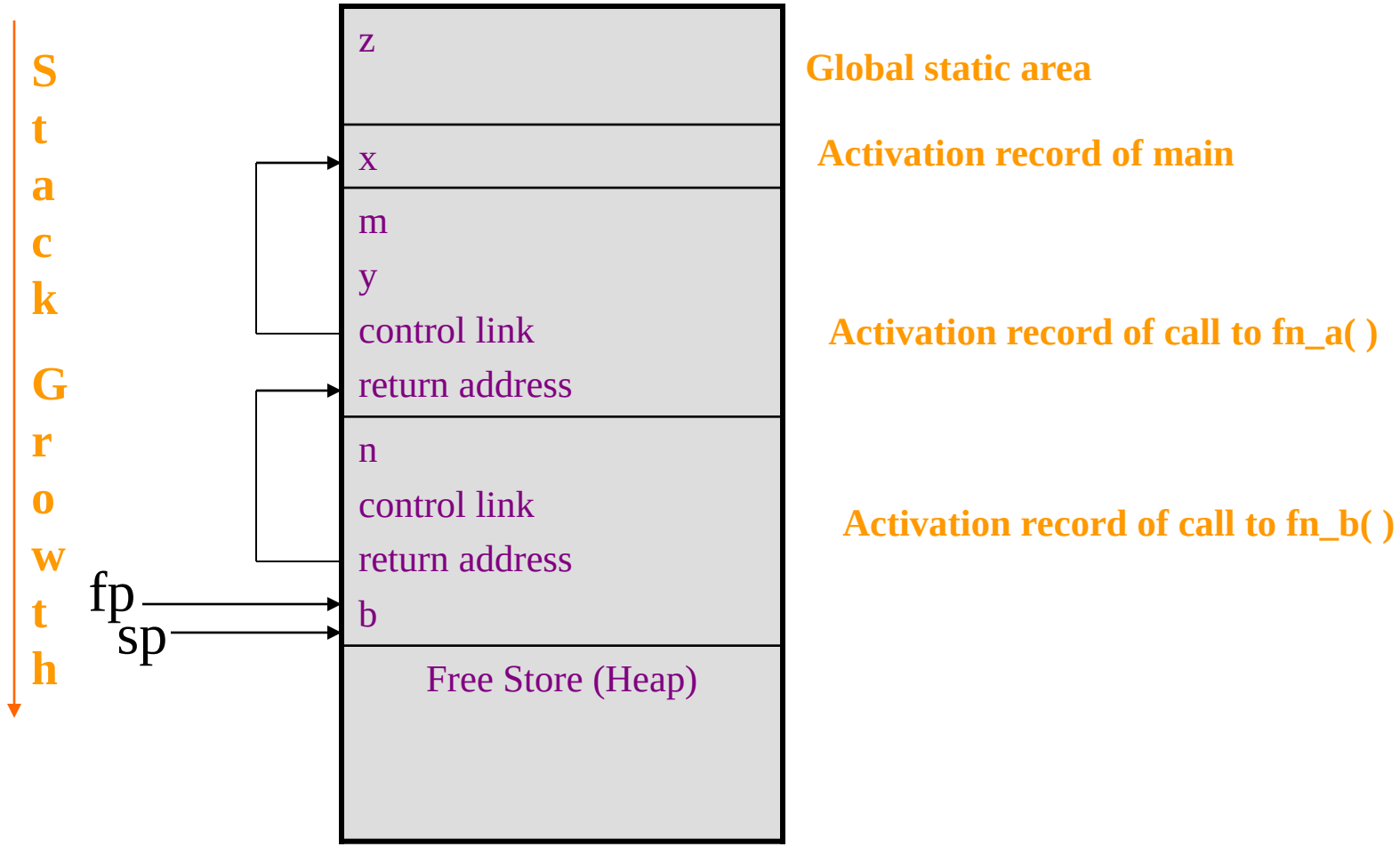
Tracing Function Calls

```
int z;
main( )
{
  int x;
  fn_a( );
  ……. ←—— return instruction
.
.
.
}
```

```
fn_a( int m )
{
 int y;
 fn_b( );
 ……. ←—— return instruction
.
.
.
}
```

```
fn_b( int n )
{
 int b;
  …..
.
.
.
}
```

A View of the Runtime Stack



Access to Variables

- In static environment, parameters and local variables can be accessed by fixed addresses.
- In a stack-based environment, they must be found by offset from the current frame pointer.
- In most languages, the offset for each local variable is still statically computable by compiler.
  - The declarations of a procedure are fixed at compile time and the memory size to be allocated for each declaration is fixed by its data type.

Variable Length-Data

- There is a possibility that data may vary, both in the number of data objects and the size of each object.

- Two examples:
  - The number of arguments in a call may vary from call to call.
  - The size of an array parameter or a local array variable may vary from call to call.

- The printf( ) function in C, where the number of arguments is determined from the format string that is passed as the first argument.

```
printf("%d%s%c", n, prompt, ch);
printf("Hello, world!");
```

Local Temporaries

- Local temporaries are partial results of computations that must be saved across procedure calls.
- Example:
- x[i]=(i+j)*(i/k+f(i));
  - Three partial results need to be saved: the address of x[i], the sum i+j, and the quotient i/k.
- They can be stored
  - In the registers
  - As temporaries on the runtime stack prior the call to f.

Nested Declarations

- Nested declarations can be treated in a similar way to temporary expressions, allocating them on the stack as the block is entered and deleting them on exit.

```
void p (int x, double y)
{  char a;
    int i;
    { double x;  // block A
       int y;
        …
     }
    {  double x; // block B
        int j;
        …
     }
     {  char *a; // block C
         int k;
         …
     }
    }
```

Passing Arguments to main( )

- Arguments are generally passed to a function at the time of call. And function calls are initiated by **main( )**.
- Since **main( )** is the first function to be executed, there is no question of **main( )** being called from any other function.
- So, if **main( )** is not going to be invoked by any other function, is it possible to pass arguments to **main( )** given the fact that arguments are generally passed to a function at the time of invocation.
- The answer lies in understanding command-line arguments.

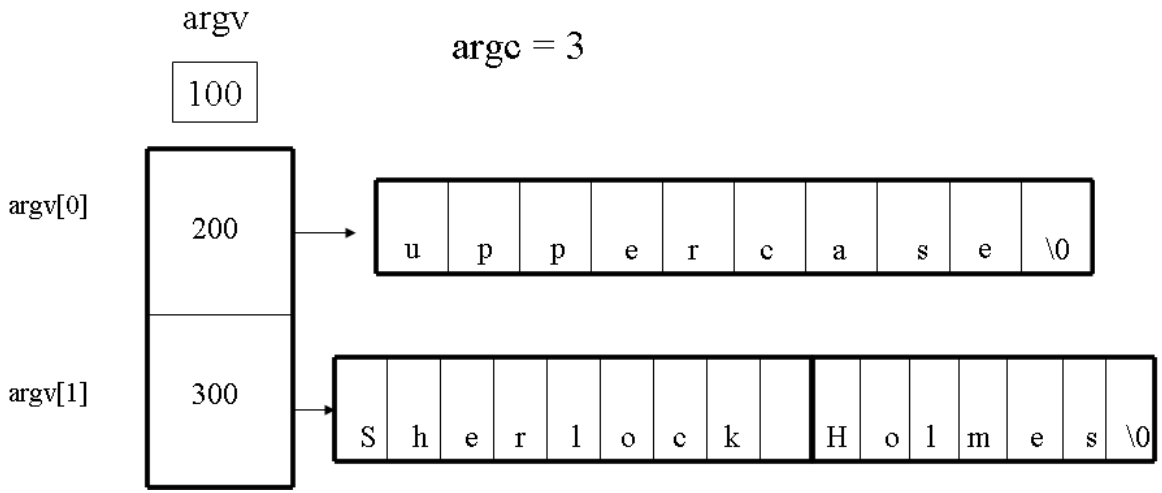
Command-Line Arguments

- The function main( ) can receive arguments from the command line.
- Information can be passed to the function main( ) from the operating system prompt as command line arguments.
- The command line arguments are accepted into special parameters to main( ), namely, argc and argv. Their declarations are as follows:
- main(int argc, char * argv[ ])

Command-Line Arguments (Contd.).

- **argc** provides a count of the number of command line arguments.
- **argv** is an array of character pointers of undefined size that can be thought of as an array of pointers to strings.
- The strings are the words that make up the command line arguments.
- Since the element **argv[0]** contains the command itself, the value of **argc** is at least 1.

Command-Line Arguments (Contd.).



- The program uppercase.c can be coded as follows:

```
#include <stdio.h>
main(int argc, char * argv[ ])
{ int i;
  for (i = 0; argv[1][i] != '\0', i++)
   { if ((argv[1][i] >= 'a') && (argv[1][i] <= 'z'))
     argv[1][i] = argv[1][i] – 32;
   }
  printf( "%s", argv[1]);
}
```

Storage Qualifiers

- The storage qualifier determines the **lifetime** of the storage
- associated with the identified variable.

- A variable also has a **scope**, which is the region of the program in which it is known.

- The storage qualifiers in C are:
  - auto
  - static
  - extern
  - register

automatic Variables

- **auto**matic variables are local to a block, and are discarded on exit from the block.
- Block implies a function block, a conditional block, or an iterative block.
- Declarations within a block create automatic variables if no storage class specification is mentioned, or if the **auto** specifier is used.
- Variable declarations, therefore, are by default, **auto**.

```
#include<stdio.h>
main( )
{ char var;
  while ((var = getchar( )) != '*')
   { if ((var >= 'A') && (var <= 'Z'))
         uppercase_count( );
   }
}
uppercase_count( )
{  auto int counter = 0;
   counter ++;
}
```

Global Variables

- Global variable is defined outside all functions.
- A typical convention of defining global variables is before the main( ) function.
- A variable declared as global is visible to the function main( ) as well as to any other functions called from main( ).
- A variable defined as global has file scope, that is, it is visible to all the functions written within the scope of a program file.

```
/* A sample C program */
 # include <stdio.h>

/* function prototype */
int sum( );

/* a and b are global variables, visible to main( ) as
    well as to sum( ) */
int a=10, b=20;

main()
{  int c;
   c = sum();
   printf("%d+%d = %d \n",a,b,c);
}

int sum()
{  return(a+b);
}
```

Static Variables

- **Static** variables may be local to a block or external to all blocks, but in either case retain their values across exit from, and reentry to functions and blocks.

- Within a block, including a block that provides the code for a function, static variables are declared with the keyword **static**.

- Let us rewrite the code for the example involving auto variables to incorporate the declaration of static variables.

```
#include<stdio.h>
main( )
{
  char var;
  while ((var = getchar( )) != '*')
   {  if ((var >= 'A') && (var <= 'Z'))
         uppercase_count( );
   }
}

uppercase_count( )
{
  static int counter = 0;
  counter ++;
}
```

Static and Global Variables – A Comparison

- From the preceding example, it seems that static variables are functionally similar to global variables in that they retain their value even after exiting from a function in which it has been defined as static.

- But an important difference is that while a global variable has file scope, and is therefore visible to all functions defined within that program file, a static variable is visible only to the function in which it has been defined as static.

Extern Variables

- Variables declared as extern are useful especially when building libraries of functions.

- This means that functions required to run a program can be saved in separate program files, which can be compiled separately, and linked up with the file containing the function main( ) prior to running the program.

- In such a case, if the functions use the same global variables, then their declarations would involve the use of the **extern** qualifier

Extern Variables (Contd.).

- **Program a.c**
- int val /* global */
- main( )
- {
- printf("Enter value");
- scanf("%d", &val);
- compute( ); /* function call */
- }

- **Program b.c**
- compute( )
- {
- extern int val; /* implies that val is defined in another program containing a function to which the current function will be linked to at the time of compilation */
- }

Standard String Handling Functions

- **strcmp( ) – compares two strings (that are passed to it as parameters) character by character using their ASCII values, and returns any of the following integer values.**

Return Value	Implication	Example
Less than 0	ASCII value of the character of the first string is less than the ASCII value of the corresponding character of the second string	i = strcmp(“XYZ”, “xyz”)
Greater than 0	ASCII value of the character of the first string is less than the ASCII value of the corresponding character of the second string	i = strcmp(“xyz”, “XYZ”)
Equal to 0	If the strings are identical	i = strcmp(“XYZ”, “XYZ”)

Standard String Handling Functions (Contd.).

- **strcpy( ) – copies the second string to the first string, both of which are passed to it as arguments.**

- Example: strcpy(string1, “XYZ”) will copy the string “XYZ” to the string string1.

- If string1 were to contain any value, it is overwritten with the value of the second string.

Standard String Handling Functions (Contd.).

- strcat( ) – appends the second string to the first string, both of which are passed to it as arguments.
  - Example

  strcat("Edson Arantes Do Nascimento","Pele");

  will give the string "Edson Arantes Do Nascimento Pele"

- strlen( ) – This function returns the length of a string passed to it as an argument. The string terminator, i.e., the null character is not taken into consideration.
  - Example:

  i = strlen("Johann Cryuff");

  will return the value value 13 into i.

String to Numeric Conversion Functions

- atoi( ) – This function accepts a string representation of an integer, and returns its integer equivalent
  - Example:

    i = atoi("22")

    will return the integer value 22 into the integer i.

- This function is especially useful in cases where main is designed to accept numeric values as command line arguments.

- It may be recalled that an integer passed as a command line argument to main( ) is treated as a string, and therefore needs to be converted to its numeric equivalent.

String to Numeric Conversion Functions (Contd.).

- atof( ) - This function accepts a string representation of a number, and returns its double equivalent
  - For example, if the string str contains the value "1234", then the following statement:

  i = atoi(str);

    - will cause i to have the value 1234.000000

- To use the function atof( ) in any function, its prototype must be declared at the beginning of the function where it is used: double atof( )

Functions for Formatting Data in Memory

- sprintf( ) – this function writes to a variable in memory specified as its first argument.
  - The syntax of the function sprintf( ) is as follows:

  sprintf(string, format specification, data)

- Example:
  - sprintf(str, "%02d-%02d-%02d", 28, 8, 71)

    will return the string 28-08-71 into str.

Functions for Formatting Data in Memory (Contd.).

- sscanf( ) – this function reads from a variable in memory, and stores the data in different memory variables specified.
  - The syntax of the function sscanf( ) is:

  Sscanf(string, format specification, variable list);

- Example:

  sscanf(str,"%2d%2s%s%d", &day, suffix, mnth, &year)

  printf("The day is : %d, suffix is %s, month is %s, year is %d\n", day, suffix, mnth, year);

  - If the data in str is "29th July 1971", the output will be:
  - The day is 29, suffix is th, month is July, year is 1971

Recursion: Factorial Function

- You may therefore write the definition of this function as follows:

- 0! = 1
- 1! = 1
- 2! = 2 * 1
- 3! = 3 * 2 * 1
- 4! = 4 * 3 * 2 * 1

- n! = 1 if n = = 0
- n! = n * (n-1) * (n-2) * ….. * 1 if n > 0.

You may therefore define:
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!

```
prod = 1
for (x = n; x > 0; x--)
 {
  prod *= x;
  return (prod)
 }
```

Evolving a Recursive Definition for the Factorial

- You will now attempt to incorporate this process into an algorithm. You want the algorithm to accept a non-negative integer, and to compute in a variable fact the non-negative integer that is n factorial.

```
if (n == 0)
 fact = 1;
else
      {
        x = n - 1;
        find the value of x!. call it y;
        fact = n * y;
       }
```

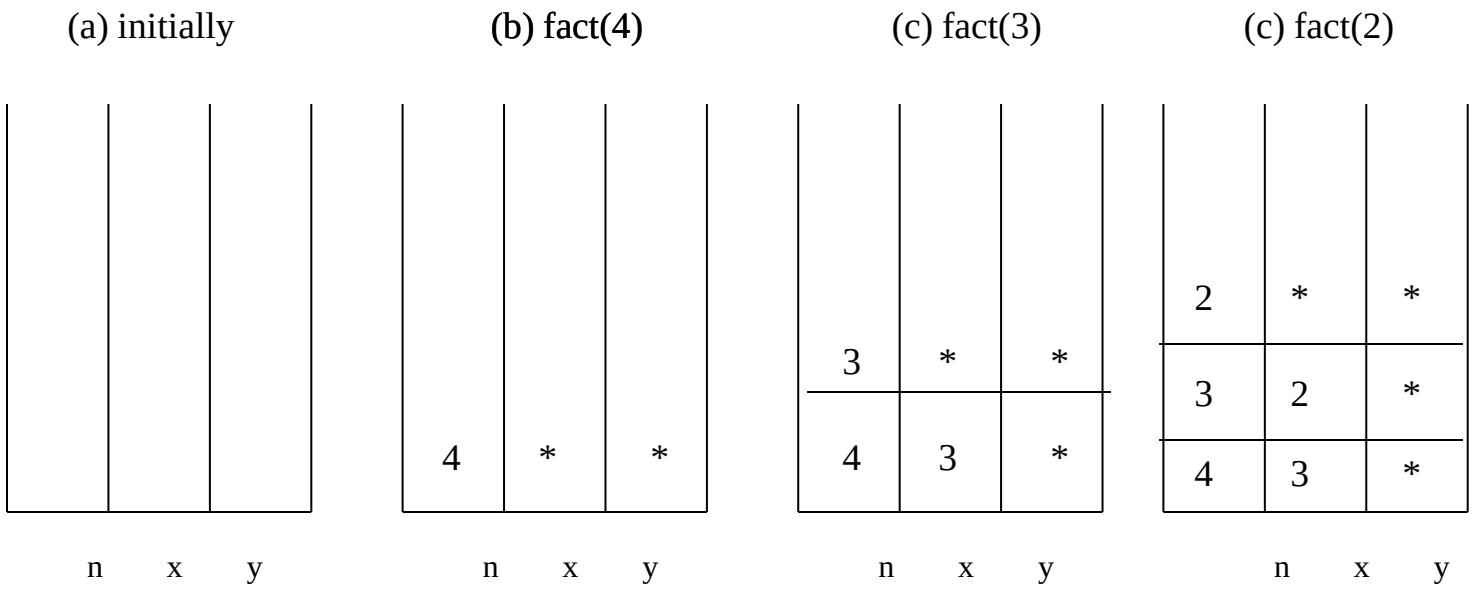
Mechanics of Recursion

- In the statement y = fact(x), the function fact( ) makes a recursive call to itself.
- This is the essential ingredient of a recursive routine.
- The programmer assumes that the function being computed has already been written, and uses it in its own definition.
- However, the programmer must ensure that this does not lead to an endless series of calls.

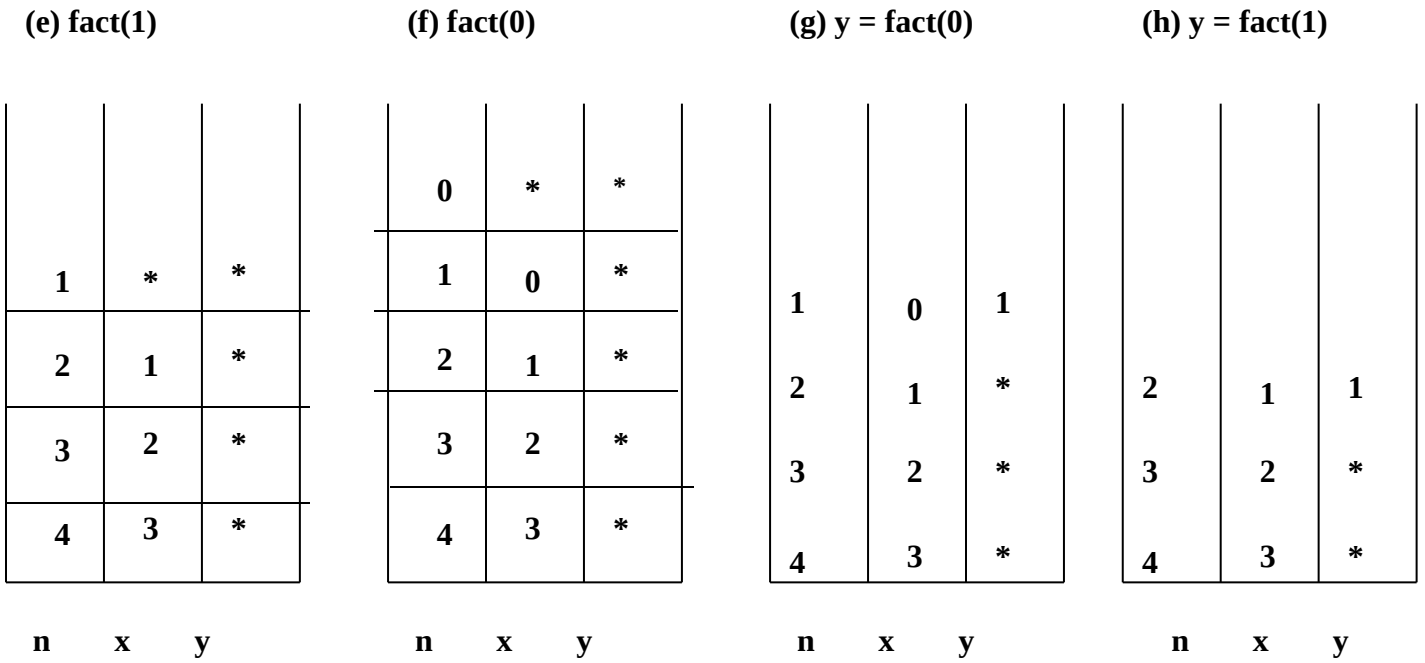
```
int fact(n)
int n;
 {
   int x, y;
   if ( n == 0)
     return (1);
   else
    {
     x = n-1;
     y = fact(x);
     return ( n * y);
    }
  }
```

Mechanics of Recursion (Contd.).

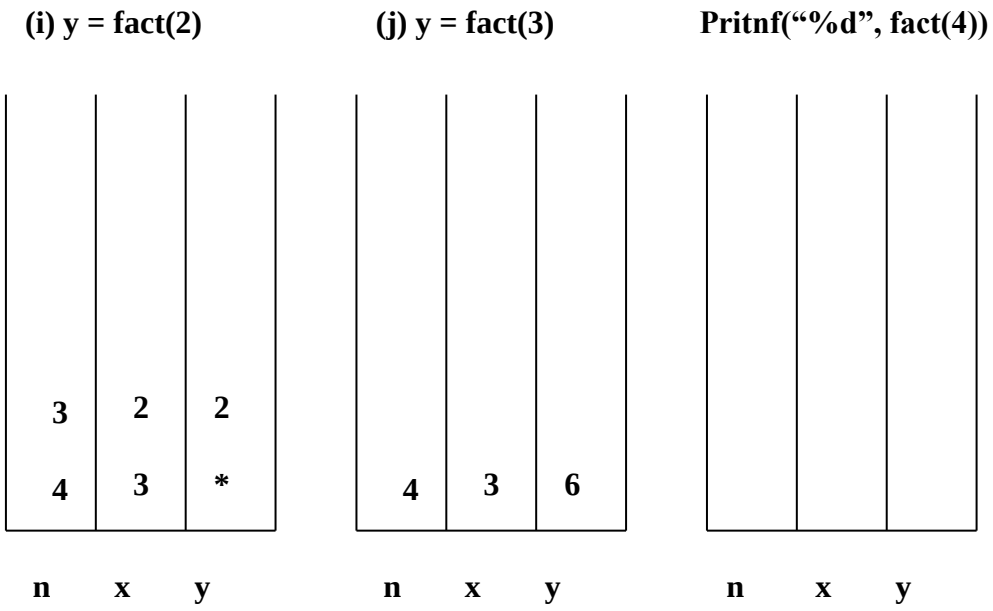
- The recursive call to fact(3) returns to the assignment of the result to y within fact(4), but the call to fact(4) returns to the printf( ) statement in the calling function.



Mechanics of Recursion (Contd.).



Mechanics of Recursion (Contd.).



The stack at various times during execution. An asterisk indicates an uninitialized value.

Function Pointers

- Function Pointers are pointers, i.e. variables, which point to the address of a function.
- Thus a function in the program code is, like e.g. a character field, nothing else than an address.
- When you want to call a function *fn()* at a certain point in your program, you just put the call of the function *fn()* at that point in your source code.
- Then you compile your code, and every time your program comes to that point, your function is called.
- But what can you do, if you don't know at build-time which function has got to be called? Or, invoke functions at runtime.

Function Pointers (Contd.).

- You want to select one function out of a pool of possible functions.
- However you can also solve the latter problem using a switch-statement, where you call the functions just like you want it, in the different branches.
- Consider the example : How are function pointers used? As stated above they are typically used so one function can take in other functions as a parameter.

Declaring and Using Function Pointers

```
#include <stdio.h>
int compute(int,int (*comp)(int) );
int doubleIt( int );
int tripleIt( int );
int main()
{   int x;
    x = compute( 5, &doubleIt );
    printf("The result is: %i\n", x );
    x = compute( 5, &tripleIt );
    printf("The result is: %i\n", x );
    return 0;
}

int compute( int y, int (*comp)(int) )
{    return comp( y );
}
int doubleIt( int y )
{     return y*2;
}
int tripleIt( int y )
{    return y*3;
}
```

Hands-on: 2 hours

- Functions, Pointers and storage taypes

Summary

In this module, we discussed:

- Programs that invoke functions through a:
  - Call by value
  - Call by reference
- Function prototypes
- Function call mechanism
- Command-line arguments
- Using the Auto, Static, and Extern storage qualifiers
- Using string handling functions, conversion functions, and functions for formatting strings in memory
- Recursive function
- Recursive calls to a function using the runtime stack
- Function Pointers

User – Defined Data Types

Module 5

Objectives

At the end of this module, you will be able to:

- Trace down the genesis of user-defined data types
- Declare user-defined data types, namely
  - Structures
  - Unions
  - Enumerations
- Use pointers to structures and unions
- Declare arrays of structures
- Pass structures to functions
- Use the typedef declaration for easier and compact coding
- Use the functions fread( ) and fwrite( ) to read and write structures to and from files

Duration: 4 hrs

User-Defined Data Types – The Genesis

- Consider a situation where your application needs to read records from a file for processing.
- The general approach would be to read the various fields of a record into corresponding memory variables.
- Computations can then be performed on these memory variables, the contents of which can then be updated to the file.
- But, this approach would involve manipulating the current file offset for the relevant fields that need to be updated.

User-Defined Data Types – The Genesis (Contd.).

- Processing would become a lot more simpler if it were possible to read an entire record into an extended variable declaration, the structure of which would be the same as the record in the file.

- This extended variable declaration in turn would be a collection of variables of different data types, each variable matching the data type of the fields in the record.

- The flexibility with such an arrangement is that the collection of variables making up the extended variable declaration can be referred to in the program using a single name.

Structures – The Definition

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

- Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

- An example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc.

Structures – The Definition (Contd.).

- Some of these in turn could be structures: a name has several components, as does an address.
- Other examples of structures are: a point is a pair of coordinates, a rectangle is a pair of points, and so on.
- The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions, and returned by functions
- Automatic structures and arrays may now also be initialized.

Structures – Defining a Type

- When we declare a structure, we are **defining a type**.
- A structure declaration results in the definition of a **template or a blueprint for a user-defined data type**.
- Upon declaring a structure, the compiler identifies the structure declaration as a user-defined data type over and above the fundamental data types, or primitive data types built into the compiler.
- **A structure therefore is a mechanism for the extension of the type mechanism in the C language.**

Declaring a Structure

- The C language provides the struct keyword for declaring a structure. The following is a structure declaration for employee attributes.

```
struct empdata {
    int empno;
    char name[10];
    char job[10];
    float salary;
};
```

Declaring a Structure - Conventions

- The variables named in a structure are called members. A structure member or tag, and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context.

- Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related structure variables.

- Variables of a structure type may immediately follow the structure declaration, or may be defined separately as follows:

Declaring a Structure Variable

```
struct empdata {
    int empno;
    char name[10];
    char job[10];
    float salary;
} emprec; /* emprec is a variable of structure type empdata */
```

- Or a structure can be declared separately as:
- struct empdata emprec;/* emprec is a variable of structure type empdata */

Accessing Elements of a Structure

- Once a structure variable has been declared, the individual members of the structure can be accessed by prefixing the structure variable to the element of the structure.

```
struct empdata {
    int empno;
    char name[10];
    char job[10];
    float salary;
}
```

- struct empdata emprec;
- emprec.empno /* referring to the element of the structure variable emprec */

Passing Structures to Functions

```
#include<stdio.h>
struct salesdata
{  int transaction_number, salesman_number, product_number;
   int units_sold;
  float value_of_sale;
 };
main( )
{   struct salesdata salesvar;
printf("enter transaction number :");
scanf("%d", &salesvar.transaction_number);   fflush(stdin);
printf("enter salesman number :");
scanf("%d", &salesvar.salesman_number);      fflush(stdin);
printf("enter product number :");
scanf("%d", &salesvar.product_number);       fflush(stdin);
printf("enter units sold :");
scanf("%d", &salesvar.units_sold);           fflush(stdin);
compute(&salesvar);
.
.
}
```

Passing Structures to Functions (Contd.).

```
compute( salesdata *salesptr)
{
   static float product_unit_price = {10.0, 20.0,
  30.0, 40.0};
   /*product unit price for products numbered 1
  through 4 */
   salesptr-> value_of_sale = (float)salesptr->
  units_sold *
   product_unit_price[salesptr->product_number - 1]
}
```

Array of Structures

- Just as it is possible to declare arrays of primitive data types, it should also be possible to declare arrays of structures as well.
- If one were to define an array of structure variables, one would do so as follows:

```
struct empdata {
    int empno;
    char name[10];
    char job[10];
    float salary;
};
```

- struct empdata employee_array[4];
- The rationale for declaring an array of structures becomes clear when one wants to improve I/O efficiency in a program.
- Once an array of structures is defined, it is possible to read in a block of records from a file using an appropriate function (fread( )), the details of which you will see shortly.

Writing Records On To a File

- The fwrite( ) function allows a structure variable to be written on to a file.

- The following statement writes the structure variable salesvar on to a file “SALES.DAT, which is pointed to by the FILE type pointer fp:
- fwrite(&salesvar, sizeof(struct salesdata), 1, fp);

- The arguments to the function fwrite( ) are explained as follows:

Writing Structures To a File

- Here &salesrec is the address of the structure variable to be written to the file.
- The second parameter is the size of the data to be written, i.e., size of the structure salesdata. The parameter to the sizeof( ) operator is the structure label, or the structure type itself, and not a variable of the structure type. The sizeof( ) operator can be used to determine the size of any data type in C (fundamental as well as user-defined data types.
- The third parameter of fwrite( ) is the number of structure variables to be written to the file. In our statement, it is 1, since only one structure variable is written to the file. In case, an array of 4 structure variables is to be written to a file using fwrite( ), the third parameter to fwrite( ) should be 4.
- The last parameter is the pointer to the file.

Reading Records from a File

- Records can be read from a file using fread( ). The corresponding read statement using fread( ) for the earlier fwrite( ) statement would be:
- fread(&salesvar, sizeof(struct salesdata), 1, fp);

- Here, the first parameter &salesvar is the address of the structure variable salesvar into which 1 record is to be read from the file pointed to by the FILE type pointer fp.

- The second parameter specifies the size of the data to be read into the structure variable.

Reading Records from a File (Contd.).

- fread( ) will return the actual number of records read from the file. This feature can be checked for a successful read.

- if ((fread(&salesvar, sizeof(struct salesdata), 1, fp)) != 1)
- error code;

- An odd feature of the fread( ) function is that it does not return any special character on encountering end of file.

- Therefore, after every read using fread( ), care must be taken to check for end of file, for which the standard C library provides the feof( ) function. It can be used thus:
- if(feof(fp))

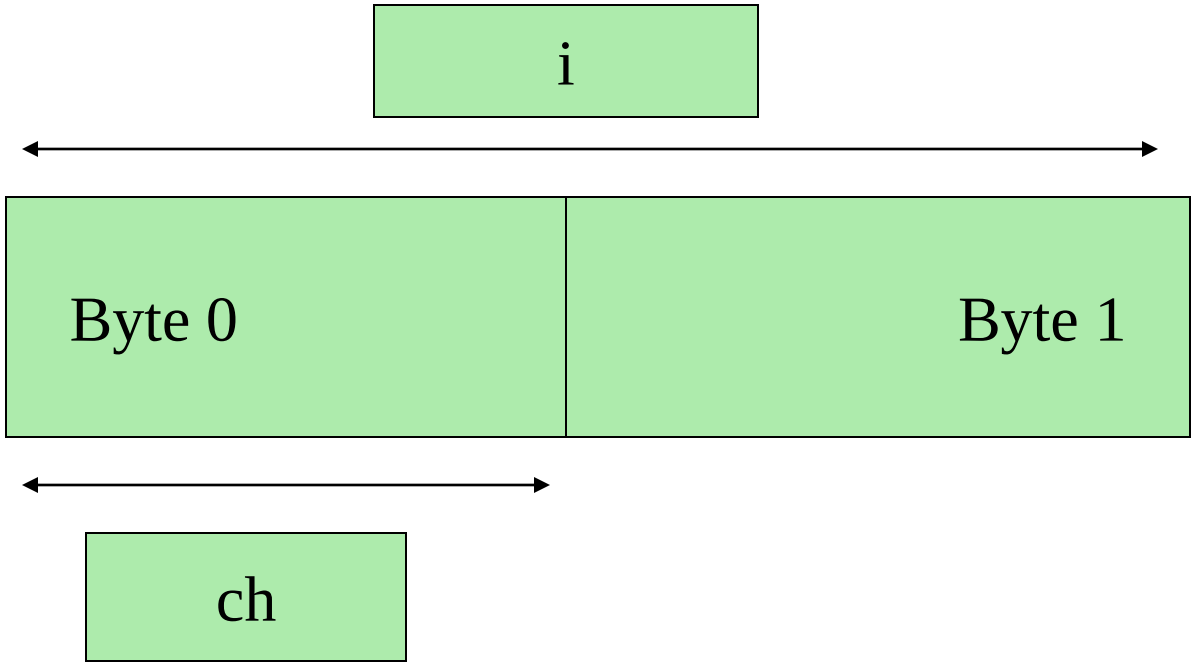
Union

- Can hold objects of different types and sizes at different times
- Syntax similar to structure but meaning is different
- All members of union share same storage space
- Only the last data member defined can be accessed
- Means of conserving memory
- union declaration similar to struct declaration

```
union u_type {
        int i;
        char  ch;
      };
      union u_type cnvt;
```

Unions

- In cnvt, both integer i and character ch share the same memory location. Of course, i occupies 2 bytes (assuming 2-byte integers, and ch uses only one byte.



Unions (Contd.).

- In the following code snippet, a pointer to cnvt is passed to a function:

```
void func1( union u_type *un)
 {
  un->i = 10; /* assign 10 to cnvt using function */
 }
```

- Using a union can aid in the production of machine-independent (portable) code. Because the compiler keeps track of the actual size of the union members, no unnecessary machine dependencies are produced.

Unions - Example

```
union pw
{  short int i;
   char ch[2];
};
int main (void)
 { FILE *fp;
  fp = fopen( "test.tmp", "wb+");
  putw(1000, fp); /* write the value 1000 as an integer */
   fclose( fp );
   return 0;
 }
int putw( short int num, FILE *fp)
 { union pw word;
  word.i = num;
  fputc( word.ch[0], fp); /* write first half */
  fputc( word.ch[1], fp); /* write second half */
 }
```

Enumeration

- Is a set of named integer constants that specify all the legal values a variable of that type can have.
- The keyword enum signals the start of an enumeration type.
- The general form for enumeration is
- enum enum-type-name { enumeration list } variable_list;
- enum coin { penny, nickel, dime, quarter, half_dollar, dollar};
- enum coin money;

Enumeration (Contd.).

- For example, the following code assigns the value of 100 to quarter:
- enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar};
- Now, the values of these symbols are:
- penny 0
- nickel 1
- dime 2
- quarter 100
- half_dollar 101
- dollar 102

Enumeration - Example

```
switch (money)
 {
   case penny : printf( “penny”);   break;
   case nickel : printf( “nickel”); break;
   case dime : printf( “dime”);     break;
   case quarter : printf( “quarter”); break;
   case half_dollar : printf( “half_dollar”); break;
   case dollar : printf( “dollar”);       break;
 }
```

Typedef Statements

- Creates synonyms (aliases) for previously defined datatypes
- Used to create shorter type names
- Format: typedef type new-type;
  - Example: typedef struct Card * CardPtr;
  - defines a new type name CardPtr as a synonym for type struct Card *
- typedef does not create a new datatype
  - Only creates an alias

Hands-on: 2 hours

- Purpose
- construct Structures, Unions & Enumeration and performing computation

Summary

In this module, we discussed:

- The genesis of user-defined data types
- User-defined data types, namely Structures, Unions, and Enumerations
- Pointers to structures and unions
- Arrays of structures
- Passing structures to functions
- The typedef declaration for easier and compact coding
- Functions fread( ) and fwrite( ) to read and write structures to and from files

File Input / Output

Module 6

Objectives

At the end of this module, you will be able to:

- Use pointers of type FILE when performing file I/O
- Perform Input/Output with files
- Use character-based file input/output functions
- Use string-based file input/output functions
- Perform random access on files using the functions
  - fseek( )
  - ftell( )
  - rewind( )
  - feof( )

Duration: 3 hrs

Formatted I/O

- C provides standard functions for performing formatted input and output. These functions accept as parameters a **format string** and a list of variables.

- The format string consists of a specification for each of the variables in terms of its data type, called the conversion character, and width of input or output.

- The format string, along with the data to be output, are the parameters to the printf( ) function.

- Format string is a string containing the format specification introduced by the character %, and ended by a conversion character.

Formatted Output

- An example:
- printf("%c\n", var);
- The conversion characters and their meanings are:

Conversion character	Meaning
d	The data is converted to decimal
c	The data is treated as a character
s	The data is a string and characters from the string are printed until a null character is reached, or until the specified number of characters has been exhausted
f	The data is output as float or double with a default precision of 6

Formatted Output (Contd.).

- Between the % character and the conversion character, there may be:

A minus sign	Implying left adjustment of data
A digit	Implying the minimum in which the data is to be output. If the data has larger number of characters than the specified width, the width occupied by the output is larger. If the data consists of fewer characters than the specified width, it is padded to the right (if minus sign is specified), or to the left (if no minus sign is specified). If the digit is prefixed with a zero, the padding is done with zeroes instead of blanks
A period	Separates the width from the next digit.
A digit	Specifying the precision, or the maximum number of characters to be output
l	To signify that the data item is a long integer, and not an integer.

Formatted Output (Contd.).

Format String	Data	Output
%2d	4	|4|
%2d	224	|224|
%03d	8	|008|
%-2d	4	|4 |
%5s	Sherlock Holmes	| Sherlock Holmes|
%15s	Sherlock Holmes	| Sherlock Holmes |
%-15s	Sherlock Holmes	| Sherlock Holmes |
%f	22.44	|22.440000|

Data Conversion Using Format String

- A variable of a data type can be output as another data type using the conversion character.

```
#include<stdio.h>
main( )
{
   int number = 97;
   printf("Value of num is %d\n", number);
   printf("The Character equivalent of %d is %c\n", number, number);
}
```

Formatted Input

- The function scanf( ) is used for formatted input, and provides many of the conversion facilities of printf( ).

- The scanf( ) function reads and converts characters from standard input according to the format string, and stores the input in memory locations specified by the other arguments.

```
#include<stdio.h>
main( )
{
   char name[10];
   int age;
   char gender;
   scanf ("%s%c%d", name, &gender, &age);
   fflush(stdin);
   printf( "% s %c %d", name, gender, age);
}
```

String Input Using scanf( )

- Remember that while accepting strings using scanf( ), a space is considered as a string terminator. Hence, scanf( ) cannot be used to accept strings with embedded spaces.

- #include<stdio.h>
- main( )
- {
- char string[40];
- printf("Enter a string of maximum 39 characters");
- scanf("%s", string);
- fflush(stdin);
- printf("%s", string);
- }

Files

- A collection of logically related information
- Examples:
  - An employee file with employee names, designation, salary etc.
  - A product file containing product name, make, batch, price etc.
  - A census file containing house number, names of the members, age, sex, employment status, children etc.
- Two types:
  - Sequential file: All records are arranged in a particular order
  - Random Access file: Files are accessed at random

File Access

- The simplicity of file input/output in C lies in the fact that it essentially treats a file as a stream of characters, and accordingly facilitates input/output in streams of characters.

- Functions are available for character-based input/output, as well as string-based input/output from/to files.

- In C, there is no concept of a sequential or an indexed file. This simplicity has the advantage that the programmer can read and write to a file at any arbitrary position.

File Access

- This structure to which the file pointer point to, is of type FILE, defined in the header file <stdio.h>.

- The only declaration needed for a file pointer is exemplified by:
- FILE *fp;
- FILE *fopen(char *name, char *mode);
- fp = fopen(“file name”,“mode”);

- Once the function fopen( ) returns a FILE type pointer stored in a pointer of type FILE, this pointer becomes the medium through which all subsequent I/O can be performed.

File Access Modes (Contd.).

- When opening a file using fopen( ), one also needs to mention the mode in which the file is to be operated on. C allows a number of modes in which a file can be opened.

Mode	Access	Explanation
"r"	Read only mode	Opening a file in "r" mode only allows data to be read from the file
"w"	Write only mode	The "w" mode creates an empty file for output. If a file by the name already exists, it is deleted. Data can only be written to the file, and not read from it

File Access Modes

Mode	Access	Explanation
“a”	Append mode	Allows data to be appended to the end of the file, without erasing the existing contents. It is therefore useful for adding data to existing data files.
“r+”	Read + Write mode	This mode allows data to be updated in a file
“w+”	Write + Read mode	This mode works similarly to the “w” mode, except that it allows data to be read back after it is written.
“a+”	Read + Append mode	This mode allows existing data to be read, and new data to be added to the end of the file.

Character-based File I/O

- In C, character-based input/output from and to files is facilitated through the functions **fgetc( )** and **fputc( )**.

- These functions are simple extensions of the corresponding functions for input/output from/to the terminal.

- The only additional argument for both functions is the appropriate file pointer, through which these functions perform input/output from/to these files.

A File Copy Program

```
#include<stdio.h>
main( )
{
  FILE *fp1, *fp2;
  fp1 = fopen( “source.dat”, “r”);
  fp2 = fopen( “target.dat”, “w”);
  char ch;
  while ( (ch = fgetc( fp1)) != EOF)
   {
     fputc (ch, fp2);
   }
fclose(fp1);
fclose(fp2);
}
```

Variation to Console-Based I/O

```
#include<stdio.h>
main( )
{
 char ch;
  while ( (ch = fgetc( stdin)) != EOF)
   {
     fputc (ch, stdout);
   }
}
```

Nuggets on FILE Type Pointers

- The important point to note is that in C, devices are also treated as files. So, the keyboard and the VDU are also treated as files.

- It would be interesting here to know what **stdin, stdout, and stderr** actually are.

- **stdin**, **stdout**, and **stderr** are pointers of type **FILE** defined in **stdio.h**. **stdin** is a **FILE** type pointer to standard input, **stdout** is a **FILE** type pointer to standard output, and **stderr** is a **FILE** type pointer to the standard error device.

Nuggets on FILE Type Pointers (Contd.).

- In case fopen( ) is unsuccessful in opening a file (file may not exist, or has become corrupt), it returns a null (zero) value called NULL.
- NULL is defined in the header file stdio.h
- The NULL return value of fopen( ) can be used for error checking as in the following line of code:
- if ((fp = fopen("a.dat","r")) = = NULL)
- printf("Error Message");

The exit( ) function

- The **exit( )** function is generally used in conjunction with checking the return value of the **fopen( )** statement.

- If **fopen( )** returns a **NULL**, a corresponding error message can be printed, and program execution can be terminated gracefully using the **exit( )** function.

```
if ((fp = fopen("a.dat", "r")) = = NULL)
{
  printf("Error Message");
  exit( );
}
```

Line Input/ Output With Files

- C provides the functions fgets( ) and fputs( ) for performing line input/output from/to files.

- The prototype declaration for fgets( ) is given below:
- char* fgets(char *line, int maxline, FILE *fp);

- The explanations to the parameters of fgets( ) is:
  - char* line – the string into which data from the file is to be read
  - int maxline – the maximum length of the line in the file from which data is being read
  - FILE *fp – is the file pointer to the file from which data is being read

Line Input/ Output With Files (Contd.).

- **fgets( )** reads the next input line (including the newline) from file fp into the character array **line**;
- At most **maxline-1** characters will be read. The resulting line is terminated with **'\0'**.
- Normally **fgets( )** returns line; on end of file, or error it returns **NULL**.

Line Input/ Output With Files (Contd.).

- For output, the function fputs( ) writes a string (which need not contain a newline) to a file:
  - int fputs(char *line, FILE *fp)
- It returns EOF if an error occurs, and non-negative otherwise.

File Copy Program Using Line I/O

```
#define MAX 81;
#include<stdio.h>
main( )
{
  FILE *fp1, *fp2;
  fp1 = fopen( "source.dat", "r");
  fp2 = fopen( "target.dat", "w");
  char string[MAX];
  while ( (fgets(string, MAX, fp1)) != NULL)
  {
     fputs (string, fp2);
   }
fclose(fp1);
fclose(fp2);
}
```

Formatted File Input/ Output

- C facilitates data to be stored in a file in a format of your choice. You can read and write data from/to a file in a formatted manner using **fscanf( )** and **fprintf( )**.

- Apart from receiving as the first argument the format specification (which governs the way data is read/written to/from a file), these functions also need the file pointer as a second argument.

- fscanf( ) returns the value EOF upon encountering end-of-file.

Formatted File Input/ Output (Contd.).

- fscanf( ) assumes the field separator to be any white space character, i.e., a space, a tab, or a newline character.

- The statement printf("The test value is %d", i); can be rewritten using the function fprintf( ) as:
  - fprintf(stdout, "The test value is %d", x);

- The statement scanf("%6s%d, string, &i) can be rewritten using the function fscanf( ) as:
  - fscanf(stdin, "%6s%d", string, &i);

Random Access

- Input from, or output to a file is effective relative to a position in the file known as the current position in the file.
- For example, when a file is opened for input, the current position in the file from which input takes place is the beginning of the file.
- If, after opening the file, the first input operation results in ten bytes being read from the file, the current position in the file from which the next input operation will take place is from the eleventh byte position.

Random Access (Contd.).

- It is therefore clear that input or output from a file results in a shift in the current position in the file.
- The current position in a file is the next byte position from where data will be read from in an input operation, or written to in an output operation.
- The current position advances by the number of bytes read or written.
- A current position beyond the last byte in the file indicates end of file.

The fseek( ) Function

- The function fseek( ) is used for repositioning the current position in a file opened by the function fopen( ).

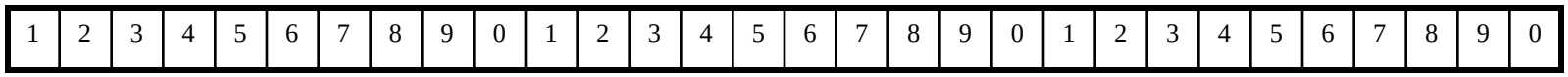
- int rtn = fseek(file pointer, offset, from where)
- where,
- int rtn is the value returned by the function fseek( ). fseek( ) returns the value 0 if successful, and 1 if unsuccessful.
- FILE file-pointer is the pointer to the file
- long offset is the number of bytes that the current position will shift on a file
- int from-where can have one of three values:
  - from beginning of file (represented as 0)
  - from current position (represented as 1)
  - from end of file (represented as 2)

The fseek( ) function (Contd.).

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

- fp = fopen("employee.dat", r)

employee.dat

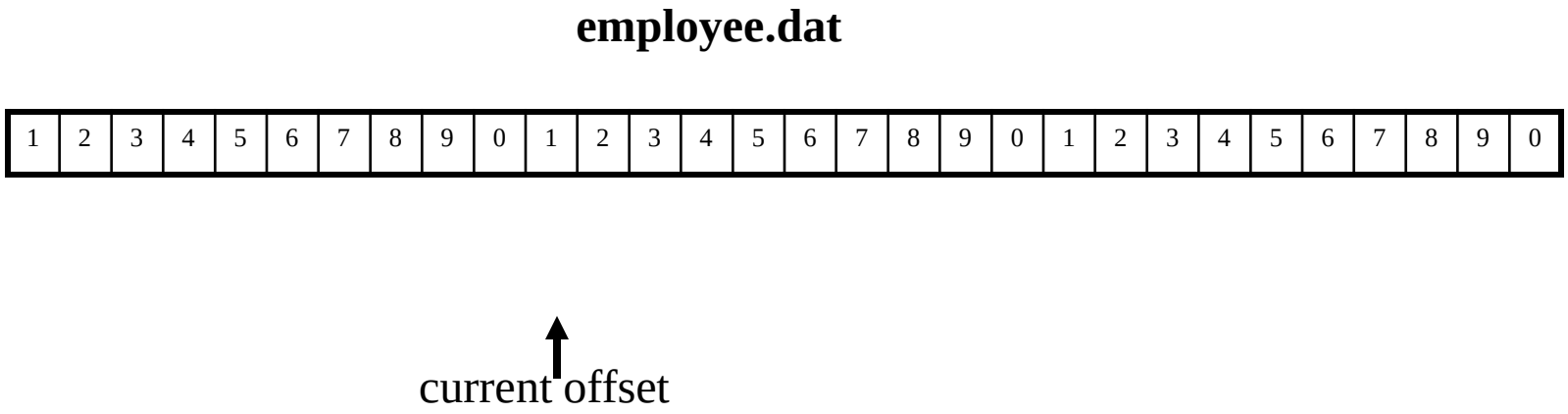


current offset

The fseek( ) function (Contd.).

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

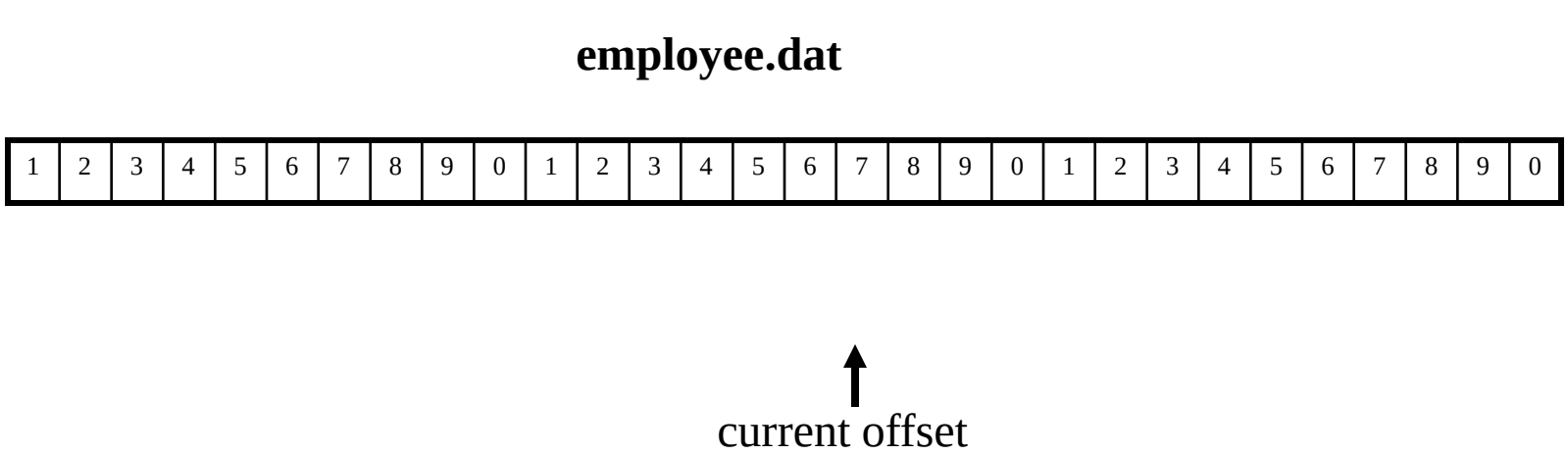
- fseek(fp, 10L, 0);



The fseek( ) function (Contd.).

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

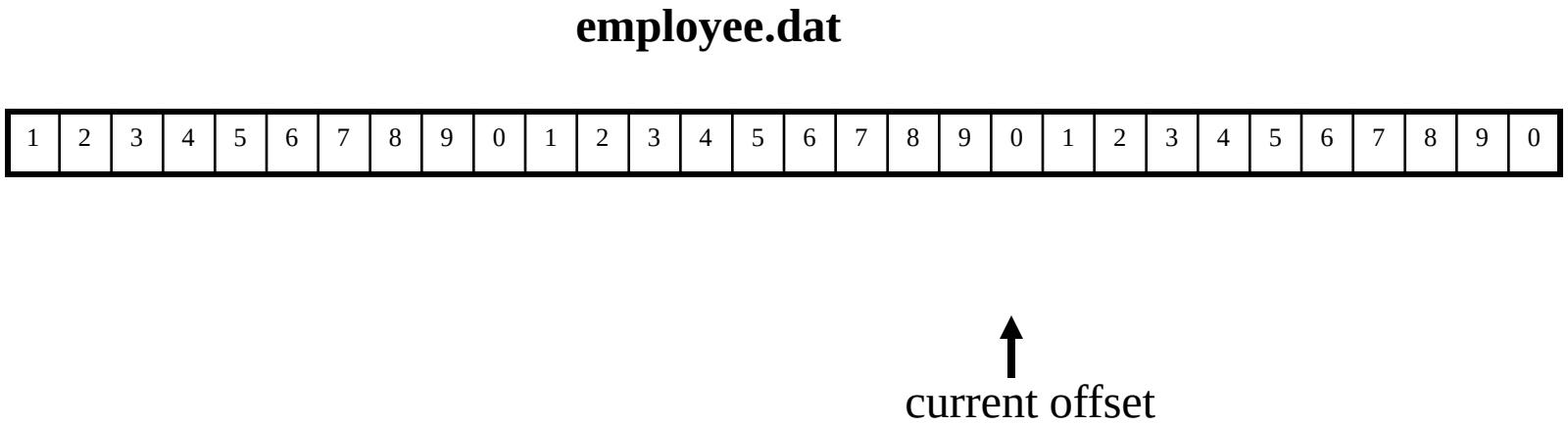
- fgets(string, 7, fp);



The fseek( ) function

- Consider the following schematic representation of a file in terms of a string of 30 bytes. The file contains records and fields that are not delimited by any special character.

- fseek(fp, -10L, 2)



The rewind( ) Function

- The function, rewind( ) is used to reposition the current position in the file (wherever it may be) to the beginning of the file.

- The syntax of the function rewind( ) is:
  - rewind (file-pointer);
  - where file-pointer is the FILE type pointer returned by the function fopen( ).

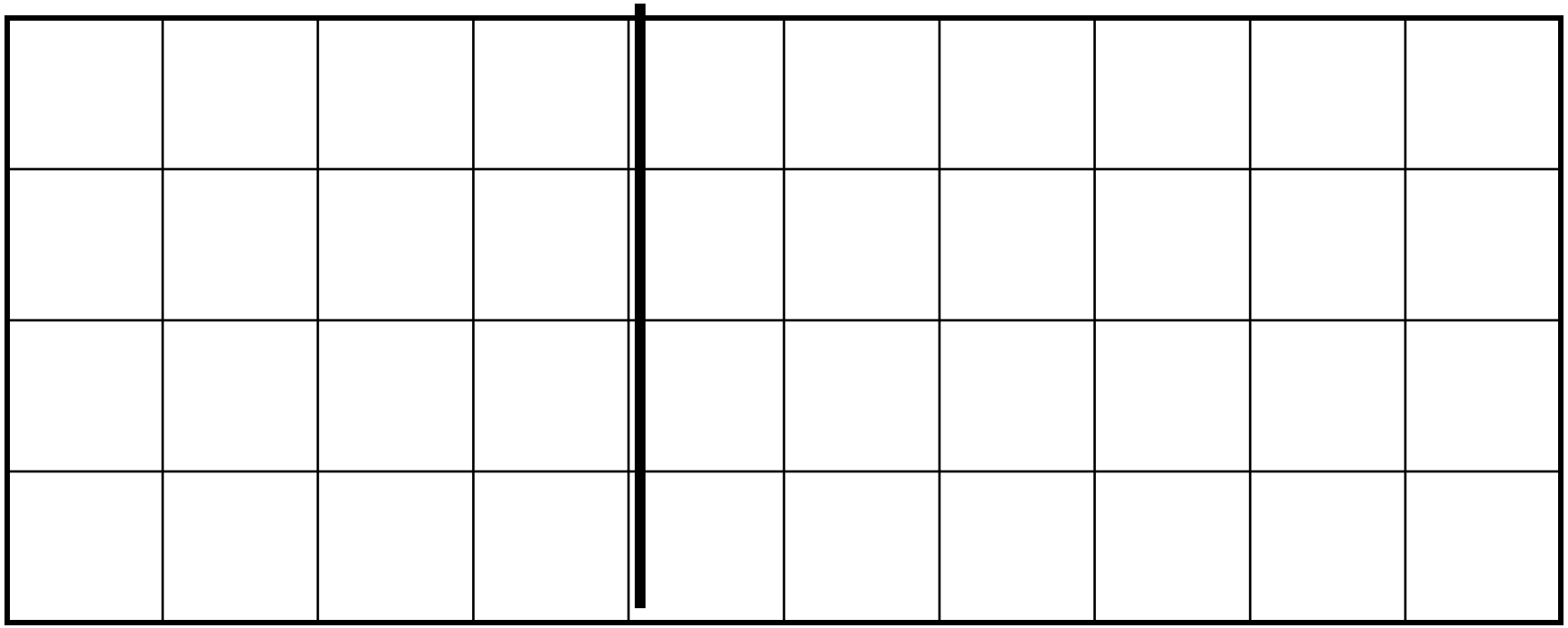
- After invoking rewind( ), the current position in the file is always the first byte position, i.e., at the beginning of the file.

Updating Data in a File

- A file that needs to be updated should be opened using fopen( ) in the "r+" mode, i.e., opening a file for read and write.
- The "r+" mode is useful for operations on files that need to be updated.
- Consider a file, "SALARY.DAT" which contains the following structure:

Updating Data in a File (Contd.).

Structure of SALARY.DAT



Updating Data in a File (Contd.).

- /* function to read the salary field (beginning at byte no. 5 and ending at byte 10) for each record in the file, and increase it by 100 */

```
#include<stdio.h>
main( )
{
 FILE *fp;
 char empno[5], salary[7];
 double fsalary, atof( );
 long int pos = 4L, offset = 4L;
 /* open the file SALARY.DAT in read-write mode */
 if ((fp = fopen( "SALARY.DAT", "r+")) = = NULL)
  {
   printf("Error opening file SALARY.DAT");
   exit( );
  }
```

Updating Data in a File (Contd.).

```
while(( fseek( fp, offset, 1)) = = 0)
{
  fgets(salary, 7, fp);
  f_salary = atof(salary) + 100;
  sprintf(salary, "%6.2f", f_salary); /*convert
  f_salary to a string */
  fseek(fp, pos, 0); /* reposition at start of
  salary field */
  fputs(salary, fp); /* update salary field
  pos += 10; /* increment pos to starting byte of
  salary field for the next record */
}
printf("The file SALARY.DAT has been updated");
fclose(fp);
}
```

The ftell( ) and feof( ) Function

- The prototype declaration for the function ftell( ) is:
  - long ftell(FILE *fp)
- ftell returns the current file position for stream, or -1 on error.
- The prototype declaration for the function feof( ) is:
  - int feof(FILE *fp)
- feof returns non-zero if the end of file indicator for stream is set.

Hands-on: 2 hours

Purpose

- Performing file related i/o operations

Summary

In this module, we discussed:

- Using pointers of type FILE when performing file I/O
- Perform Input/Output with files
- Using character-based file input/output functions
- Using string-based file input/output functions
- Performing random access on files using the functions
  - fseek( )
  - ftell( )
  - rewind( )
  - feof( )

Introduction to Data Structures

Module 7

Objectives

At the end of this module, you will be able to:

- Define and Data Structure
- Classify Data Structures
- List significance of Data Structures
- Describe the operations on Data Structures
- State the need for dynamic memory allocation
- Use the malloc() function to allocate memory
- Define self-referential structures and their advantages

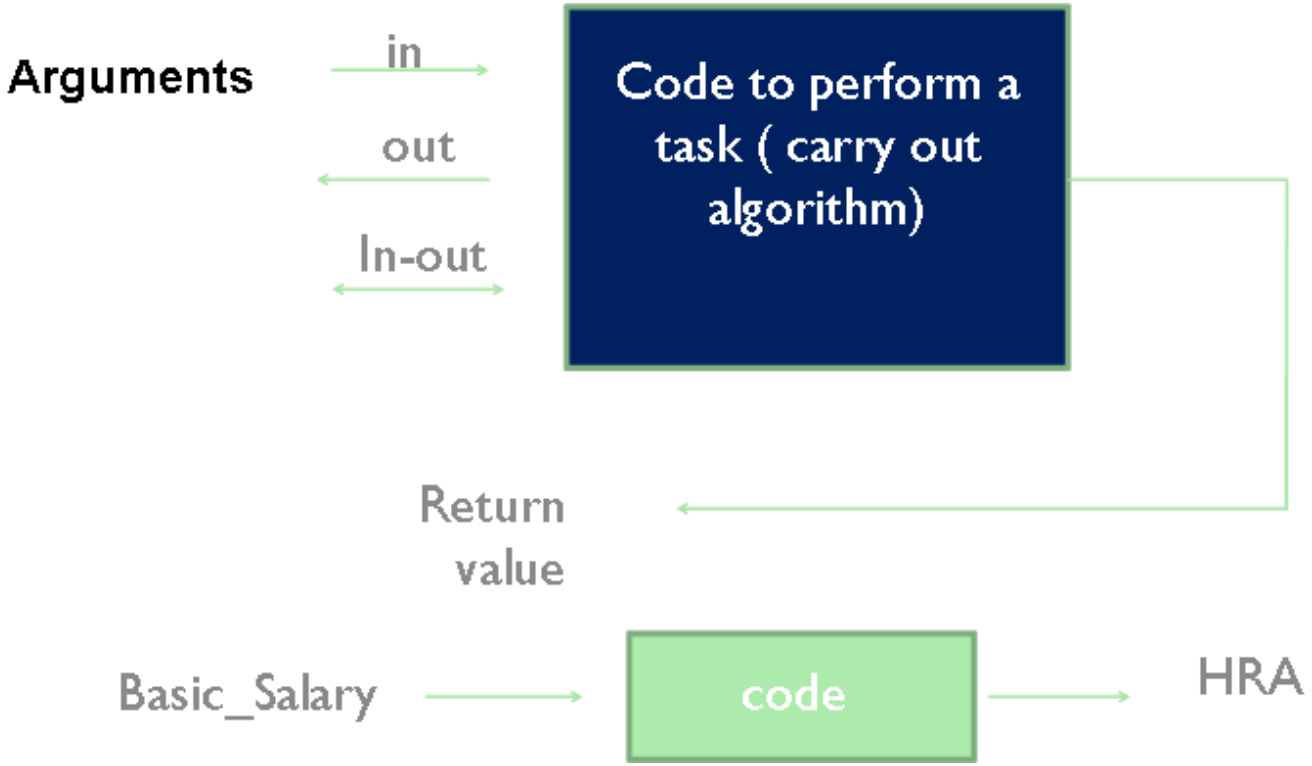
Duration: 3 hrs

Introduction

- Goal of Data Types and Data Structures is to organize data
  - Data Structure is a way of organizing of data element(s) in computer memory in a manner that is convenient to perform operations on it
- Criteria: to facilitate efficient
  - storage of data
  - retrieval of data
  - manipulation of data
- Design Issue:
  - select and design appropriate data types

What is Program

- A Set of Instructions
- Data Structures + Algorithms
- Algorithm = Logic + Control

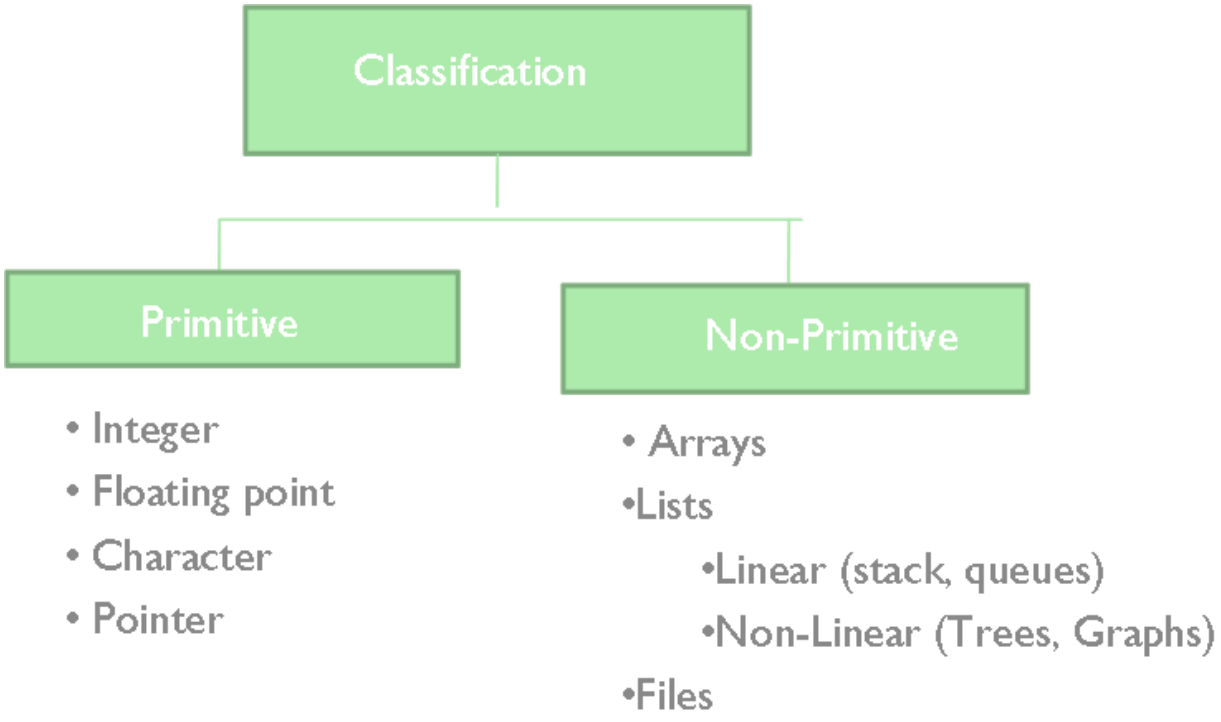


Functions of Data Structures

- Add / create — Allocation of Memory
  - Index
  - Key
  - Position
  - Priority
- Delete / Destroy — De-Allocation of Memory
- Get / Selection — Accessing the data elements
- Change / Updation — Modify and Store

Common Data Structures

- Array
- Stack
- Queue
- Linked List
- Tree
- Heap
- Hash Table
- Priority Queue



- How many Algorithms?
  - Countless

Which Data Structure or Algorithm is better?

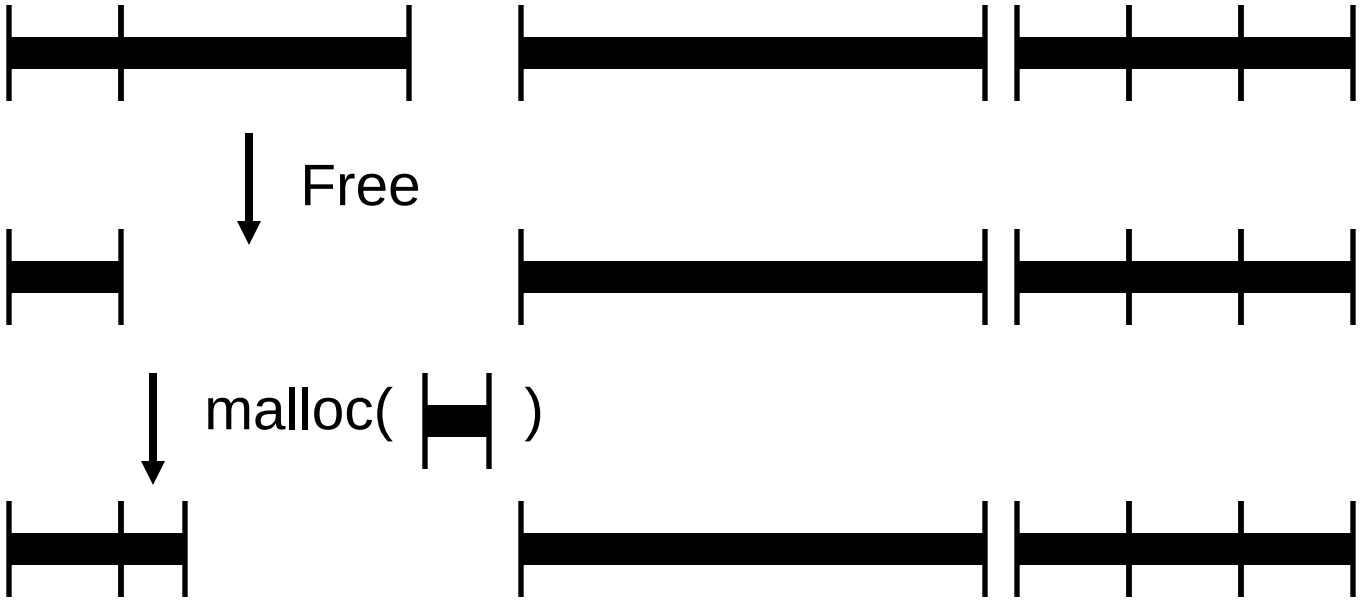
- Must Meet Requirement
- High Performance
- Low RAM footprint
- Easy to implement
  - Encapsulated

Dynamic Storage Allocation

- Three issues:
- How to maintain information about free memory
  - Linked list
- The algorithm for locating a suitable block
  - First-fit
- The algorithm for freeing an allocated block
  - Coalesce adjacent free blocks
- The C language provides the malloc() function which a program can use to declare variables dynamically.

Dynamic Storage Allocation (Contd.).

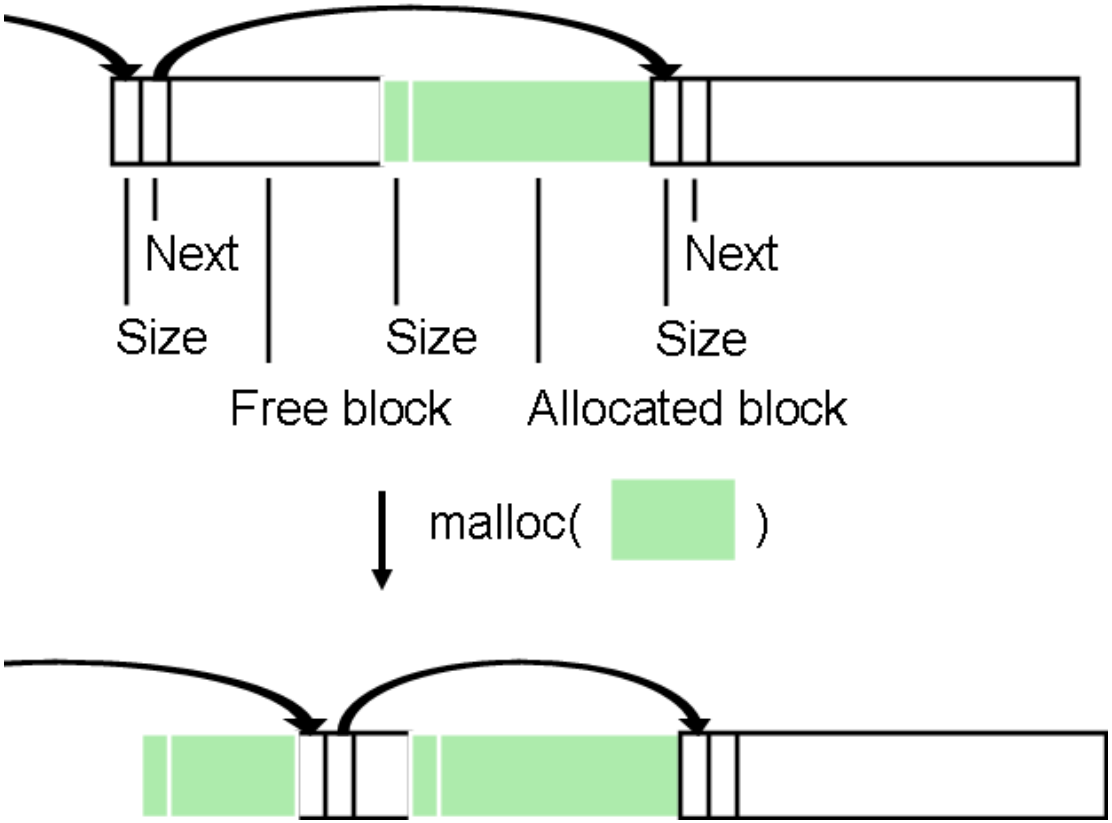
- What are malloc() and free() actually doing?
- Pool of memory segments:



Dynamic Storage Allocation (Contd.).

- Rules:
  - Each segment contiguous in memory (no holes)
  - Segments do not move once allocated
- malloc()
  - Find memory area large enough for segment
  - Mark that memory is allocated
- free()
  - Mark the segment as unallocated

Simple Dynamic Storage Allocation



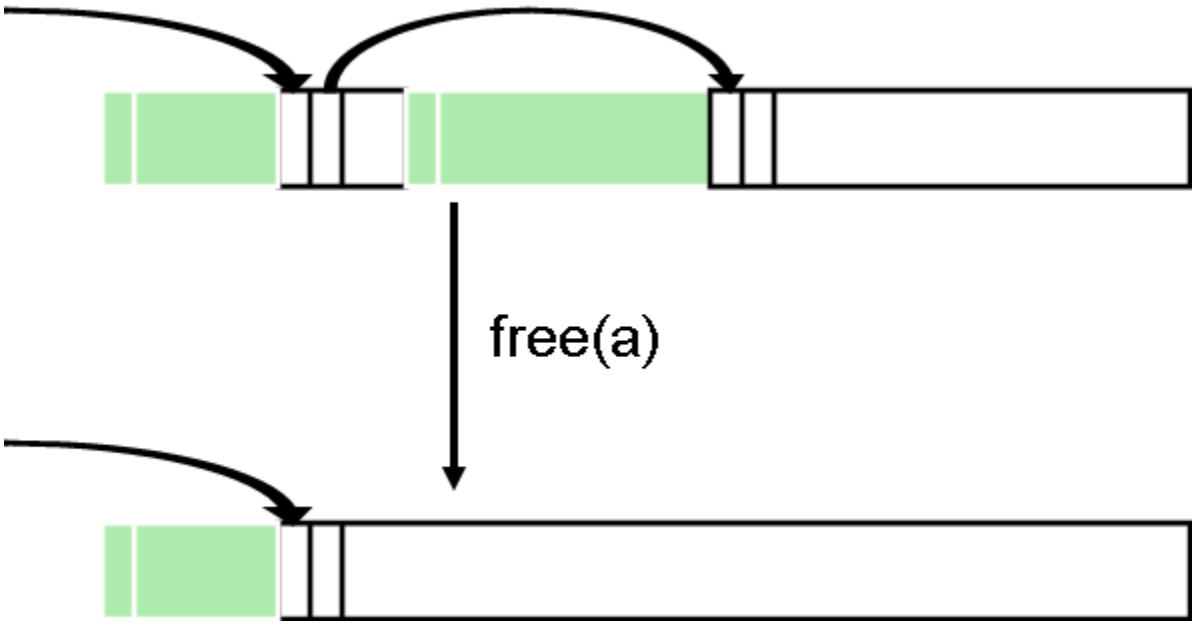
First large-enough free block selected

Free block divided into two

Previous next pointer updated

Newly-allocated region begins with a size value

Simple Dynamic Storage Allocation (Contd.).



Appropriate position in free list identified

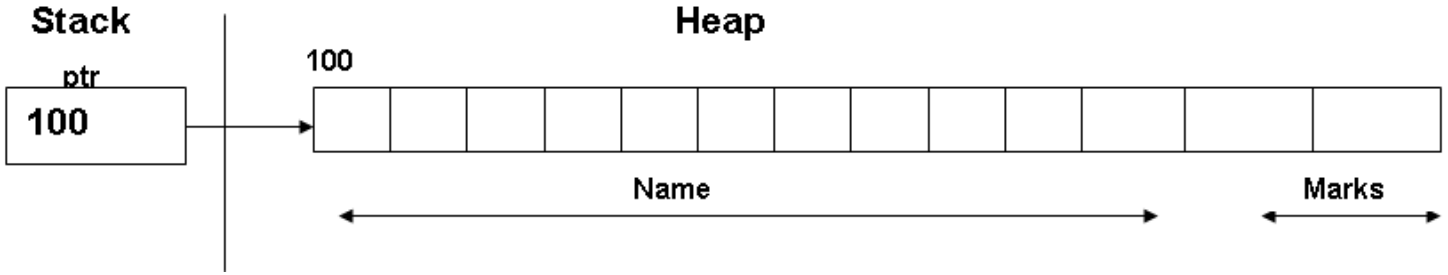
Newly-freed region added to adjacent free regions

malloc() and free()

- The parameter to **malloc( )** is an unsigned integer which represents the number of bytes that the programmer has requested **malloc( )** to allocate on the heap.

```
int *a;
a = (int *) malloc(sizeof(int) * k);
a[5] = 3;
free(a);
```

- The **sizeof( )** operator can be used to determine the size of any data type in C, instead of manually determining the size and using that value. Therefore, the benefit of using **sizeof( ) operator in any program makes it portable**.



malloc() and free() (Contd.).

- More flexible than automatic variables (stacked)
- More costly in time and space
  - malloc() and free() use complicated non-constant-time algorithms
  - Each block generally consumes two additional words of memory
    - Pointer to next empty block
    - Size of this block
- Common source of errors
  - Using uninitialized memory
  - Using freed memory
  - Not allocating enough
  - Neglecting to free disused blocks (memory leaks)

The malloc( ) Function

- The malloc( ) function also returns the starting address to the marks_data structure variable on the heap.

- However, malloc( ) returns this not as a pointer to a structure variable of type marks_data , but as a void pointer.

- Therefore, the cast operator was used on the return value of malloc( ) to cast it as a pointer to a structure of type marks_data before being assigned to ptr, which has accordingly been defined to be a pointer to a structure of type marks_data.

The malloc( ) Function – Example

```
#include<stdio.h>
main()
  {
    struct marks_data
     {
      char name[11];
      int marks;
     };
     struct marks_data *ptr;
    /* declaration of a stack variable */
     ptr = (struct marks_data *)
     malloc(sizeof(struct marks_data));
/* declaration of a block of memory on the heap and  the block in turn being referenced by ptr */
  }
```

Self-Referential Structures

- Suppose, you have been given a task to store a list of marks. **The size of the list is not known**.

- If it were known, then it would have facilitated the creation of an array of the said number of elements and have the marks entered into it.

- Elements of an array are contiguously located, and therefore, array manipulation is easy using an integer variable as a subscript, or using pointer arithmetic.

Self-Referential Structures (Contd.).

- However, when runtime variables of a particular type are declared on the heap, let's say a structure type in which we are going to store the marks, each variable of the structure type marks will be located at a different memory location on the heap, and not contiguously located.

- Therefore, these variables cannot be processed the way arrays are processed, i.e., using a subscript, or using pointer arithmetic.

- An answer to this is a self-referential structure.

Self-Referential Structures (Contd.).

- A self-referential structure is so defined that one of the elements of the structure variable is able to reference another subsequent structure variable of the same type, wherever it may be located on the heap.
- In other words, each variable maintains a link to another variable of the same type, thus forming a non-contiguous, loosely linked data structure.
- This self-referential data structure is also called a linked list.

Summary

In this module, we discussed:

- Classification of Data Structures
- Significance of Data Structures
- Operations on Data Structures
- Dynamic memory allocation
- Using the *malloc* function and *free* function
- Self-referential structures and their advantages

Linked Lists

Module 8

Objectives

- At the end of this module, you will be able to:

- Define Linked lists
- Classify Linked Lists
- Compare Array with Linked List
- Write code to:
  - Create a sorted linked list,
  - Insert nodes into a sorted linked list
  - Traverse a linked list
  - Delete nodes from a linked list

Duration: 6 hrs

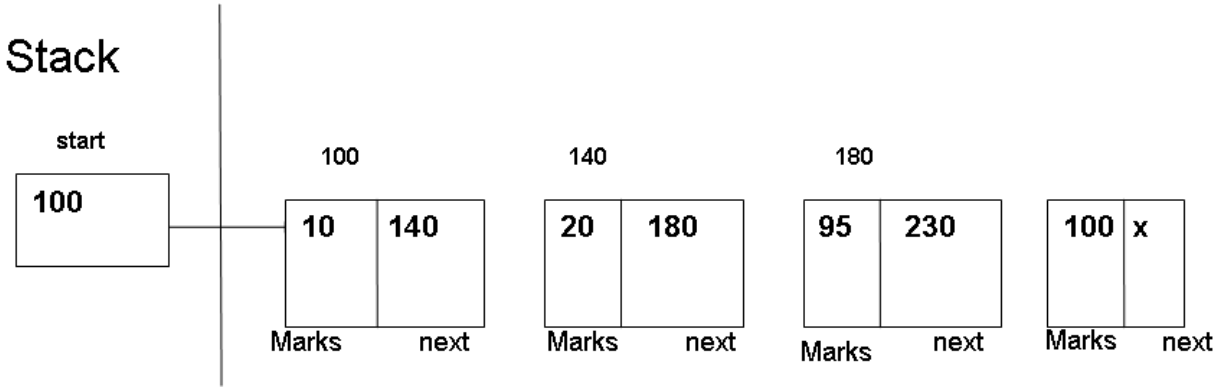
Linked Lists

- A **linked List** is a collection of structures ordered not by their *physical placement* in memory but by *logical links* that are stored as part of the data in the structure itself.

- The link is in the form of a pointer to another structure of the same time.

- Each structure of the list is called a **node** and consists of two fields, one containing the item,and the other containing the address of the next item.
  Such structures are called *Self-referential*.

```
struct node
    {
    type item;
    struct node *next;
    };
```



Arrays & Linked Lists

Arrays

- The sequential organization is provided implicitly by its index.
- The memory allocation is **static**.
- Elements are *physically and logically adjacent.*
- Binary and Linear search applicable. **Searching is fast**
- Insertion and deletion of elements takes more time.
- Arrays take less memory than linked lists.

Linked Lists

- The order of the list is given by links from one item to another.
- Memory allocation is **dynamic**
- Elements are *logically adjacent*
- Only linear search applicable. **Searching is slow**
- Insertion and deletion of elements is fast.
- Linked Lists take more memory than arrays (address of the next node is to be stored)

Creating a Linked List..

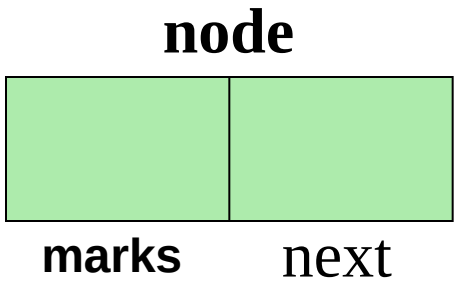
```
Struct node
 {
    int marks;
    struct node *next;
 };
```

- The struct declaration merely describes the format of the nodes and does not allocate storage.
- Storage space for a node is created only when the function **malloc** is called.

```
struct node *head;
head=(struct node *)malloc(sizeof(node));
```

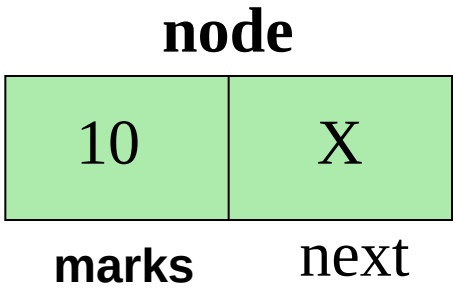
Creating a Linked List. (Contd.).

- malloc allocates a piece of memory that is sufficient to store a node and assigns its address to the pointer variable head



- The following statements store values in the member fields

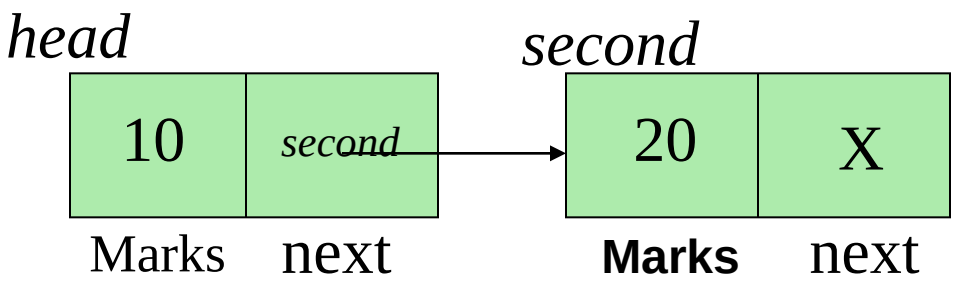
```
head->no=10;
head->NULL;
```



Creating a Linked List. (Contd.).

- The second node can be added as follows:

```
second=(struct node *)malloc(sizeof(struct node));
second->marks=20;
second->next=NULL;
head->next=second;
```



- This process can be easily implemented using iteration techniques.
- The pointer can be moved from current node to the next by a self-replacement statement:

```
head=head->next;
```

Inserting an element

- The process of insertion precedes a search for the place of insertion.A search involves in locating a node after which the new item is to be inserted.

- The general algorithm for insertion is as follows:
-
- Begin

  if list is empty or the new node comes before the head node then,

  insert the new node as the head node.

  else if the new node comes after the last node ,then,

  insert the new node as the end node.

  else insert the new node in the body of the list.
- End

Deleting an element

- To delete a node ,only one pointer value needs to be changed.

- The general algorithm for deletion :

Begin

if the list is empty, then, node cannot be deleted.

else if node to be deleted is the first node,*then*,make the head to point to the second node.

else delete the node from the body of the list.

End

- The memory space of deleted node may be replaced for re-use. The process of deletion also involves search for the item to be deleted.

Creating a Sorted Linked List

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

struct marks_list *start, *prev;
/* variables declared outside main() are global
      in nature and can be accessed by other
      functions called from main() */

struct marks_list
 {
  int marks;
  struct marks_list *next;
 };
main()
   { struct marks_list * makenode();
```

- struct marks_list *new;
- start = NULL;
- char menu = '0 ';

```
while (menu != '4')
  {
    printf( "Add Nodes     :\n");
    printf( "Delete Nodes  :\n");
    printf( "Traverse a list :\n");
    printf( "Exit          :\n");
    menu = getchar( );
switch (menu)
 {
   case '1' : addnode( );          break;
   case '2' : deletenode( );       break;
   case '3' : traverse( );         break;
   case '4': exit( );              break;
} /* end of switch */
   } /* end of main( ) */
```

Creating a Sorted Linked List (Contd.).

```
addnode( )
 {   char ch = 'y';
    while ( ch = = 'y' )
     {       new = makenode();
      /* creation of a list is treated as a
     special case of insertion  */
      if ( start == NULL)
            {          start = new;        }
      else {           insert();
                       traverse( );       }
 printf("%s","Want to add more
     nodes\n");
  scanf( "%c", &ch );
  fflush( stdin );
     } /* end of while  */
} /* end of addnode( )
```

```
struct marks_list * makenode()
  {
   struct marks_list *new;
   new=(struct marks_list *)
    malloc(sizeof(struct(marks_list));
   scanf("%d",&new->marks);
   new->next = NULL;
   return(new);
  }
```

Creating a Sorted Linked List (Contd.).

```
insert(struct marks_list *start)
{    struct marks_list *ptr, *prev;

     for(ptr=start,prev=start;(ptr);prev=ptr,
     ptr=ptr->next)
 {   if (new->marks < start->marks)
   {  /* insertion at the beginning of a  list
      */
    new->next = start;
    start = new;
  }
```

```
/* insertion in the middle of a list */
    if(new->marks  > ptr->marks)
     {        continue;              }
    else {  prev->next = new;
             new->next = ptr;  }
      } /* end of for loop */

/* insertion at the end of the list */
if (ptr == null)
  { prev->next = new;
    new->next = null;
  }
} /* end of insert */
```

Linked List – Search, Delete

```
struct marks_list *search( int val)
{
  for( ptr = start; (ptr); ptr = ptr-
     >next)
   {
     if (val = = ptr-> marks)
       return ptr;
   }
```

```
/* this code would hold true for
     deletion in the middle and at
     the end of a linked list */
 if (score = = ptr-> marks)
    {
      prev-> next = ptr-> next;
      free(ptr);
     }
   }/* end of for loop */
} /* end of delete */
```

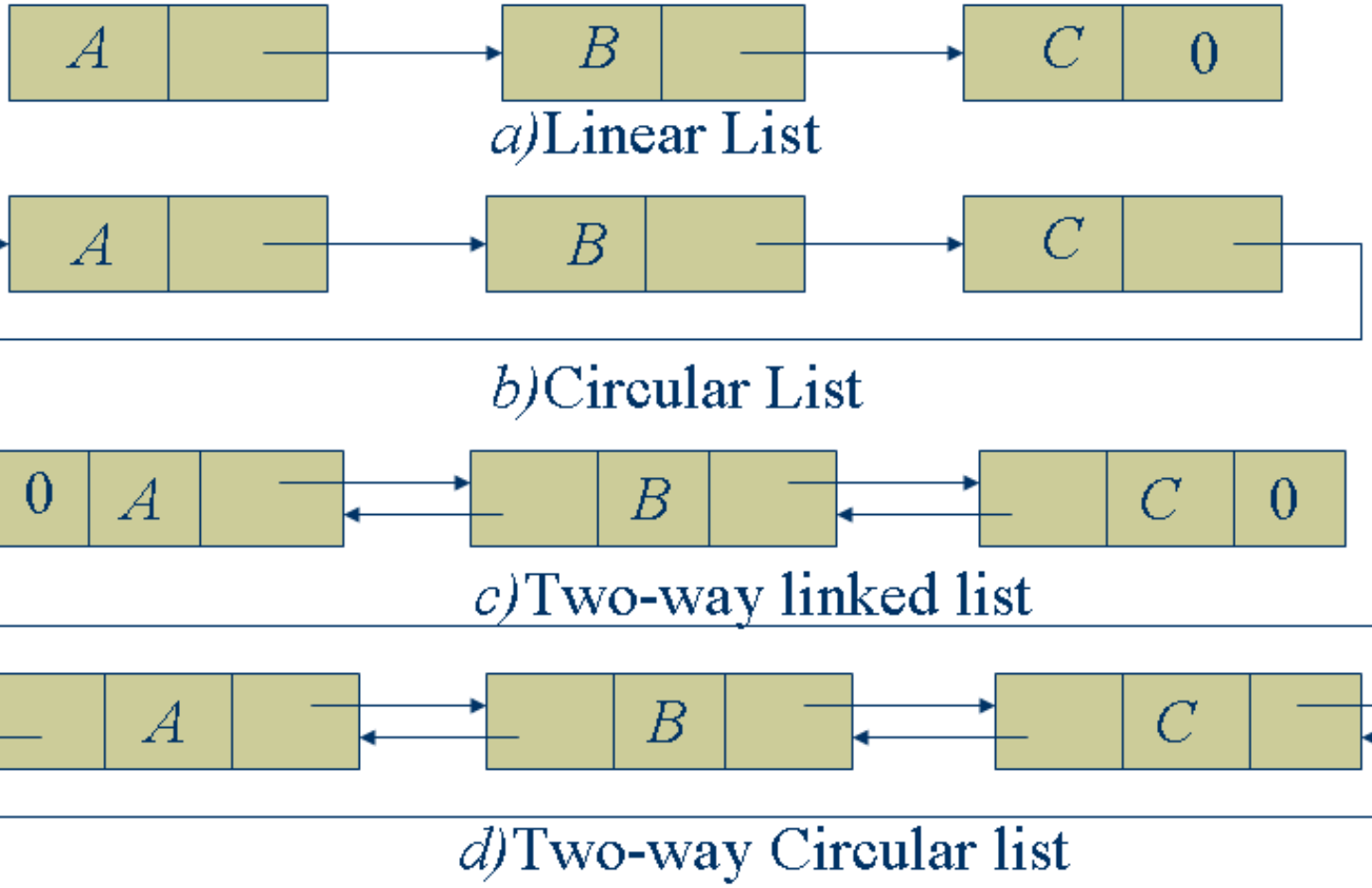
Types of Linked lists

- There are 4 types of Linked lists.
  - Linear singly Linked list.
  - Circular Linked list.
  - Two-way or doubly Linked list.
  - Circular doubly Linked list.
- The circular linked lists have no beginning and no end.The last item
- points back to the first item.It allows to traverse to any node.

- The doubly linked lists uses double set of pointers, one pointing to the
- next and other pointing to the preceding item.This allows us to traverse
- the list in either direction

- Circular doubly linked lists employ both the forward pointer and
- backward pointer in circular form.

Different types of Linked lists



Advantages and Disadvantages of Linked lists

- A linked list is a *dynamic data structure*. Linked lists can grow or shrink in size during the execution of a program.
- Linked list uses the memory that is just needed for the list at any point of time.(It is not necessary to specify the no of nodes to be used in the list).
- The linked lists provide flexibility in allowing the items to be rearranged efficiently.
- The major limitation is that the access to any arbitrary item is little cumbersome and time consuming.
- Linked list use more storage than an array with the same no of items.

Doubly Linked List

- The disadvantage with a singly linked list is that traversal is possible in only direction, i.e., from the beginning of the list till the end.
- If the value to be searched in a linked list is toward the end of the list, the search time would be higher in the case of a singly linked list.
- It would have been efficient had it been possible to search for a value in a linked list from the end of the list.

Properties of a Doubly Linked List

- This would be possible only if we have a doubly linked list.
- In a doubly linked list, each node has two pointers, one say, **next** pointing to the next node in the list, and another say, **prior** pointing to the previous node in the list.
- Therefore, traversing a doubly linked list in either direction is possible, **from the start to the end using next**, and **from the end of the list to the beginning of the list using prior**.

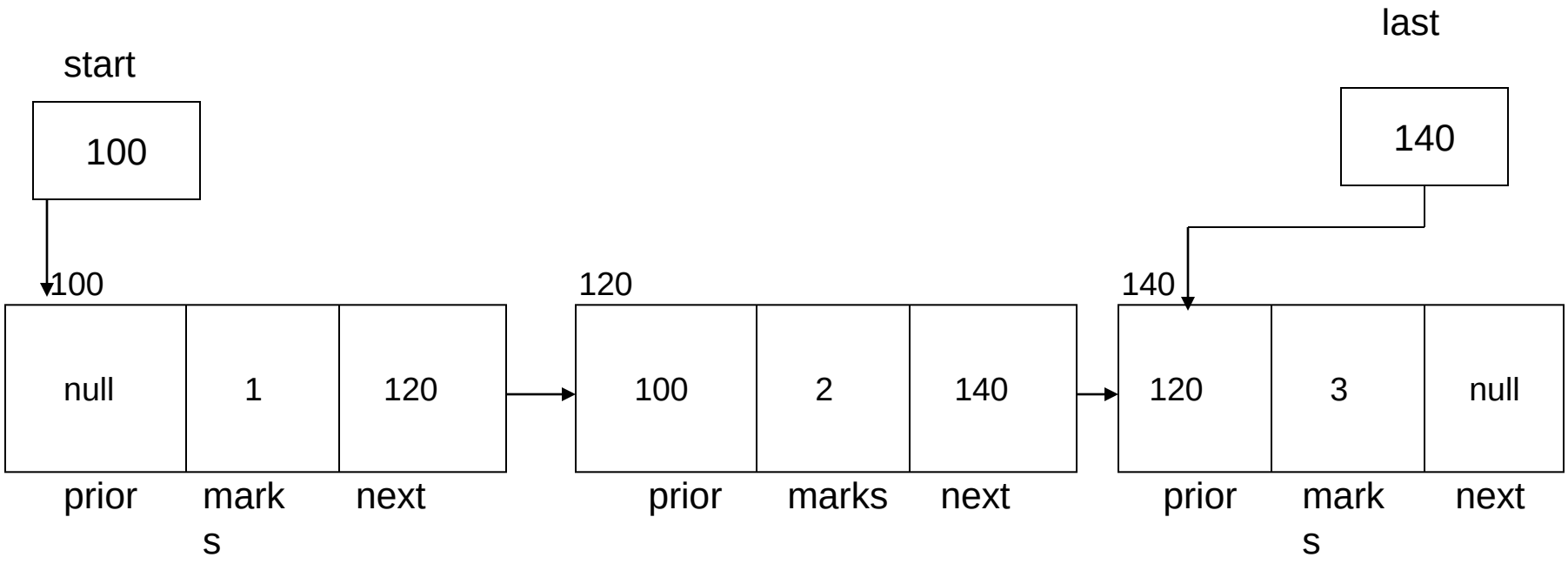
Properties of a Doubly Linked List (Contd.).

- In a doubly linked list, the prior pointer of the first node will be null, as there is no node before the first node.
- In a doubly linked list, the next pointer of the last node will be null, as there is no node after this list.
- Bidirectional traversal of a doubly linked list is useful for implementing page up, and page down functionality when using doubly linked lists to create editors.
- A doubly linked list would have two pointers, start and last to facilitate traversal from the beginning and end of the list respectively.

Declaration of a Doubly Linked List

```
struct marks_list
 {
   struct double_list *prior;
   int info;
   struct marks_list *next;
 }
```

Visualizing a Doubly Linked List



Creating a Sorted Doubly Linked List


```
#include<stdio.h>
#include<malloc.h>
struct marks_list *start, *last;
/* variables declared outside main() are global in nature and can be accessed by other functions called from main() */
struct marks_list
 {
  struct marks_list *prior;
  int marks;
  struct marks_list *next;
 };
main()
   {
    struct marks_list * makenode();
    /* function  prototype declaration */
```

Creating a Sorted Doubly Linked List (Contd.).

```
struct marks_list *new;
start = NULL;
char menu = '0 ';
while (menu != '5')
 {
   printf( "Add Nodes      :\n");
   printf( "Delete Nodes   :\n");
   printf( "Forward Traverse a list :\n");
   printf( "Reverse Traverse a list :\n");
   printf( "Exit                    :\n");
   menu = getchar( );
```

```
switch (menu)
 {
   case '1' : addnode( );
                break;
   case '2' : deletenode( )
                break;
   case '3' : forward_traverse( );
                break;
   case '4' : reverse_traverse( );
                break;
   case '5': exit( );
                break;
  } /* end of switch */
} /* end of main( ) */
```

Creating a Sorted Doubly Linked List (Contd.).

```
addnode( )
 {
   char ch = 'y';
    while ( ch = = 'y' )
     {     new = makenode();
      /* creation of a list is treated as a special case of
      insertion  */
      if ( start = = NULL)
       {        start = new;
        start->prior =  null;
       }
      else
       {        insert();
        traverse( );
       }
 printf("%s","Want to add more nodes\n");
  scanf( "%c", &ch );
  fflush( stdin );
 } /* end of while  */
} /* end of addnode( )
```

```
struct marks_list * makenode()
   { struct marks_list *new;
    new=(struct marks_list *)
     malloc(sizeof(struct(marks_list));
    scanf("%d",&new->marks);
    new->prior = NULL;
    new->next = NULL;
    return(new);
   }
insert( )
{     struct marks_list *ptr, *prev;
for(ptr=start,prev=start;(ptr);prev=ptr,ptr=ptr-
      >next)
 {    if (new->marks < start->marks)
   {  /* insertion at the beginning of a  list */
    new->next = start;
    new->prior = NULL;
    start->prior = new;
    last = start;
    start = new;
    }
```

Creating a Sorted Doubly Linked List (Contd.).

```
/* insertion in the middle of a list */
   if(new->marks  > ptr->marks)
         continue;
    else
    { prev->next = new;
     new->prior = prev;
     new->next = ptr;
     ptr->prior = new;
    }
} /* end of for loop */
/* insertion at the end of the list */
if (ptr = = null)
  {   prev->next = new;
   new->prior = prev;
   new->next = null;
   last = new;
  }
} /* end of insert */
```

Searching a Value in a Doubly Linked List

```
struct marks_list *search( int val)
{
  for( ptr = start; (ptr); ptr = ptr->next)
   {
     if (val = = ptr-> marks)
       return ptr;
   }

struct marks_list *search( int val)
{
  for( ptr = last; (ptr); ptr = ptr->prior)
   {
     if (val = = ptr-> marks)
       return ptr;
   }
```

Deleting a Node From a Doubly Linked List

```
delete ( )
 {
  struct marks_list *ptr, *prev, *temp;
  int score;
   /* search the linked list for the value to be deleted */
   scanf("%d", &score);
  fflush(stdin);
  for (ptr = start, prev = start; (ptr); prev = ptr, ptr = ptr->next)
  {
   /* deletion of the first node in the list */
   if (score = = start-> marks)
   {
    temp =start;
    start = start-> next;
    start->prior = null;
    free(temp);
   }
```

Deleting a Node From a Linked List

```
/* deletion in the middle of a linked list */
 if (score = = ptr-> marks)
    {
      prev-> next = ptr-> next;
      ptr->next->prior = ptr->prior;
      free(ptr);
     }
/* deletion at the end of the list */
 if (ptr->next = = null)
 {
   temp = ptr;
   prev->next = null;
   last = ptr->prior;
  }
} }
```

Traversal of a Doubly Linked List

```
/* forward traversal of a doubly linked list */
for( ptr = start; (ptr); ptr = ptr->next)
{
  printf("%d", ptr->marks);
}

/* reverse traversal of a doubly linked list */
for( ptr = last; (ptr); ptr = ptr->prior)
{
  printf("%d", ptr->marks);
}
```

Hands-on: 2 hours

- Purpose
- construct single and doubly linked list and perform insertion, deletion and append operations

Summary

- In this module, we discussed:

- Limitations of using a data structure such as an array
- Stack and heap variables
- Characteristics of stack and heap variables
- Creating a sorted linked list with the following operations
  - Insertion
  - Traversing
  - Deletion

Stacks & Queues

Module 9

Objectives

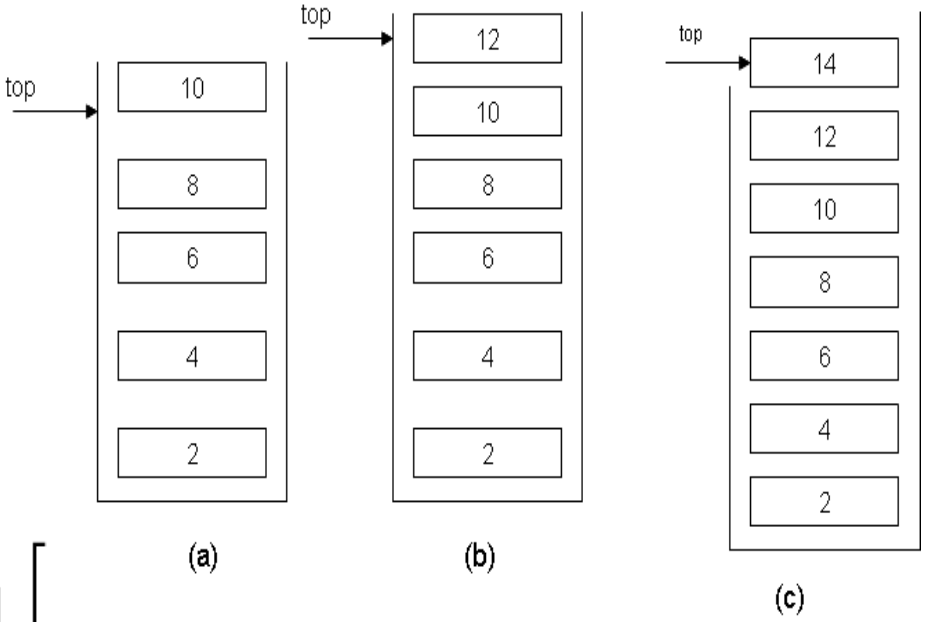
At the end of this module, you will be able to:

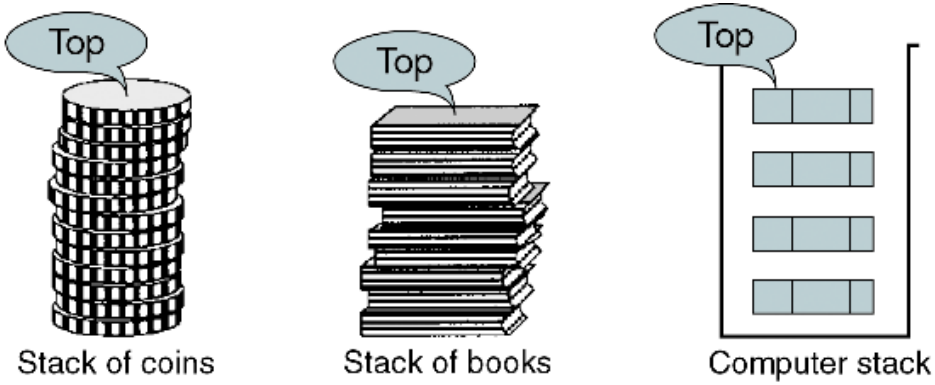
- Explain the design, use, and operation of a stack
- Implement a stack using a linked list structure
- Describe applications of stacks
- Discuss reversing data, parsing, postponing and backtracking
- Define a queue
- Describe the operations on a queue
- Implement a queue as a special case of a linked list
- Describe applications of queues

Duration: 4 hrs

What is a Stack?

- A stack is a linear list in which all additions and deletions are restricted to one end, called the top.
- Last in – first out (LIFO)





Operations on Stacks

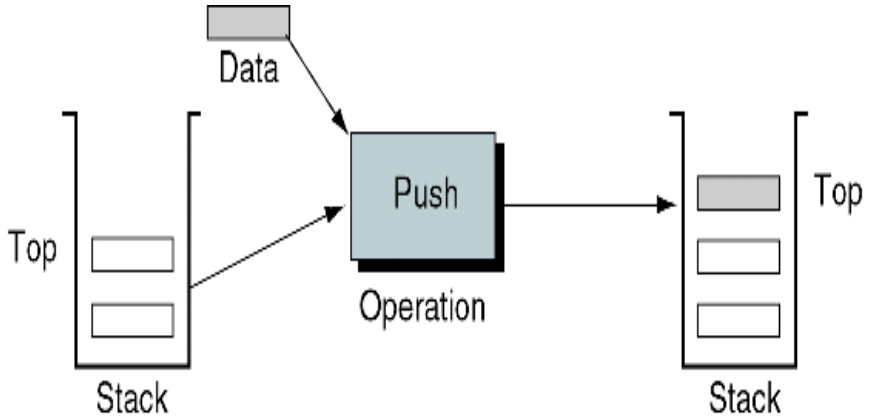
- Some of the typical operations performed on stacks are:

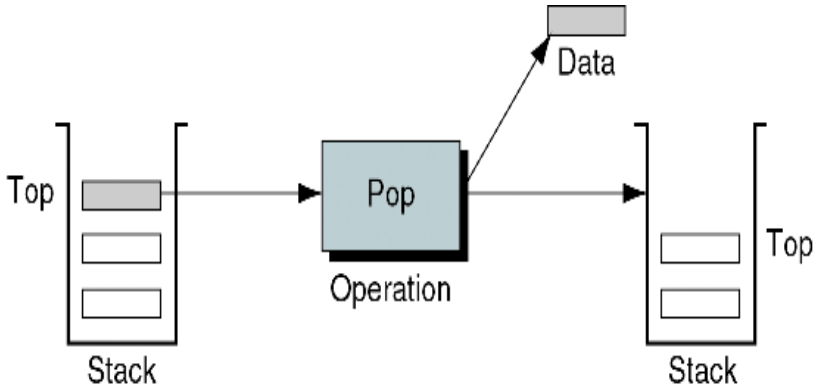
- create (s)
  - to create s an empty stack
- push (s, i)
  - to insert element i on top of the stack
- pop (s)
  - to remove top element of the stack
- top (s)
  - to return top element of stack
- empty(s)
  - to check whether the stack is empty or not
- Others
  - Full stack, Stack count, Destroy stack

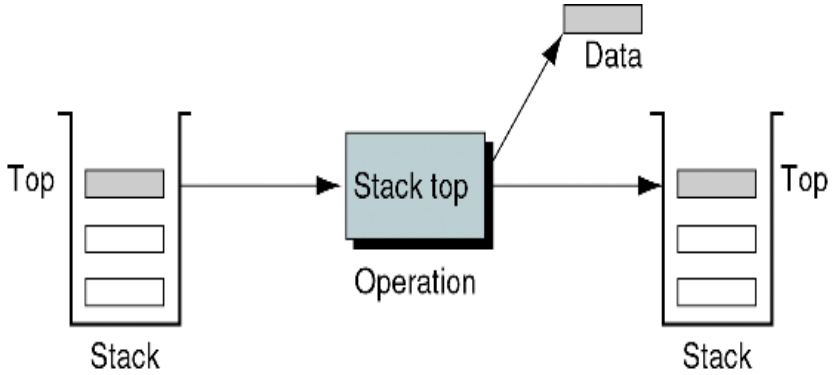
Implementation of Stacks

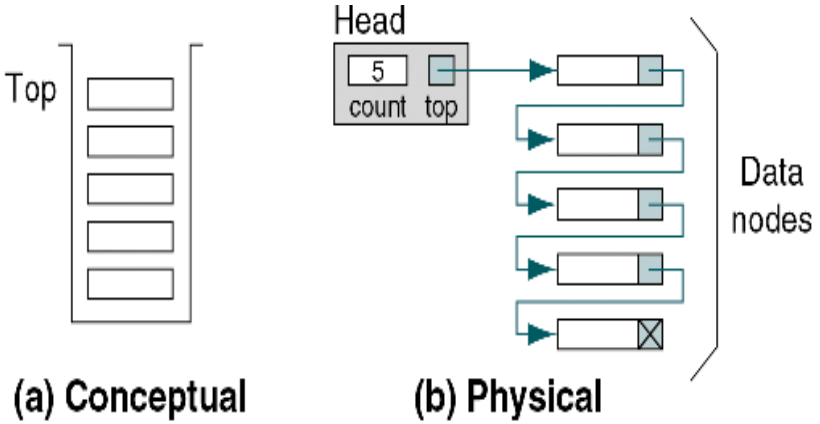
- An array can be used to implement a stack.
- An ideal implementation for a stack is a linked list that can dynamically grow and shrink at runtime.
- A stack, by definition, is a data structure that cannot be full since it can dynamically grow and shrink at runtime.

Push, Pop, Stack top

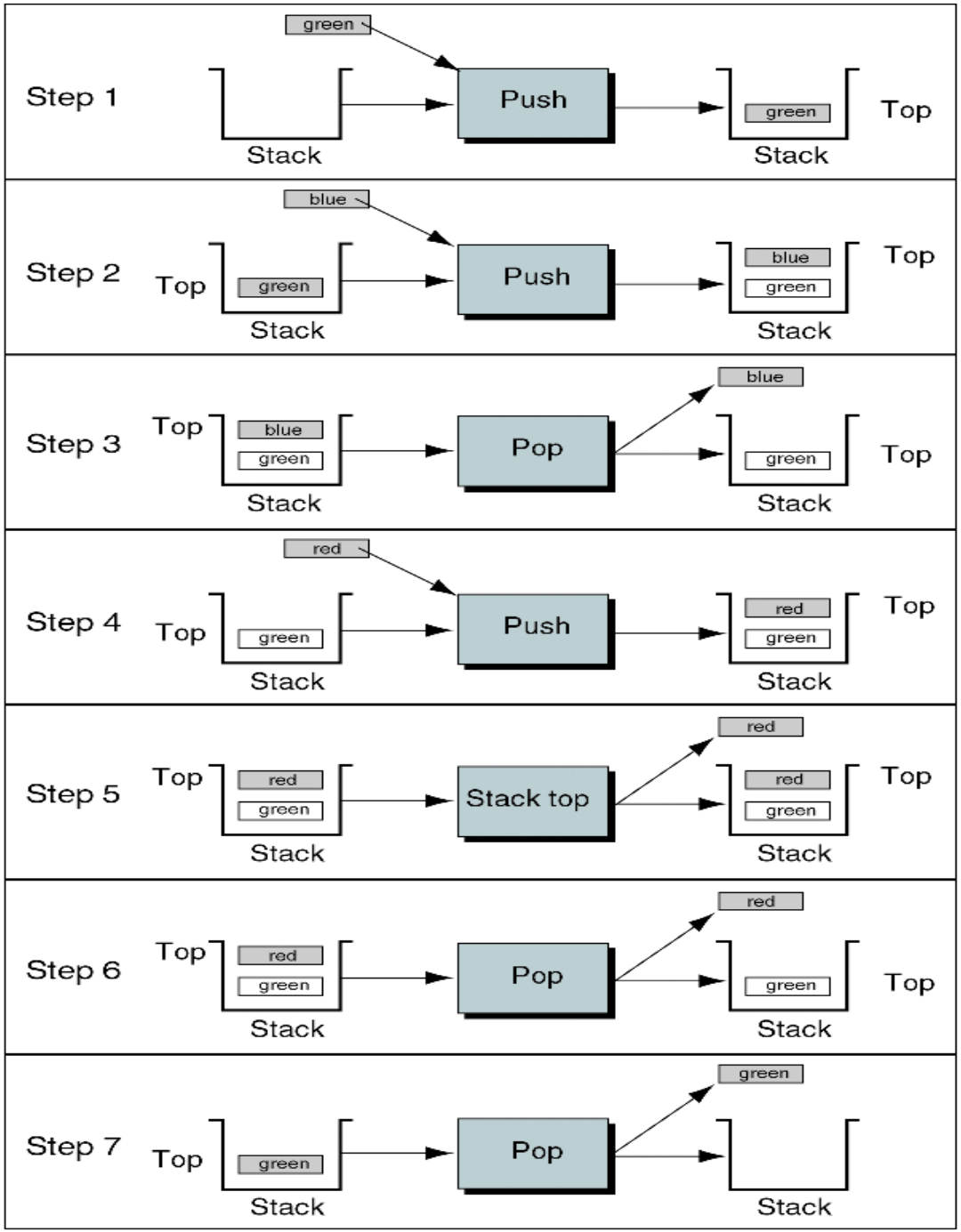




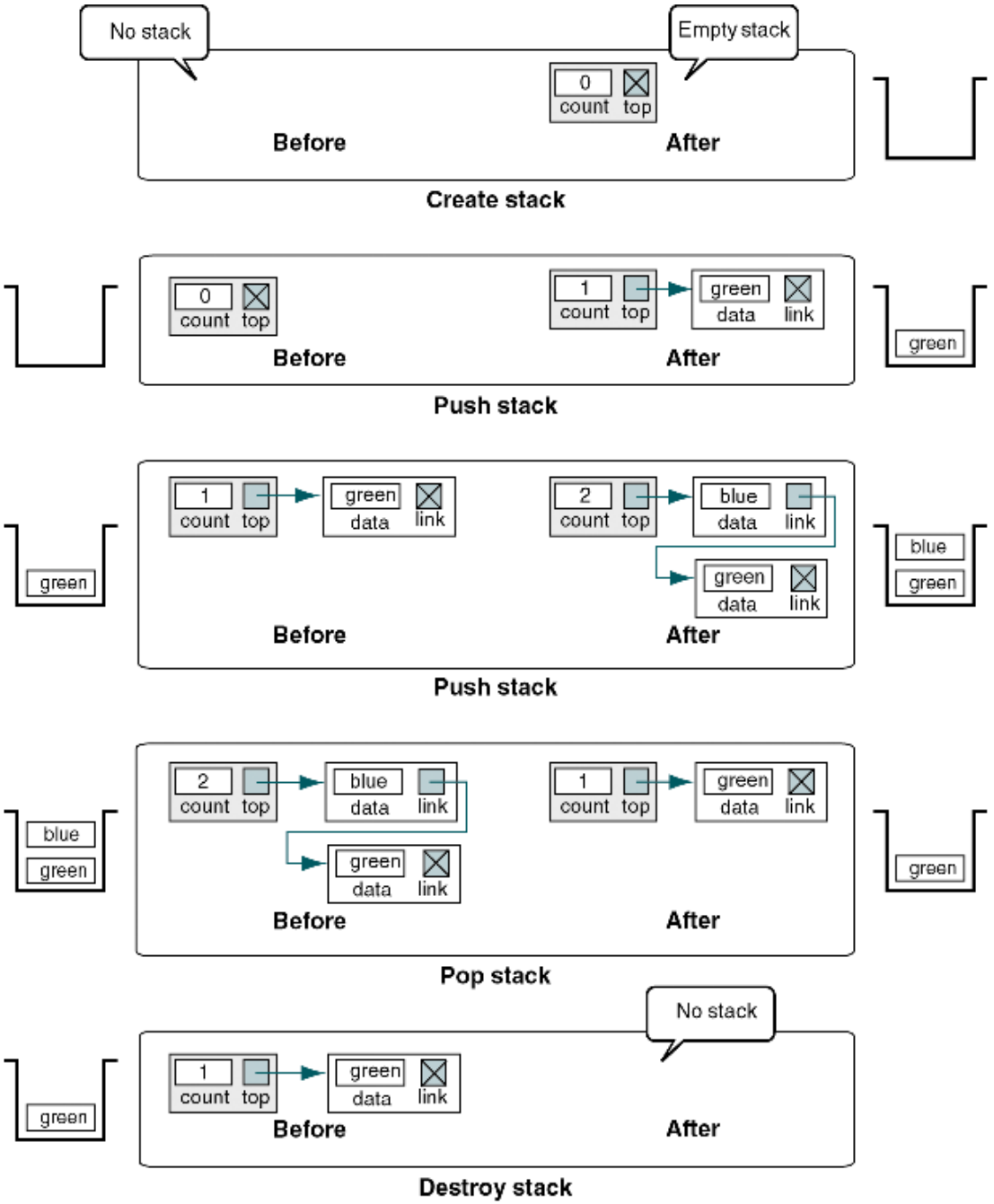




Example



Example (Contd.).



Code Implementing for a Stack

- The push( ) and the pop( ) operations on a stack are analogous to insert-first-node and delete-first-node operations on a linked list that functions as a stack.

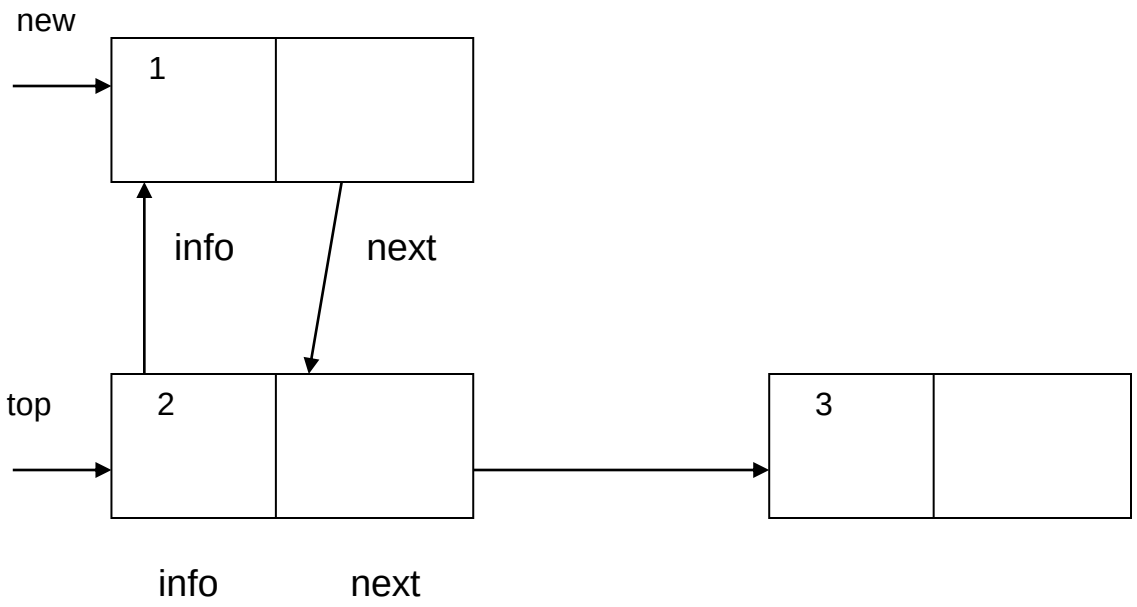
```
struct stack
 {  int info;
  struct stack *next;
};
/* pointer declaration to point to
     the top of the stack */
struct stack *top;
```

```
main( )
{
top = NULL;
char menu = '0 ';
while (menu != '3')
  {
    printf( "Add Nodes     :\n");
    printf( "Delete Nodes  :\n");
    printf( "Exit          :\n");
    menu = getchar( );
   switch (menu)
 {   case '1' : push( );        break;
   case '2' : pop( )  ;         break;
 case '3': exit( );             break;
} /* end of switch */
   } /*
```

Implementing push( )

```
push( )
{    struct stack * new;
     char ch;
     ch = 'y';
  while (ch = = 'y')
   { new = getnode( )
     /* checking for an empty stack */
     if ( top = = null)
     top = new;
     else   { new->next = top;
       top = new;
             }
  printf("%s","Want to add more nodes\n");
  scanf( "%c", &ch );
  fflush( stdin );
 } /* end of while  */
} /* end of push( )
```

A View of the Stack After Insertion



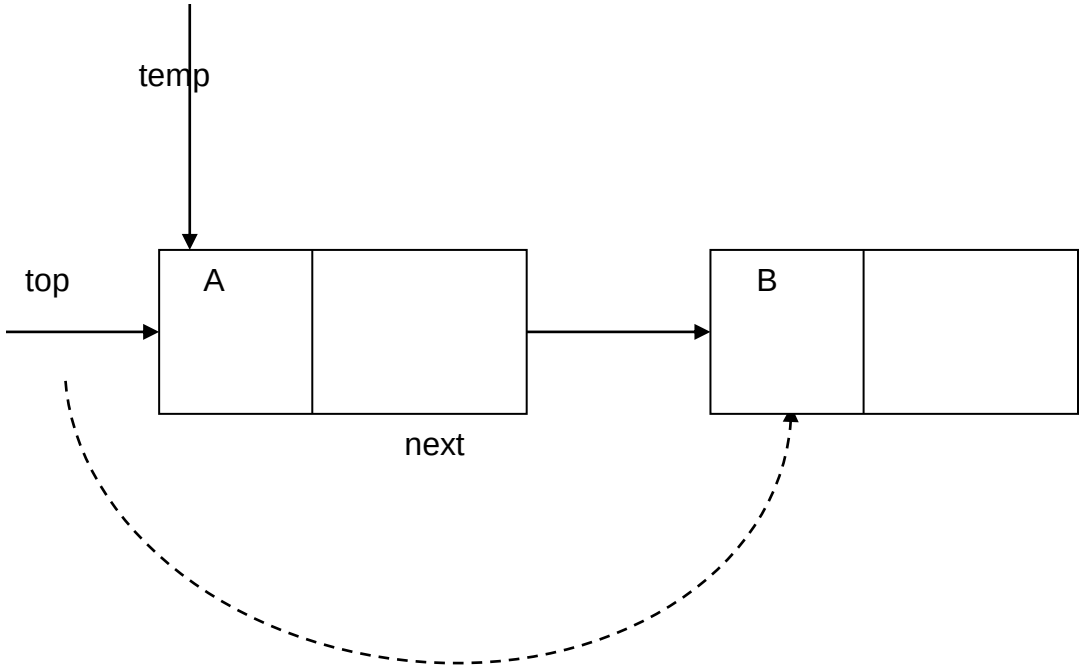
Creating a Node on a Stack

```
struct stack * makenode()
   {
    struct stack *new;
    new=(struct stack *)  malloc(sizeof(struct(stack));
    scanf("%d",&new->info);
    new->next = NULL;
    return(new);
   }
```

Implementing pop( )

```
pop( )
{    struct stack * temp;
  int x;
  /* check for an empty stack */
  if (top = = null)
    {
      printf ("Cannot remove nodes from an empty stack */
      exit( );
     }
else
 {   temp = top;
     x = top->info;
     top = top->next;
     free( temp);
     return x;
  }
```

A View of the Stack After Deletion



Applications of Stacks

- As a stack is a LIFO structure, it is an appropriate data structure for applications in which information must be saved and later retrieved in reverse order.
- Consider what happens within a computer when function main( ) calls another function.
- How does a program remember where to resume execution from after returning from a function call?
- From where does it pick up the values of the local variables in the function main( ) after returning from the subprogram?

Applications of Stacks (Contd.).

- As an example, let main( ) call a( ). Function a( ), in turn, calls function b( ), and function b( ) in turn invokes function c( ).
- main( ) is the first one to execute, but it is the last one to finish, after a( ) has finished and returned.
- a( ) cannot finish its work until b( ) has finished and returned. b( ) cannot finish its work until c( ) has finished and returned.

Applications of Stacks (Contd.).

- When **a( )** is called, its calling information is pushed on to the stack (calling information consists of the address of the return instruction in **main( )** after **a( )** was called, and the local variables and parameter declarations in **main( )**.

- When **b( )** is called from **a( )**, b( )'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **a( )** after **b( )** was called, and the local variables and parameter declarations in **a( )**).

Applications of Stacks (Contd.).

- Then, when **b( )** calls **c( )**, **c( )**'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **b( )** after **c( )** was called, and the local variables and parameter declarations in **b( )**).

- When **c( )** finishes execution, the information needed to return to **b( )** is retrieved by popping the stack.

- Then, when **b( )** finishes execution, its return address is popped from the stack to return to **a( )**

Applications of Stacks (Contd.).

- Finally, when **a( )** completes, the stack is again popped to get back to **main( )**.
- When **main( )** finishes, the stack becomes empty.
- Thus, a stack plays an important role in function calls.
- The same technique is used in recursion when a function invokes itself.

Defining a Queue

- A Queue is a data structure in which elements are added at one end (called the **rear**), and elements are removed from the other end (called the **front**).

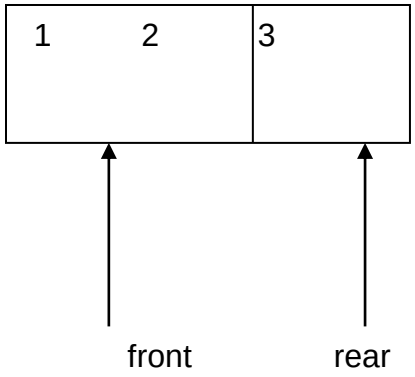
- You come across a number of examples of a queue in real life situations.

- For example, consider a line of students at a fee counter. Whenever a student enters the queue, he stands at the end of the queue (analogous to the addition of nodes to the queue)

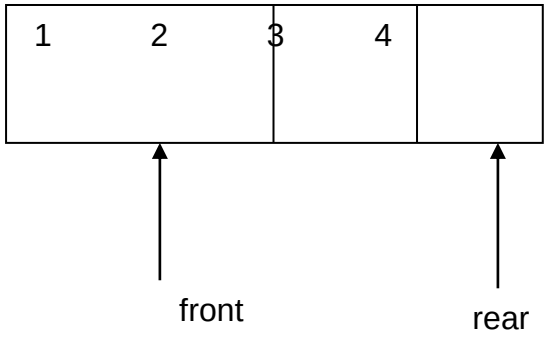
Defining a Queue (Contd.).

- Every time the student at the front of the queue deposits the fee, he leaves the queue (analogous to deleting nodes from a queue).
- The student who comes first in the queue is the one who leaves the queue first.
- **Therefore, a queue is commonly called a first-in-first-out or a FIFO data structure**.

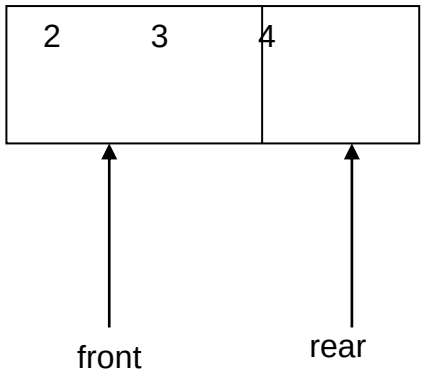
Queue Insertions and Deletions



(a)



(b)



(c)

Queue Operations

- To complete this definition of a queue, you must specify all the operations that it permits.

- The first step you must perform in working with any queue is to initialize the queue for further use. Other important operations are to add an element, and to delete an element from a queue.

- Adding an element is popularly known as ENQ and deleting an element is know as DEQ. The following slide lists operations typically performed on a queue.

Queue Operations (Contd.).

- create(q) – which creates q as an empty queue
- enq(i) – adds the element i to the rear of the queue and returns the new queue
- deq(q) – removes the element at the front end of the queue (q) and returns the resulting queue as well as the removed element
- empty (q) – it checks the queue (q) whether it is empty or not. It returns true if the queue is empty and returns false otherwise
- front(q) – returns the front element of the queue without changing the queue
- queuesize (q) – returns the number of entries in the queue

Implementing Queues

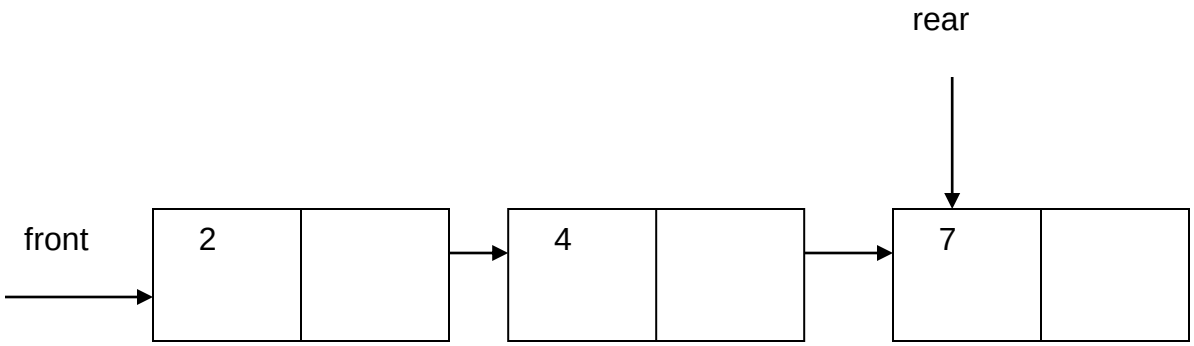
- Linked lists offer a flexible implementation of a queue since insertion and deletion of elements into a list are simple, and a linked list has the advantage of dynamically growing or shrinking at runtime.

- Having a list that has the functionality of a queue implies that insertions to the list can only be done at the rear of the list, and deletions to the list can only be done at front of the list.

Implementing Queues (Contd.).

- Queue functionality of a linked list can be achieved by having two pointers **front** and **rear**, pointing to the first element, and the last element of the queue respectively.

- The following figure gives a visual depiction of linked list implementation of a queue.



Queue Declaration & Operations

- struct queue
- {
- int info;
- struct queue *next;
- };
- struct queue *front, *rear;

- An empty queue is represented by q->front = q->rear = null. Therefore, clearq( ) can be implemented as follows:

- void clearq(struct queue * queue_pointer)
- {
- queue_pointer->front = queue_pointer ->rear = null;
- }

Queue Operations

- You can determine whether a queue is empty or not by checking its front pointer. The front pointer of a queue can be passed as an argument to emptyq( ) to determine whether it is empty or not.

```
int emptyq (queue_pointer)
 {
   if (queue_pointer = = null)
    return (1);
   else
    return(0);
}
```

Insertion into a Queue

```
struct queue
 {  int info;
   struct queue *next;
};
/* pointer declarations to point to the front, and rear of the queue */
struct queue *front, *rear;
main( )
{
  front = NULL;
   rear = NULL;
```

Insertion into a Queue (Contd.).

```
char menu = '0 ';
while (menu != '3')
  {
    printf( "Add Nodes     :\n");
    printf( "Delete Nodes  :\n");
    printf( "Exit          :\n");
    menu = getchar( );
   switch (menu)
 {
   case '1' : enq( );
                break;
   case '2' : deq( )
                break;
case '3': exit( );
            break;
} /* end of switch */
   } /* end of main( ) */
```

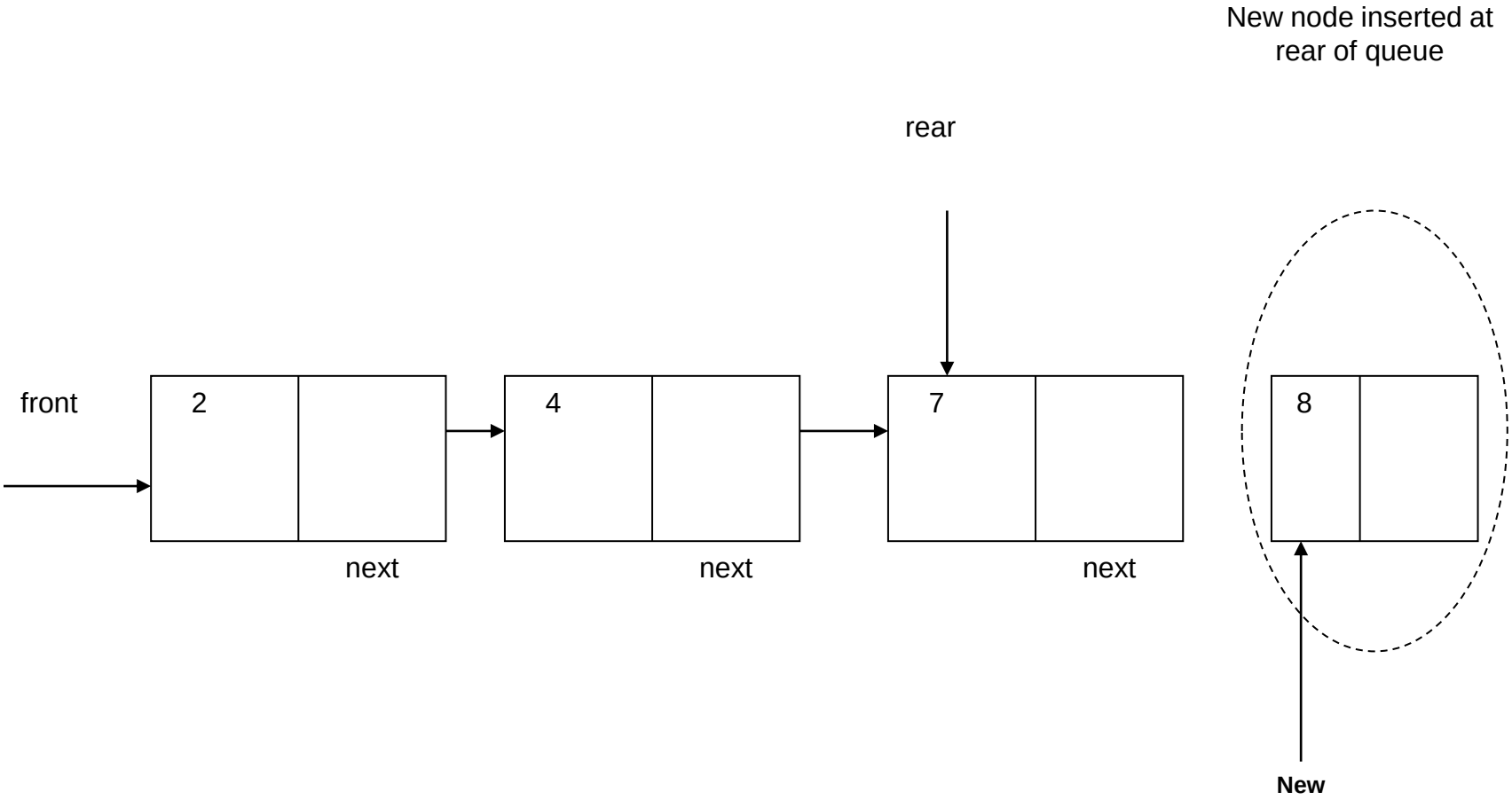
Insertion into a Queue (Contd.).

```
void enq( )
    {
       struct queue *new;
       new = getnode( );
      if(queue_pointer->front= =queue_pointer->rear = = null)
       {
         queue_pointer->front = new;
         queue_pointer->rear = new;
       }
       else
       {
         rear->next = new;
         rear = new;
       }
    }
```

Creating a Node on a Queue

```
struct queue * makenode()
   {
    struct queue *new;
    new=(struct queue *)  malloc(sizeof(struct(queue));
    scanf("%d",&new->info);
    new->next = NULL;
    return(new);
   }
```

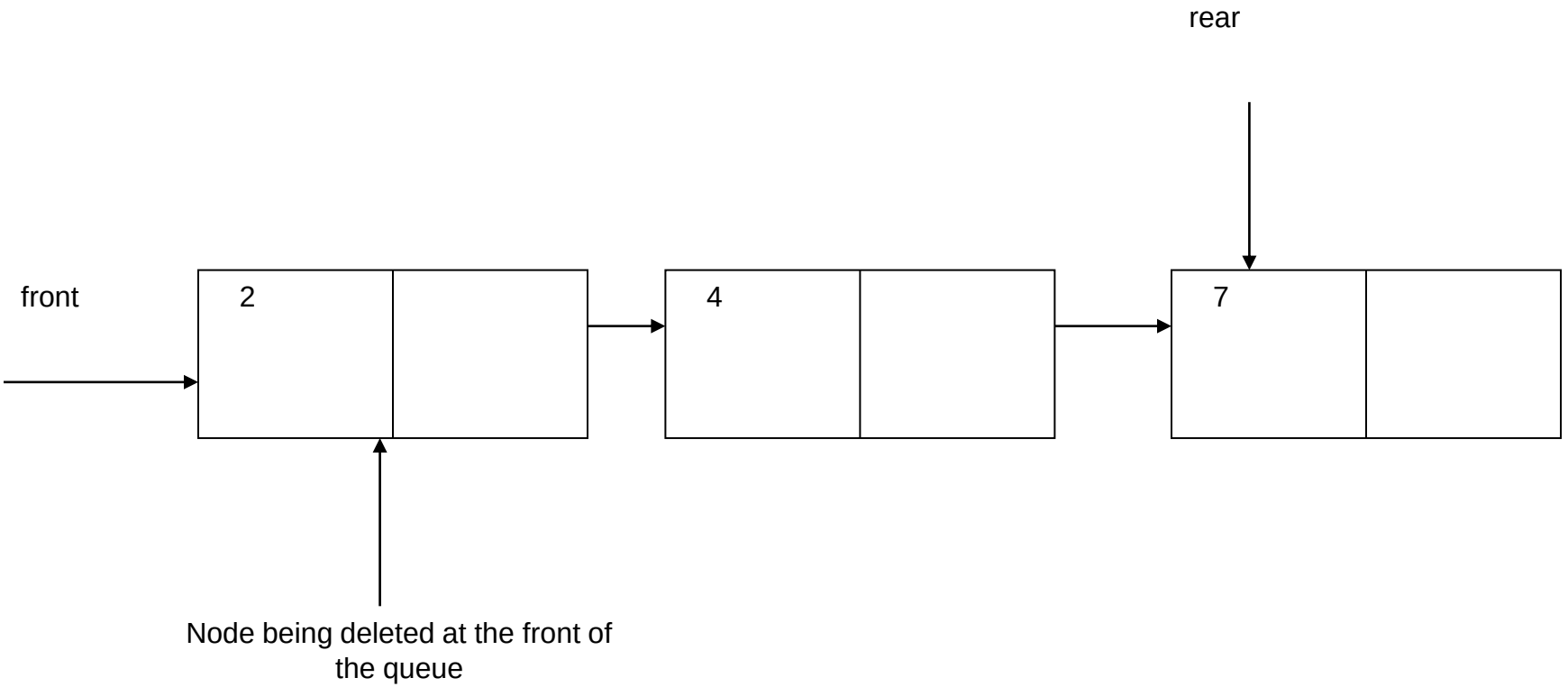
Insertion into a Queue



Deletion from a Queue

```
int deq( )
 {
    struct queue *temp;
    int x;
    if(queue_pointer->front= =queue_pointer->rear = = null)
     {
       printf( "Queue Underflow\n");
       exit (1);
      }
     temp = front;
     x=temp->info;
     front = front->next;
     free(temp);
    if(front = = null) /* check for queue becoming empty after node deletion */
     rear = null;
    return(x);
 }
```

Deletion of a Node From a Queue



Applications of Queues

- Queues are also very useful in a time-sharing multi-user operating system where many users share the CPU simultaneously.

- Whenever a user requests the CPU to run a particular program, the operating system adds the request (by first of all converting the program into a process that is a running instance of the program, and assigning the process an ID).

- **This process ID is then added at the end of the queue of jobs waiting to be executed.**

Applications of Queues (Contd.).

- Whenever the CPU is free, it executes the job that is at the front of the job queue.
- Similarly, there are queues for shared I/O devices. Each device maintains its own queue of requests.
- An example is a print queue on a network printer, which queues up print jobs issued by various users on the network.
- The first print request is the first one to be processed. New print requests are added at the end of the queue.

Hands-on: 2 hours

- Purpose
- construct stack and queue so that to perform related operations

Summary

In this module, we discussed:

- Stack and operations on stack
- Stack as a special case of a linked list
- Applications of stacks
- Queue and operations on queue
- Queue as a special case of a linked list
- Applications of queues

Binary Trees

Module 10

Objectives

At the end of this module, you will be able to:

- Define a binary tree
- Describe the terminologies associated with a binary tree
- Use a dynamically allocated data structure to represent a binary tree
- Traverse a binary tree
- Add nodes to a binary tree
- Remove nodes from a binary tree
- Search a binary tree

Duration: 5 hrs

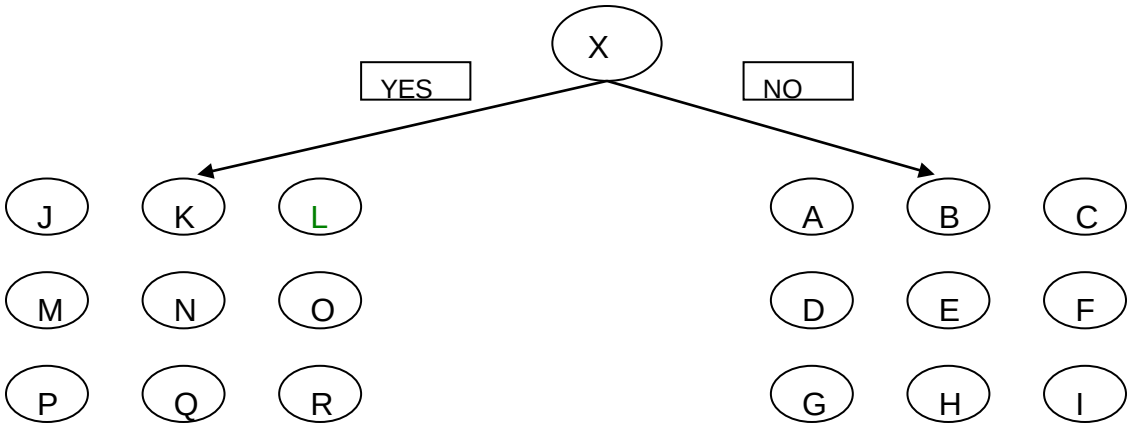
Eliminative or a Binary Search

- The mode of accessing data in a linked list is linear. Therefore, search through a linked list is always linear.
- A linear search is fine, if the nodes to be searched in a linked list are small in number.
- The linear search becomes ineffective as the number of nodes in a linked list increase.
- The search time increases in direct proportion with the size of the linked list.

Example - Binary Search

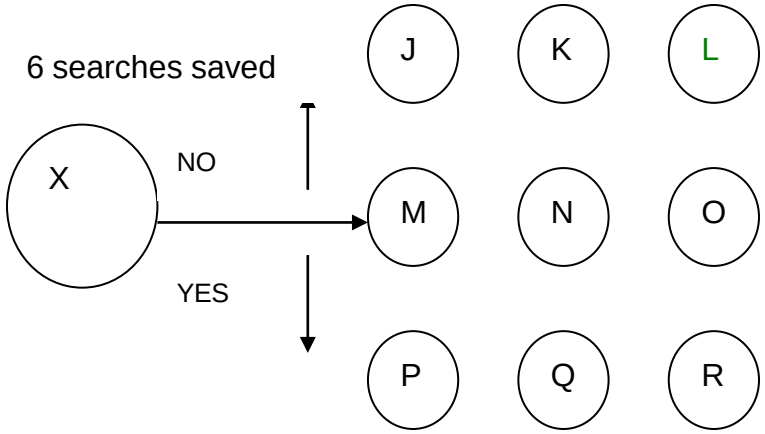
- Game that will highlight a new mechanism of searching
- A coin is with one of the members in the audience divided into the two sections on the left and the right respectively as shown in the diagram.
- The challenge facing X, the protagonist, is to find the person with the coin in the least number of searches.

Coin Search in a Group



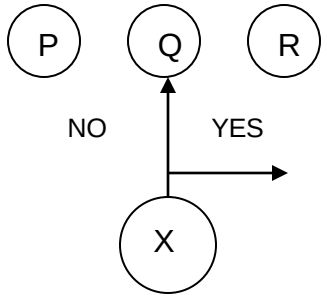
Example - Binary Search (Contd.).

- X has in the process now eliminated 6 more searches, that is, the middle row and the row to the left of the middle as shown in the diagram below.



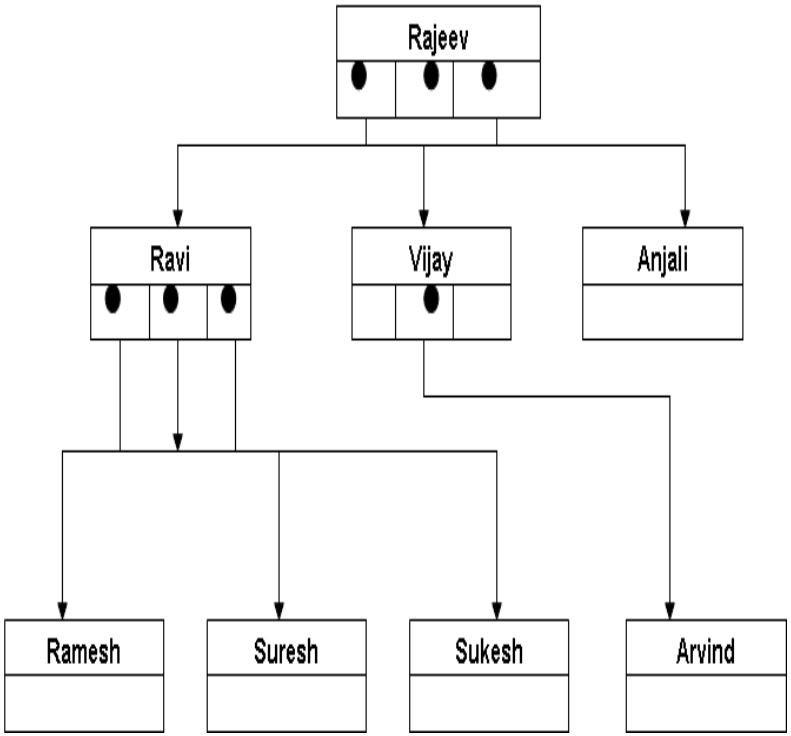
Example - Binary Search (Contd.).

- X is now left with row to the right of the middle row containing P, Q, and R that has to be searched.
- Here too, he can position himself right at the middle of the row adjacent to Q and pose the same question,
- "Which side of the audience has the coin?" 'Q' replies that the coin is to his left.



Trees

- Trees are non-linear data structures.
  - In a tree structure, each node may point to several nodes
- Tree are very flexible and a powerful data structure
  - Used for a wide variety of applications.
- A tree is a collection of nodes that are connected to each other.
  - contains a unique first element known as the root, which is shown at the top of the tree structure.
  - A node which points to other nodes is said to be the parent of the nodes
  - the nodes that the parent node points to are called the children, or child nodes of the parent node.



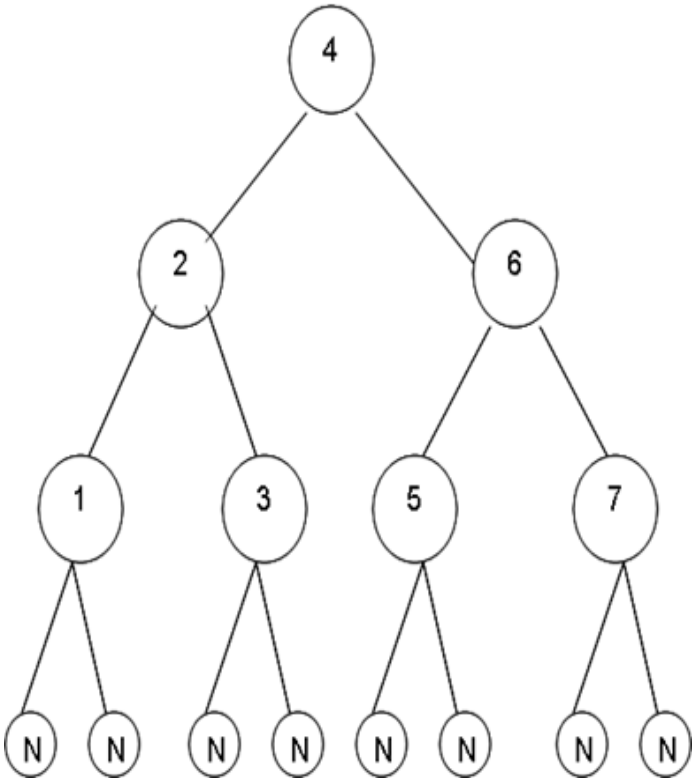
Tree (Contd.).

- An important feature of a tree is that there is a single unique path from the root to any particular node.
- The length of the longest path from the root to any node is known as the depth of the tree.
- The root is at level 0 and the level of any node in the tree is one more than the level of its parent.
- In a tree, any node can be considered to be a root of the tree formed by considering only the descendants of that node. Such a tree is called the subtree that itself is a tree.

Binary Tree

- A complete binary tree can be defined as one whose non-leaf nodes have non-empty left and right subtrees and all leaves are at the same level.
- Each node will have a maximum of two children or two child nodes.
- If a binary tree has the property that
  - all elements in the left subtree of a node n are less than the contents of n, and
  - all elements in the right subtree are greater than the contents of n

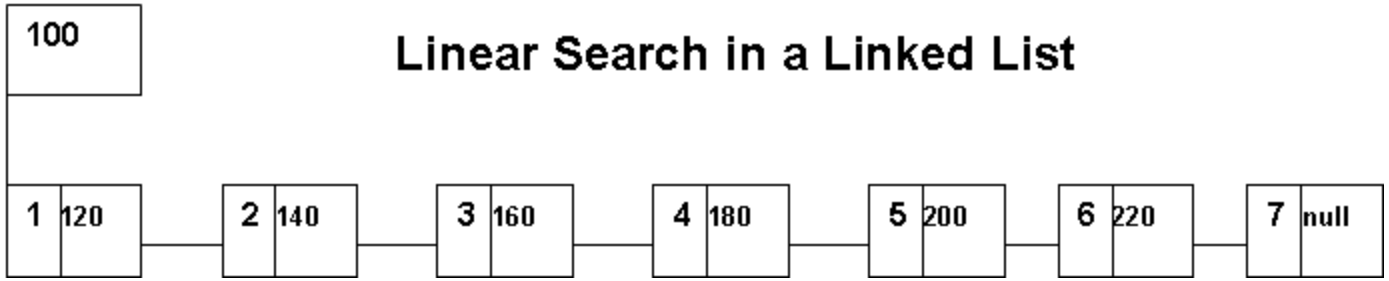
The following is an example of a balanced binary search tree.

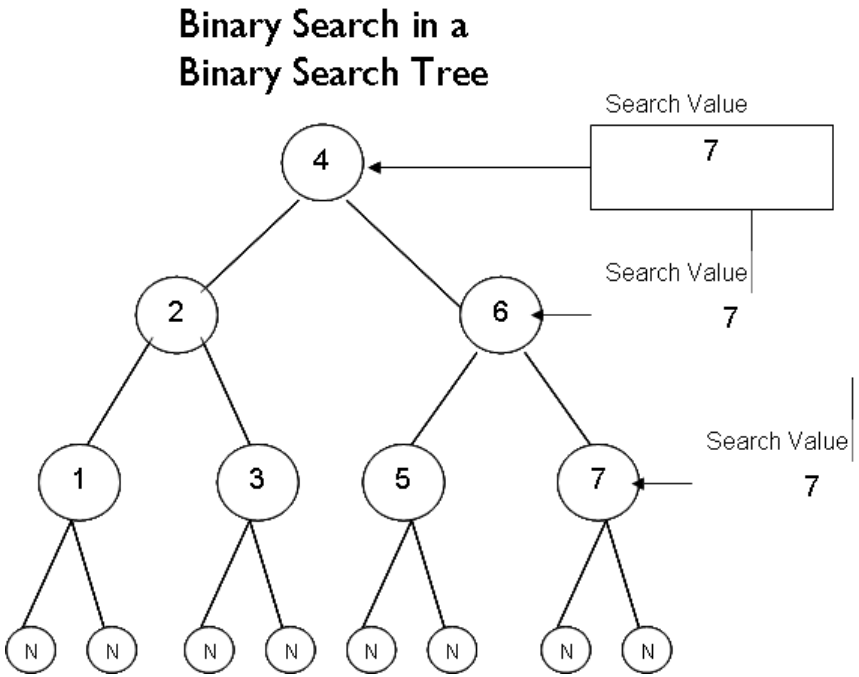


Binary Vs. Linear Search

- As the name suggests, balanced binary search trees are very useful for searching an element just as with a binary search.
- If we use linked lists for searching, we have to move through the list linearly, one node at a time.
- If we search an element in a binary search tree, we move to the left subtree for smaller values, and to the right subtree for larger values, every time reducing the search list by half approximately.

Binary Vs. Linear Search (Contd.).





The Essence of a Binary Search

- To summarize, you have completely done away with searching with the entire left subtree of the root node and its descendant subtrees, in the process doing away with searching one-half of the binary search tree.

- Even while searching the right subtree of the root node and its descendant subtrees, we keep searching only one-half of the right subtree and its descendants.

- This is more because of the search value in particular, which is 7. The left subtree of the right subtree of the root could have been searched in case the value being searched for was say 5.

The Essence of a Binary Search (Contd.).

- Thus we can conclude that while searching for a value in a balanced binary search tree, the number of searches is cut by more than half (3 searches in a balanced binary search tree) compared to searching in a linked list (7 searches).

- **Thus a search that is hierarchical, eliminative and binary in nature is far efficient when compared to a linear search.**

Data Structure Representation of a Binary Trees

- A tree node may be implemented through a structure declaration whose elements consist of a variable for holding the information and also consist of two pointers, one pointing to the left subtree and the other pointing to the right subtree.

- A binary tree can also be looked at as a special case of a doubly linked list that is traversed hierarchically.

- The following is the structure declaration for a tree node:

```
struct btreenode
 {
   int info;
   struct btreenode *left;
   struct btreenode *right;
};
```

Traversing a Binary Tree

- Traversing a binary tree entails visiting each node in the tree exactly once.
- Binary tree traversal is useful in many applications, especially those involving an indexed search.
- Nodes of a binary search tree are traversed hierarchically.
- The methods of traversing a binary search tree differ primarily in the order in which they visit the nodes.

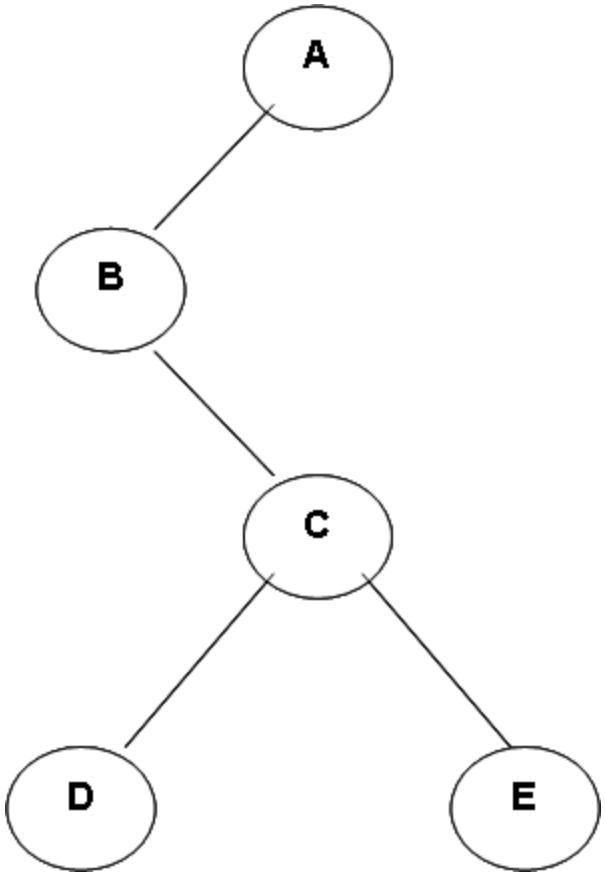
Traversing a Binary Tree (Contd.).

- At a given node, there are three things to do in some order. They are:
  - To visit the node itself
  - To traverse its left subtree
  - To traverse its right subtree

- If we designate the task of visiting the root as R', traversing the left subtree as L and traversing the right subtree as R, then the three modes of tree traversal discussed earlier would be represented as:
  - R'LR – Preorder
  - LRR' – Postorder
  - LR'R – Inorder

Traversing a Binary Tree (Contd.).

- Depending on the position at which the given node or the root is visited, the name is given.
- If the root is visited *before* traversing the subtree, it is called the preorder traversal.
- If the root is visited *after* traversing the subtrees, it is called postorder traversal.
- If the root is visited in *between* the subtrees, it is called the inorder traversal.



Preorder Traversal

- When we traverse the tree in preorder, the root node is visited first.
  - i.e., the node containing A is traversed first.
- Next, we traverse the left subtree.
  - This subtree must again be traversed using the preorder method.
- Therefore, we visit the root of the subtree containing B and then traverse its left subtree.
- The left subtree of B is empty
  - so its traversal does nothing.
  - Next we traverse the right subtree that has root labeled C.

Preorder Traversal (Contd.).

- Then, we traverse the left and right subtrees of C getting D and E as a result.
- Now, we have traversed the left subtree of the root containing A completely, so we move to traverse the right subtree of A.
- The right subtree of A is empty, so its traversal does nothing. Thus the preorder traversal of the binary tree results in the values **ABCDE**.

Inorder Traversal

- For inorder traversal, we begin with the left subtree rooted at B of the root.
- Before we visit the root of the left subtree, we must visit its left subtree, which is empty.
- Hence the root of the left subtree rooted at B is visited first. Next, the right subtree of this node is traversed inorder.

Inorder Traversal (Contd.).

- Again, first its left subtree containing only one node D is visited, then its root C is visited, and finally the right subtree of C that contains only one node E is visited.

- After completing the left subtree of root A, we must visit the root A, and then traverse its right subtree, which is empty.

- Thus, the complete inorder traversal of the binary tree results in values **BDCEA**.

Postorder Traversal

- For postorder traversal, we must traverse both the left and the right subtrees of each node before visiting the node itself.
- Hence, we traverse the left subtree in postorder yielding values D, E, C and B.
- Then we traverse the empty right subtree of root A, and finally we visit the root which is always the last node to be visited in a postorder traversal.
- Thus, the complete postorder traversal of the tree results in **DECBA**.

Code - Preorder Traversal

```
void preorder (p)
struct btreenode *p;
 {
  /* Checking for an empty tree */
  if ( p != null)
   {
    /* print the value of the root node */
    printf("%d", p->info);
    /* traverse its left subtree */
    preorder(p->left);
    /* traverse its right subtree */
    preorder(p->right);
   }
 }
```

Code – Inorder Traversal

```
void inorder(p)
struct btreenode *p;
 {
  /* checking for an empty tree */
  if (p != null)
   {
    /* traverse the left subtree inorder */
    inorder(p->left);
    /* print the value of the root node */
    printf("%d", p->info);
    /*traverse the right subtree inorder */
    inorder(p->right);
   }
 }
```

Code – Postorder Traversal

```
void postorder(p)
struct btreenode *p;
 {
  /* checking for an empty tree */
  if (p != null)
   {
    /* traverse the left subtree */
    postorder(p->left);
    /* traverse the right subtree */
    postorder(p->right);
    /* print the value of the root node */
    printf("%d", p->info);
   }
 }
```

Accessing Values From a Binary Search Tree Using Inorder Traversal

- You may note that when you traverse a binary search tree inorder, the keys will be in sorted order because all the keys in the left subtree are less than the key in the root, and all the keys in the right subtree are greater than that in the root.

- The same rule applies to all the subtrees until they have only one key.

- Therefore, given the entries, we can build them into a binary search tree and use inorder traversal to get them in sorted order.

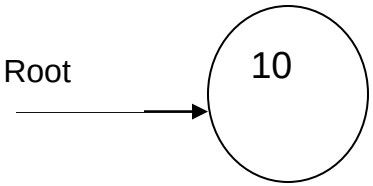
Insertion into a Tree

- Another important operation is to create and maintain a binary search tree.
- While inserting any node, we have to take care the resulting tree satisfies the properties of a binary search tree.
- A new node will always be inserted at its proper position in the binary search tree as a leaf.
- Before writing a routine for inserting a node, consider how a binary tree may be created for the following input: 10, 15, 12, 7, 8, 18, 6, 20.

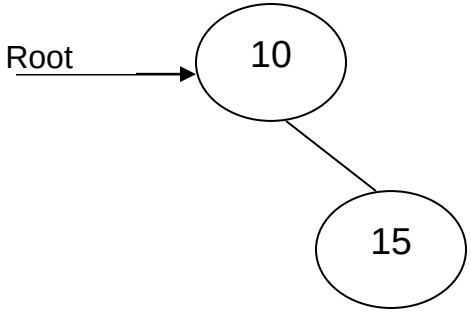
Insertion into a Tree (Contd.).

- First of all, you must initialize the tree.

- To create an empty tree, you must initialize the root to null. The first node will be inserted into the tree as a root node as shown in the following figure.

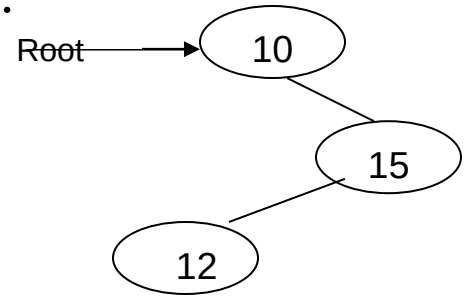


- Since 15 is greater than 10, it must be inserted as the right child of the root as shown in the following figure.

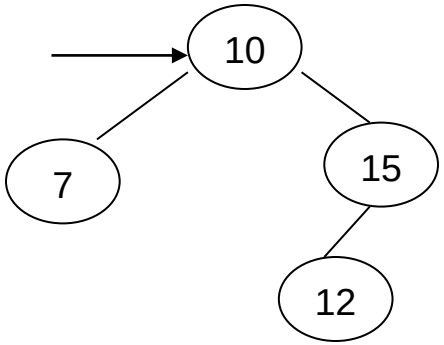


Insertion into a Tree (Contd.).

- Now 12 is larger than the root; it must go to the right subtree of the root.
- Further, since it is smaller than 15, it must be inserted as the left child of the root as shown below.

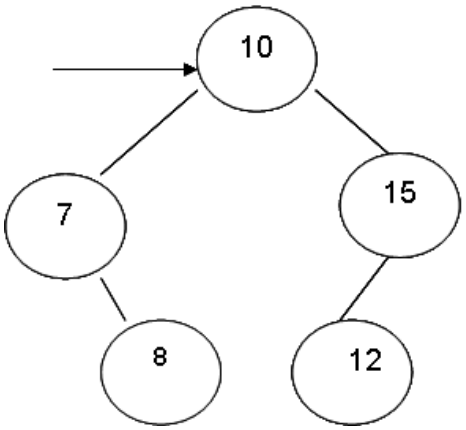


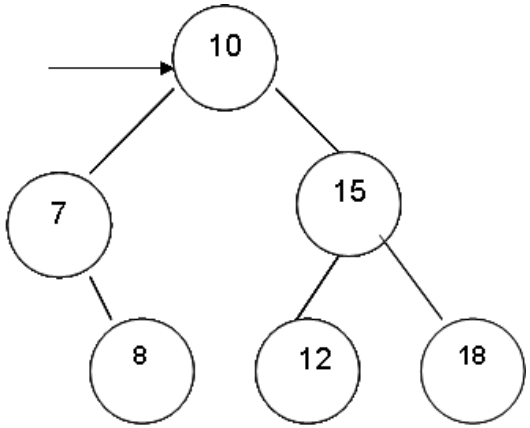
- Next, 7 is smaller than the root. Therefore, it must be inserted as the left child of the root as shown in the following figure.

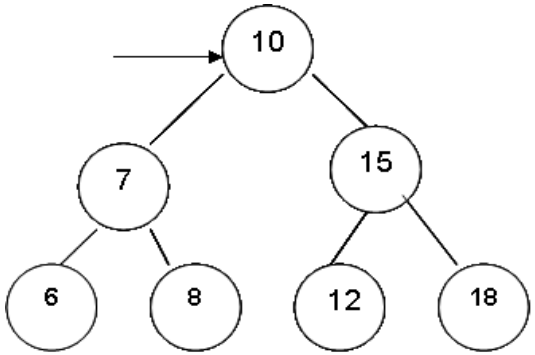


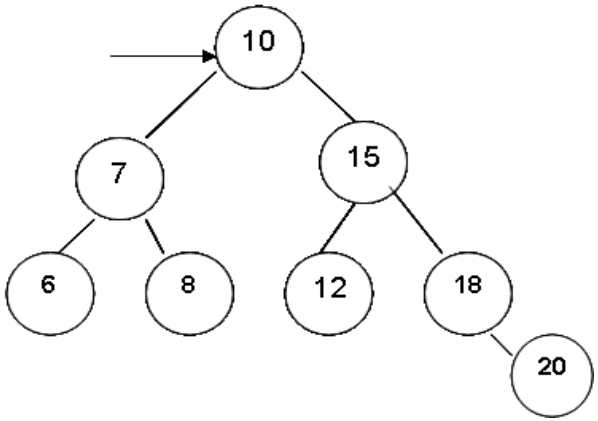
Insertion into a Tree (Contd.).

- Similarly, 8, 18, 6 and 20 are inserted at the proper place as shown in the following figures.









Insertion into a Tree (Contd.).

- This example clearly illustrates that given the root of a binary search tree and a value to be added to the tree, we must search for the proper place where the new value can be inserted.

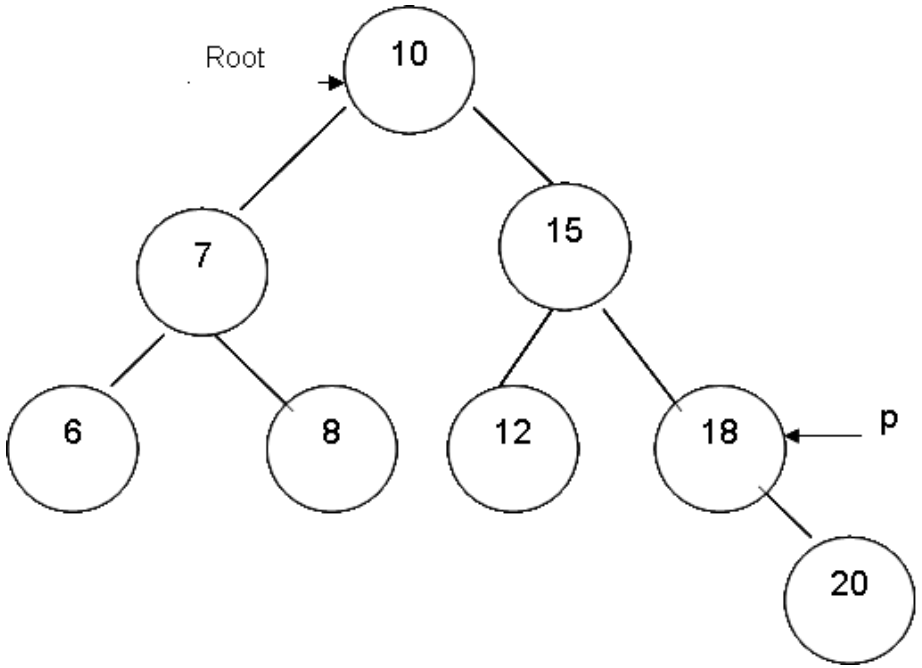
- We must also create a node for the new value and finally, we have to adjust the left and right pointers to insert the new node.

- To find the insertion place for the new value, say 17, we initialize a temporary pointer p, which points to the root node.

Insertion into a Tree (Contd.).

- We can change the contents of p to either move left or right through the tree depending on the value to be inserted.

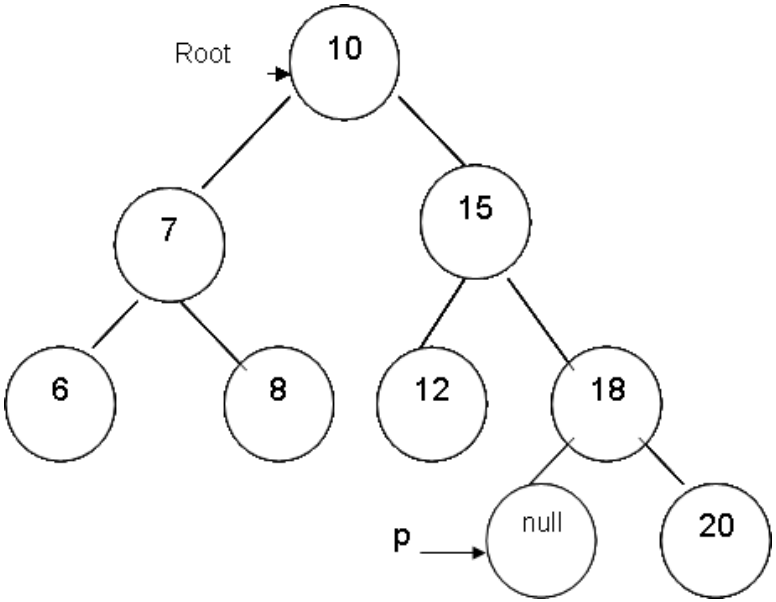
- When p becomes null, we know that we have found the insertion place as in the following figure.



Insertion into a Tree (Contd.).

- But once p becomes null, it is not possible to link the new node at this position because there is no access to the node that p was pointing to (node with value 18) just before it became null.

- From the following figure, p becomes null when we have found that 17 will be inserted at the left of 18.



Insertion into a Tree (Contd.).

- You therefore need a way to climb back into the tree so that you can access the node containing 18, in order to make its left pointer point to the new node with the value 17.
- For this, you need a pointer that points to the node containing 18 when p becomes null.
- To achieve this, you need to have another pointer (trail) that must follow p as p moves through the tree.
- When p becomes null, this pointer will point to the leaf node (the node with value 18) to which you must link the new node (node with value 17).

Insertion into a Tree (Contd.).

- Once you know the insertion place, you must adjust the pointers of the new node.
- At this point, you only have a pointer to the leaf node to which the new node is to be linked.
- You must determine whether the insertion is to be done at the left subtree or the right subtree of the leaf node.
- To do that, you must compare the value to be inserted with the value in the leaf node.
- If the value in the leaf node is greater, we insert the new node as its left child; otherwise we insert the new node as its right child.

Creating a Tree – A Special Case of Insertion

- A special case of insertion that you need to watch out for arises when the tree in which you are inserting a node is an empty tree.
- You must treat it as a special case because when p equals null, the second pointer (trail) trailing p will also be null, and any reference to info of trail like trail->info will be illegal.
- You can check for an empty tree by determining if trail is equal to null. If that is so, we can initialize root to point to the new node.

Code Implementation For Insertion into a Tree

- The C function for insertion into a binary tree takes two parameters; one is the pointer to the root node (root), and the other is the value to be inserted (x).

- You will implement this algorithm by allocating the nodes dynamically and by linking them using pointer variables. The following is the code implementation of the insert algorithm.

Code Implementation For Insertion into a Tree (Contd.).

```
tree insert(s,x)
int x;
tree *s;
{ tree *trail, *p, *q;
 q = (struct tree *) malloc (sizeof(tree));
 q->info = x;
 q->left = null;
 q->right = null;
 p = s;
 trail = null;
 while (p != null)
 {  trail = p;
   if (x < p->info) p = p->left;
   else          p = p->right;
 }
```

```
/*insertion into an empty tree; a special
   case of insertion */
if (trail == null)
 { s = q;
   return (s);
 }
if(x < trail->info)
   trail->left = q;
else
  trail->right = q;
return (s);
}
```

Code Implementation For Insertion into a Tree using Recursion

- You have seen that to insert a node, you must compare x with root->info.
- If x is less than root->info, then x must be inserted into the left subtree.
- Otherwise, x must be inserted into the right subtree.
- This description suggests a recursive method where you compare the new value (x) with the one in the root and you use exactly the same insertion method either on the left subtree or on the right subtree.

Code Implementation For Insertion into a Tree using Recursion (Contd.).

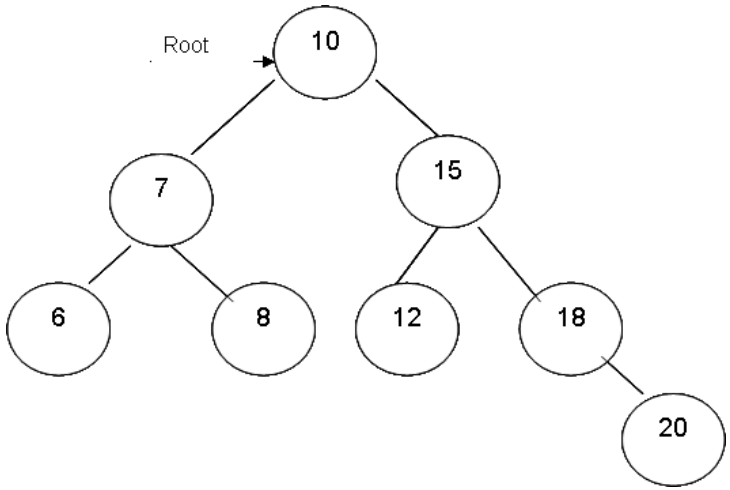
- The base case is inserting a node into an empty tree. You can write a recursive routine (rinsert) to insert a node recursively as follows:

```
tree rinsert (s,x)
tree *s;
int x;
 { /* insertion into an empty tree; a special case of insertion */
  if (!s)
   {    s=(struct tree*) malloc (sizeof(struct tree));
        s->info = x;
        s->left = null;
        s->right = null;
        return (s);
   }
   if (x < s->info)    s->left = rinsert(x, s->left);
 else if (x > s->info) s->right = rinsert(x, s->right);
return (s);
}
```

Circumstances When a Binary Tree Degenerates into a Linked List

- The shape of a binary tree is determined by the order in which the nodes are inserted.

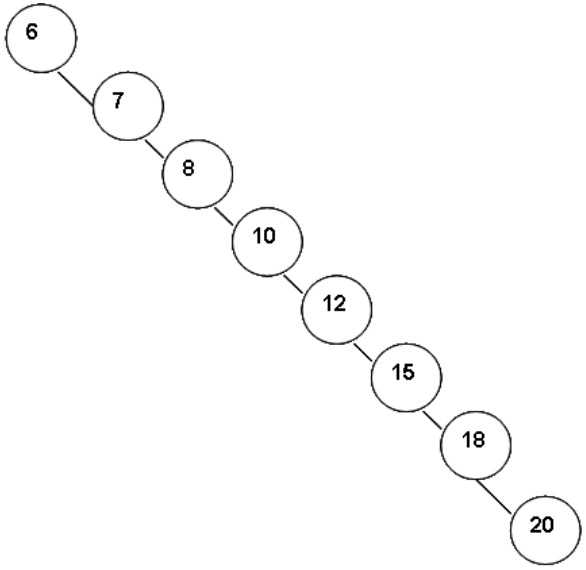
- Given the following input, their insertion into the tree in the same order would more or less produce a balanced binary search tree as shown below:
  - Input values: 10, 15, 12, 7, 8, 18, 6, 20



Circumstances When a Binary Tree Degenerates into a Linked List (Contd.).

- If the same input is given in the sorted order as
- 6, 7, 8, 10, 12, 15, 18, 20, you will construct a lopsided tree with only right subtrees starting from the root.

- Such a tree will be conspicuous by the absence of its left subtree from the top.

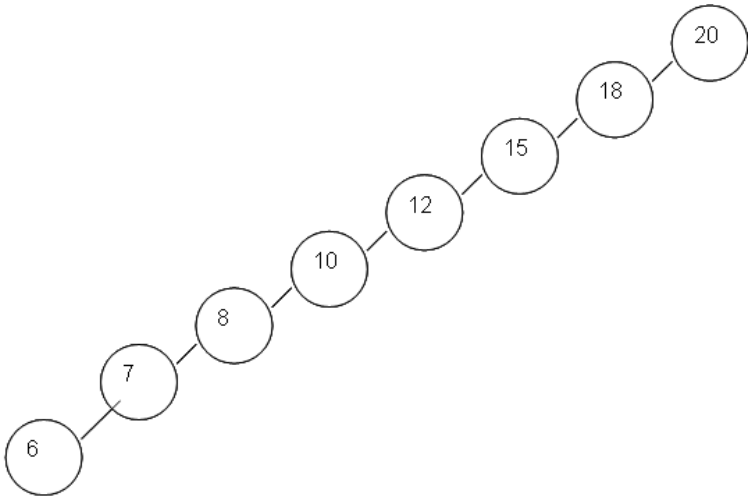


A Lopsided Binary Tree With Only Right Subtrees

Circumstances When a Binary Tree Degenerates into a Linked List (Contd.).

- However if you reverse the input as
- 20, 18, 15, 12, 10, 8, 7, 6, and insert them into a tree in the same sequence, you will construct a lopsided tree with only the left subtrees starting from the root.

- Such a tree will be conspicuous by the absence of its right subtree from the top.



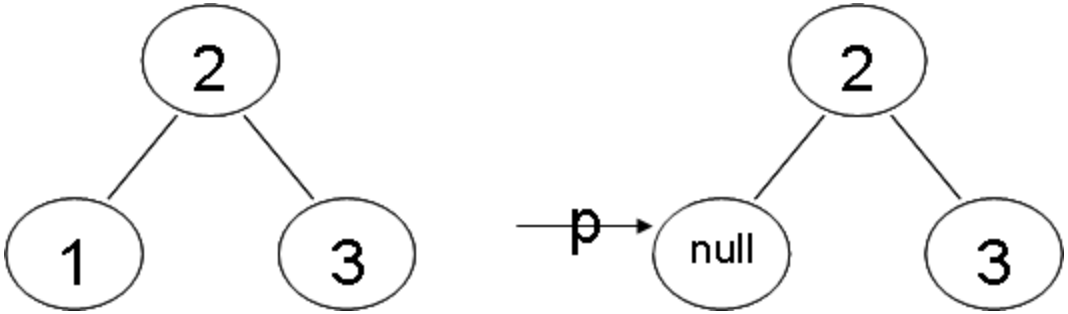
A Lopsided Binary Tree With Only Left Subtrees

Deletion from a Binary Search Tree

- An important function for maintaining a binary search tree is to delete a specific node from the tree.
- The method to delete a node depends on the specific position of the node in the tree.
- The algorithm to delete a node can be subdivided into different cases.

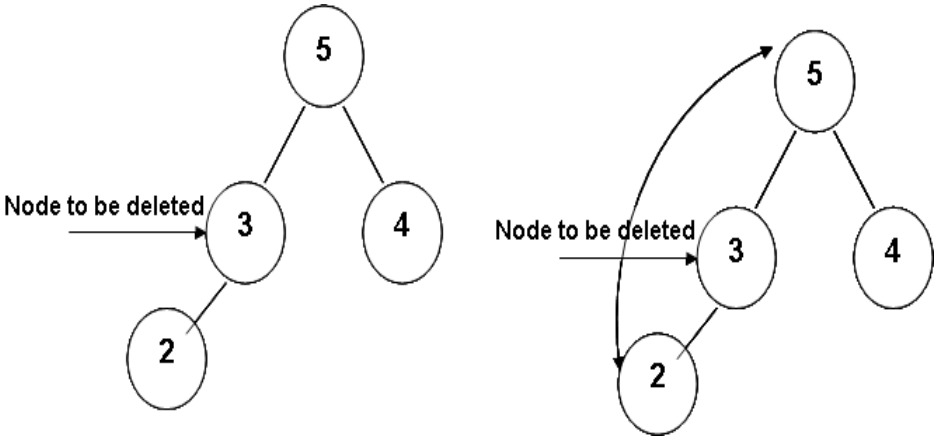
Case I – Deletion Of The Leaf Node

- If the node to be deleted is a leaf, you only need to set appropriate link of its parent to null, and do away with the node that is to be deleted.
- For example, to delete a node containing 1 in the following figure, we have to set the left pointer of its parent (pointing to 1) to null.
- The following diagram illustrates this.



Case II – Deletion Of a Node With a Single Child

- If the node to be deleted has only one child, you cannot simply make the link of the parent to nil as you did in the case of a leaf node.
- Because if you do so, you will lose all of the descendants of the node that you are deleting from the tree.
- So, you need to adjust the link from the parent of deleted node to point to the child of the node you intend to delete. You can subsequently dispose of the deleted node.



To delete node containing the value 3, where the right subtree of 3 is empty, we simply make the link of the parent of the node with the value 3 (node with value 5) point to the child of 3 (node with the value 2).

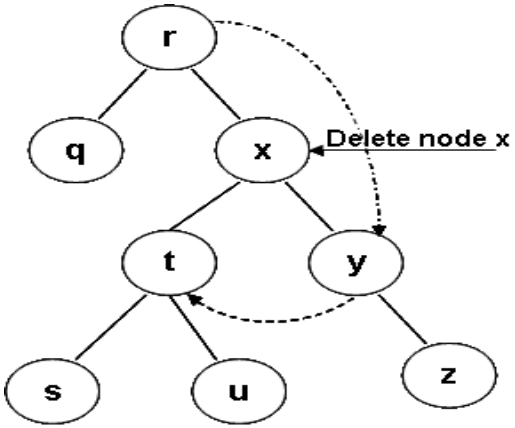
Case III – Deletion Of a Node With Two Child Nodes

- Complications arise when you have to delete a node with two children.
- There is no way you can make the parent of the deleted node to point to both of the children of the deleted node.
- So, you attach one of the subtrees of the node to be deleted to the parent, and then link the other subtree onto the appropriate node of the first subtree.
- You can attach the right subtree to the parent node and then link the left subtree on to the appropriate node of the right subtree.

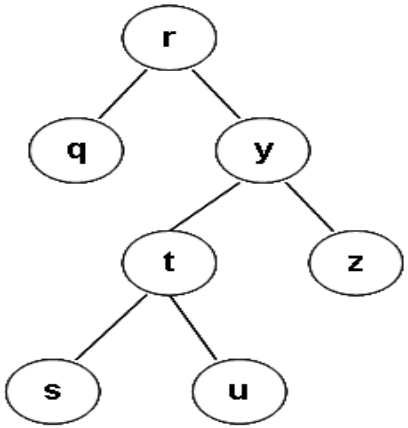
Case III – Deletion Of a Node With Two Child Nodes (Contd.).

- Therefore, you must attach the left subtree as far to the left as possible. This proper place can be found by going left until an empty left subtree is found.

- For example, if you delete the node containing x as shown in the following figure, you make the parent of x (node with the value r) point to the right subtree of x (node containing y) and then go as far left as possible (to the left of the node containing y) and attach the left subtree there.



Before Deletion of Node x



After Deletion of Node x

Code Implementation for Node Deletion for Cases I, II & III

```
void delete (p)
Struct tree *p
 {
  struct tree *temp
  if (p == null)
  printf("Trying to delete a non-existent node");
  else if (p->left == null)
   {
    temp = p;
    p = p->right;
    free(temp);
   }
  else if (p->right == null)
   {
    temp = p;
    p = p->left;
    free (temp);
   }
```

Code Implementation for Node Deletion for Cases I, II & III (Contd.).

```
else if(p->left != null && p->right!= null)
    {
     temp = p->right;
     while (temp->left != null)
      {
       temp = temp->left;
      }
     temp->left = p->left;
     temp = p;
     p = p->right;
     free (Temp);
    }
 }
```

Code Implementation for Node Deletion for Cases I, II & III (Contd.).

- Note that the while loop stops when it finds a node with an empty left subtree so that the left subtree of the node to be deleted can be attached here.

- Also, note that you first attach the left subtree at the proper place and then attach the right subtree to the parent node of the node to be deleted.

Search The Tree (Contd.).

- To search a tree, you employ a traversal pointer p, and set it equal to the root of the tree.
- Then you compare the information field of p with the given value x. If the information is equal to x, you exit the routine and return the current value of p.
- If x is less than p->info, you search in the left subtree of p.
- Otherwise, you search in the right subtree of p by making p equal to p->right.

Search the Tree (Contd.).

- You continue searching until you have found the desired value or reach the end of the tree. You can write the code implementation for a tree search as follows:

```
search (p,x)
int x;
struct tree *p;
 {
  p = root;
  while (p != null && p->info != x)
   {
    if (p->info > x)
     p = p->left;
    else
     p = p->right;
   }   return (p);
  }
```

Hands-on: 2 hours

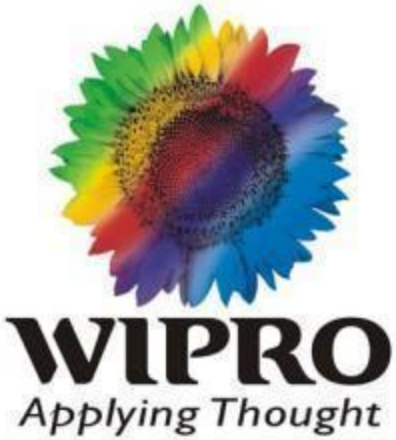
Purpose

- construct binary tree and perform search operations

Summary

In this module, we discussed:

- Binary tree and terminologies associated with a binary tree
- Dynamic allocated data structure to represent a binary tree
- Traversing a binary tree
- Adding nodes to a binary tree
- Removing nodes from a binary tree
- Searching a binary tree



Thank You