COMPUTER SCIENCE & INFORMATION TECHNOLOGY

Programing and Data Structures

Comprehensive Theory
with Solved Examples *and* Practice Questions

MADE EASY PUBLICATIONS

www.madeeasypublications.org

MADE EASY Publications Pvt. Ltd.

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016 | **Ph. :** 9021300500

Email : infomep@madeeasy.in | **Web :** www.madeeasypublications.org

Programming and Data Structures

© Copyright by MADE EASY Publications Pvt. Ltd.
All rights are reserved. No part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photo-copying, recording or otherwise), without the prior written permission of the above mentioned publisher of this book.

MADE EASY Publications Pvt. Ltd. has taken due care in collecting the data and providing the solutions, before publishing this book. Inspite of this, if any inaccuracy or printing error occurs then **MADE EASY Publications Pvt. Ltd.** owes no responsibility. We will be grateful if you could point out any such error. Your suggestions will be appreciated.

EDITIONS

First Edition : 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020
Seventh Edition : 2021
Eighth Edition : 2022
Ninth Edition : 2023

Tenth Edition : 2024

CONTENTS

Programming and Data Structures

CHAPTER 1

Programming Methodology 2-78

1.1 Data Segments in Memory 2
1.2 Scope of Variable 4
1.3 C Variable 6
1.4 Operators in C 9
1.5 Address arithmetic in C 16
1.6 Value of Variable in C Language 16
1.7 Flow Control in C 17
1.8 Function 25
1.9 Recursion 31
1.10 C Scope Rules 34
1.11 Storage Class 37
1.12 Pointers 45
1.13 Sequence Points in C 58
1.14 Declarations and Notations 60
1.15 Const Qualifier 61
1.16 Strings in C 62
Student Assignments 64

CHAPTER 2

Arrays 79-97

2.1 Definition of Array 79
2.2 Declaration of Array 79
2.3 Properties of Array 80
2.4 Accessing Elements of an Array 83
Student Assignments 92

CHAPTER 3

Stack 98-126

3.1 Introduction 98
3.2 Operation on Stack 98
3.3 Simple Representation of a Stack 100
3.4 ADT of Stack 100
3.5 Operations of Stack 100
3.6 Average Stack Lifetime of an Element 105
3.7 Applications of Stack 106
3.8 Tower of Hanoi 116
Student Assignments 119

CHAPTER 4

Queue 127-146

4.1 Introduction 127
4.2 Operations of Queue 127
4.3 Application of Queue 129
4.4 Circular Queue 129
4.5 Implement Queue using Stacks 130
4.6 Implement Stack Using Queues 131
4.7 Average Lifetime of an Element in Queue 134
4.8 Types of Queue 134
4.9 Double Ended Queue (Dequeue) 135
4.10 Priority Queue 136
Student Assignments 139

CHAPTER 5

Linked Lists 147-180

5.1 Introduction 147
5.2 Linked Lists 148
5.3 Uses of Linked lists 148
5.4 Singly Linked List or One Way Chain 148
5.5 Circular Single Linked List 157

5.6 Doubly Linked Lists or Two-way chain 159
5.7 List Implementation of Queues 164
5.8 List Implementation of Stacks 165
5.9 List Implementation of Priority Queues 166
5.10 Other operation on Linked List 166
5.11 Polynomial Addition Using Linked List 167
5.12 Polynomial Multiplication Using Linked List 168
Student Assignments 170

CHAPTER 6

Trees 181-235

6.1 Introduction 181
6.2 Glossary 181
6.3 Applications of Tree 182
6.4 Tree Traversals for Forests 183
6.5 Binary Trees 184
6.6 Types of Binary Trees 184
6.7 Applications of Binary Tree 186
6.8 Internal and External Nodes 190
6.9 Expression Trees 192
6.10 Binary Tree Representations 193
6.11 Implicit Array Representation of Binary Trees 194
6.12 Threaded Binary Trees 196
6.13 Representing Lists as Binary Trees 196
6.14 Binary Search Tree 199
6.15 AVL Tree (Adelson Velski Landis) 208
Student Assignments 223

CHAPTER 7

Hashing Techniques 236-252

7.1 Introduction 236
7.2 Hash Function 236
7.3 Collisions 237
7.4 Collision Resolution Techniques 237
7.5 Hashing Function 244
7.6 Comparison of Collision Resolution Techniques 246
7.7 Various Hash Function 246
Student Assignments 248

Programming and Data Structures

Goal of the Subject

Computer Science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

A data structure is a specialized format for organizing and storing data. To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

Introduction

In this book we tried to keep the syllabus of Software Programming and Data structures around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into seven chapters as described below.

1. Programming Methodology: In this chapter we will study about the different segments and their organization, variables and their scope, flow of control in a program, function evaluation types, storage classes, and pointers and finally we discuss the application of pointers.
2. Arrays: In this chapter we will study properties and application of arrays, accessing methods for two and three dimensional arrays and finally we discuss the arrays in the form of special matrices.
3. Stack: In this chapter we will study the ADT of stack, operations on stack, applications and different types of notations evaluated by stack and finally we discuss the tower of Hanoi (application).
4. Queue: In this chapter we will study about the Queue, operations on queue, applications and finally we discuss different types of queues.
5. Linked List: In this chapter we will study types and applications of linked list, operations on linked list, priority queue and finally we discuss implementation of stack, queue and priority queue using lists.
6. Trees: In this chapter we introduce trees, their applications, types of trees (BST, B-tree, and AVL), different types tree traversals and finally we discuss operations on trees.
7. Hashing Techniques: In this chapter we introduce the Hash function, collision resolution techniques and comparisons of different collision techniques.

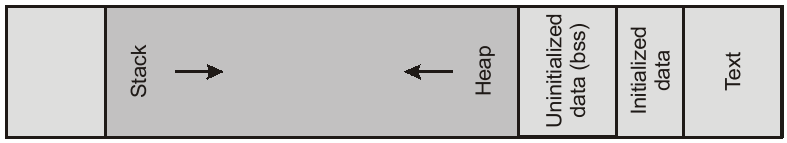
■■■■

CHAPTER

1 Programming Methodology

1.1 DATA SEGMENTS IN MEMORY

The running program stored the machine instructions (program code) and data in the same memory space. The memory is logically divided into text and data segments. A single text segment is used by modern systems to store program instructions, but to store data need more than one segment, depending upon the storage class of the data being stored there. The below figure shown the different segments in memory:



1. **Text (Code) Segment:** Text segment contains machine code of the compiled program. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified.
2. **Initialized Data Segment:** Initialized data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. Data segment is not read only, since the values of the variables can be altered at run time. This segment can be further classified into initialized read-only area and initialized read-write area.

Example 1.1 Consider the following C code:

Code

```
#include<stdio.h>
char c[] = "Madeeasy";          /* global variable stored in initialized data
                                   segment in read-write area */
const char p[]="Madeeasy";      /* global variable stored in initialized data
                                   segment in read-only area */
int main() {
    static int a=11;            /* static variable stored in initialized data
                                   segment */
    return 0;
}
```

3. **Uninitialized Data Segment:** Uninitialized data segment, also called bss segment. Data in this segment is initialized by the Kernel to 0 before the program starts executing.

Uninitialized data segment starts at the end of initialized data segment and contains all global and static variables that do not have explicit initialization in source code.

Example 1.2 What will be output of following C code?

Code

```
#include<stdio.h>
char c;                          /* Uninitialized variable stored in basic bss */
int main()
{
    static int i;                /* Uninitialized static variable stored in bss */
    return 0;
}
```

4. **Heap:** Heap is the segment where dynamic memory allocation takes place. The heap area begins at the end of uninitialized data segment and grows upwards. It is managed by malloc, calloc, ralloc and free. The heap area is shared by all shared libraries and dynamically loaded modules in a process.

Example 1.3 What will be output of following C code?

Code

```
#include<stdio.h>
int main()
{
    char *p = (char*)malloc (sizeof (char));   /*  Memory  allocating  in  heap
                                                   segment */
    return 0;
}
```

5. **Stack:** It is adjoined with heap area and grows is opposite direction. When the stack pointer meets the heap pointer, it means free memory has exhausted.
   Stack segment is used to store local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function is over.
   Local variables have a scope within the defined block.
   Recursive functions use stack. Each time a recursive function calls itself, a new stack frame is used, so that one set of variables don't interfere with variables from another instance of the function.

Example 1.4 What will be output of following C code?

Code	Solution

`#include<stdio.h>`
`int main(){`
`  int i;`
`  for(i=0;i<3;i++){`
`  static int a=5;`
`  printf("%d",a++);`
`  }`
`  return 0;`
`}`
	Output: 5 6 7
	Static variable scope is limited to the block but its life time is upto program termination.
	In every iteration of the loop, a retains its previous value.
	First iteration a=5
	Second iteration a=6
	Third iteration a=7

1.2 SCOPE OF VARIABLE

In a program, a scope is a region where a defined variable can have its existence and beyond that variable cannot be accessed.

There are two type of scoping techniques: (1) Static scoping and (2) Dynamic scoping.

Static scoping: Static scoping is defined in terms of the physical structure of the program. The determination of scope is done at a time of compilation i.e. binding are resolved at the time of execution of program. Ada and C++, Pascal, C are the example of static scoping.

Syntax of static scoping

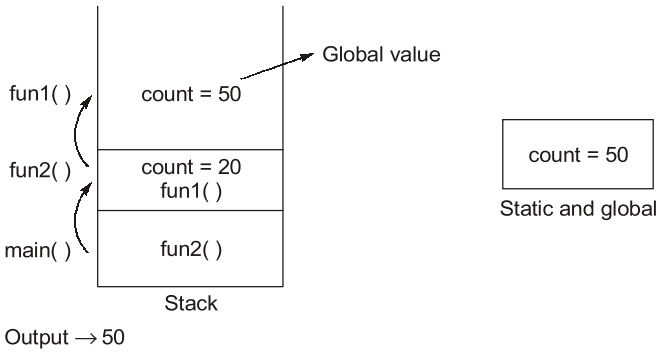
```
for(...){
    int i;
    {...
        {
            j = 5;
        }
    }
    ...
}
```

In static scoping, if a variable is not found in the scope, compiler looks for the variable in the global section and then in successively smaller scopes.

Example 1.5 What will be output of following program?

```
int count=50;
void fun1(){
    printf("In fun1=%d", count);
}
void fun2(){
    int count=20;
    fun1();
}
int main(void){
    fun2();
    return0;
}
```

Solution:



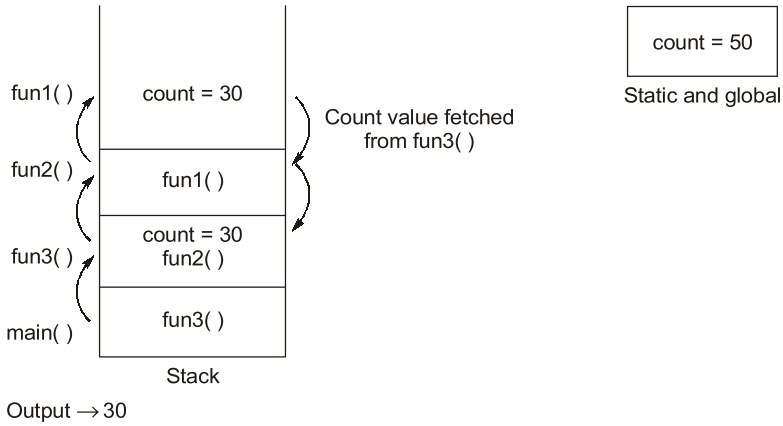
Dynamic Scoping: Dynamic scoping does not care how the code is written, but instead how it executes.

In dynamic scoping, binding depends on the flow of control at run time and the order in which functions are called, refers to the closest active binding. List is the example of dynamic scoping.

Example 1.6 What will be output of following program?

```
int count=50;
void fun1()
{
    Printf("In fun1=%d", count);
}
void fun2()
{
    fun1();
}
void fun3()
{
    int count=30;
    fun2();
}
int main(void)
{
    fun3();
    return0;
}
```

Solution :



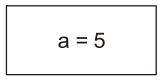
Example 1.7 What will be the output of following program using static and dynamic scoping?

Code

```
#include <stdio.h>
int a=5;
int main()
{
    int a=10;
    B();
}
```

```
B();
{
    int a=20;
    C();
}
C();
{
    int a=30;
    D();
}
D();
{
    printf("%d", a);
}
```

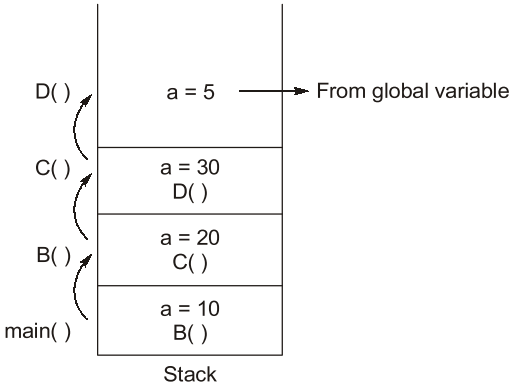
Solution :



Static and global

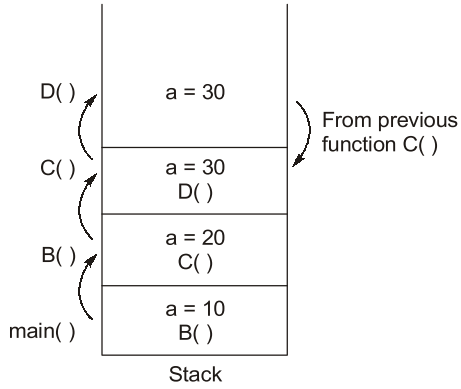
Static is global

Using static scoping:



Output → 5

Using dynamic scoping:



Output → 30

1.3 C VARIABLE

A variable is named location of data. In other word we can say variable is container of data. In real world you have used various type of containers for specific purpose. For example you have used suitcase to store clothes, match box to store match sticks etc. In the same way variables of different data type is used to store different types of data. For example integer variables are used to store integers char variables is used to store characters etc. On the basis of type of data a variable will store, we can categorize the all C variable in three groups.

(a) Variables which can store only one data at time.
Example: Integer variables, char variables, pointer variables etc.

(b) Variables which can store more than one data of similar type at a time.
Example: Array variables

(c) Variables, which can store more than one value of dissimilar type at a time.
Example: Structure or union variables.

1.3.1 Properties of C Variable

Every variable in C have three most fundamental attributes. They are Name, Value and Address.

Name of a Variable: Every variable in C has its own name. A variable without any name is not possible in C. Most important properties of variables name are its unique names.

- No two variables in C can have same name with same visibility.
- It is possible that two variable with same name but different visibility. In this case variable name can access only that variable which is more local. In C there is not any way to access global variable if any local variable is present of same name.

Example 1.8 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int main(){`
`  auto int a=5;  //Visibility is within main block`
`  static int a=10;  //Visibility is within main block`
`  /* Two variables of same name */`
`  printf("%d",a);`
`  return 0;`
`}`
	Output: compilation error
	Here variable 'a' is declared twice in main block. So it generate compile time error.

Example 1.9 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int a=50;  //Visibility is in whole program`
`int main()`
`{  int a=10;  //Visibility within main block`
`  printf("%d",a);`
`  return 0;`
`}`
	Output: 10
	Here variable 'a' has different visibility at two different places. So printf will print main block variable value.

NOTE: In C any name is called identifier. This name can be variable name, function name, enum constant name, micro constant name, goto label name, any other data type name like structure, union, enum names or typedef name.

1.3.2 Identifier Naming Rule in C

In C any name is called identifier. This name can be variable name, function name, enum constant name, micro constant name, goto label name, any other data type name like structure, union, enum names or typedef name.

Rule 1: Name of identifier includes alphabets, digit and underscore.

Valid name: world, addition23, sum_of_number etc.

Invalid name: factorial#, avg value, display*number etc.

Rule 2: First character of any identifier must be either alphabets or underscore.

Valid name: _calculate, _5,a_, __ etc.

Invalid name: 5_, 10_function, 123 etc.

Student's Assignment

Q.1 What is the meaning of the following declaration?
int * *p*(char *a[ ]);
(a) P is a pointer to a function that accepts an argument which is a pointer to a character array and returns an integer quantity.
(b) P is a function that accepts an argument which is an array of pointers to characters and returns a pointer to an integer.
(c) P is a function that accepts an argument which is an array of pointers to characters and returns an integer quantity.
(d) P is a pointer to a function that accepts an argument which is an array of pointers to characters and returns a pointer to an integer quantity.

Q.2 Consider the following program:

```
#define funct(x) x*x+x
int main()
{
  int x;
  x=36+funct(5)*funct(3);
  printf("%d",x);
  return 0;
}
```

What will be the output of the above program?
(a) 73 (b) 396
(c) 109 (d) 360

Q.3 Consider the following function given below:

```
int function(int n){
  if(n-1)
  return 2*function(n-1)+n;
  else
  return 0;
}
```

What is the value returned by function (5)?
(a) 33 (b) 41
(c) 57 (d) 65

Q.4 What is the output of the following program if dynamic scoping is used?

```
int a, b, c;
void func1(){
  int a,b;
  a=6;
  b=8;
  func2();
  a=a+b+c;
  print(a);
}
  void func2(){
  int b,c;
  b=4;
  c=a+b;
  a+= 11;
  print(c);
}
  void main(){
  a=3;
  b=5;
  c=7;
  func1();
}
```

Output of program:
(a) 7 19 (b) 10 1
(c) 10 23 (d) 10 32

Q.5 Assuming only numbers and letters given in the input, what does the following program do?

```
#include<stdio.h>
int main()
{
   int i,j, ascii[128];
   char ip[30];
   printf("Enter Input string: ");
   scanf("%s",ip);
   for(i=0;i<128;i++)
{
   ascii[i]=0;
}
   i=0;
   while(ip[i]!='\0')
{
   j=(int)ip[i];
   ascii[j]++;
   if(ascii[j]>1)
{
   printf("%c",ip[i]);
   return 0;
```

```
}
    i++;
}
   return 0;
}
```

(a) Prints the position of first repeated character in the string
(b) Prints the first repeated character in the string
(c) Prints the position of last repeated character in the string
(d) Prints the last repeated character in the string

Q.6 Consider the following code segment:

```
void foo(int x, int y){
  X+=y;
  Y+=x;
}
  main(){
  int x=4.5;
  foo(x,x);
}
```

What is the final value of x in both call by value and call by reference respectively?

(a) 4 and 12 (b) 5 and 12
(c) 12 and 16 (d) 4 and 16

Q.7 What will be the output of following C-code?

```
int main(){
  int a[3]={67,43,23};
  int *p=a;
  printf("%d",++*p);
  printf("%d",*++p);
  printf("%d",*p++);
  return 0;
}
```

(a) 68 43 43 (b) 37 43 43
(c) 67 43 23 (d) 68 43 23

Q.8 Consider following C-code:

What is the value returned by fun (10)?

```
int fun (int n){
 Static int i=10;
 if (n>=200)
 return (n+i);
 else{
    n=n+i;
    i=n+i;
    return fun(n);
  }
}
```

(a) 890 (b) 210
(c) 340 (d) None of these

Q.9 What will be the output of following C-code?

```
void fun(void *p);
static int i;
int main()
{
   void *ptr;
   ptr=&i;
   fun (ptr);
   return 0;
}
   void fun( void *p)
{
   int **q;
   q=(int**)&p;
   printf("%d",**q);
```

(a) Garbage value (b) 0
(c) 1 (d) compile-time error

Q.10 Output of the following program will be

```
int main()
{
 char *p1="Graduate";
 char *p2=p1;
 printf ("%c,%d",*++p2,sizeof(p2));
}
```

(a) G, 8 (b) r, 7
(c) r, 4 (d) r, 8

Q.11 What is the output of following program?

```
int f(int a)
{  printf("%d", a++);
   return(++a);
}
   main()
{
   int b=1;
   b=f(b);
   b=f(b);
   b=f(1+f(b));
}
```

(a) 1 3 3 5 (b) 1 3 5 8
(c) 2 3 3 5 (d) 2 4 6 8

Q.12 Consider the following C code:

```
int*P,A[3]={0,1,2};
P=A;
*(P+2)=5;
P=A++;
*P=7;
```

What are the values stored in the array A from index 0 to index 2 after execution of the above code?

(a) 7, 5, 2 (b) 7, 1, 5
(c) 0, 7, 5 (d) None of these

Q.13 Consider the following code:

```
int f(int a, int b)
{
  if(b==0) return 1;
  else if(b%2==0)
  {
    return(f(a,b/2)*f(a,b/2));
  }
  else
  {
    return(a*f(a,b/2)*f(a,b/2));
  }
}
```

What is the return value of f(2, 10)?

Q.14 What is the output of the following C-code? (Assume that the address of X is 2000 (in decimal) and on integer requires 4 B of memory).

```
int main( )
{
    unsigned int X[4] [3] = {{1, 2, 3},
                            {4, 5, 6},
                            {7, 8, 9},
                            {10, 11, 12}};
    printf("%u %u %u", X+3,*(X+3),*(X+2) + 3);
}
```

(a) 2036, 2036, 2036 (b) 2012, 4, 2024
(c) 2036, 10, 10 (d) 2012, 4, 6

Q.15 Which of the following statements is correct?

(a) Void pointers can be used for dereferencing.
(b) Arithmetic operations can be applied on void pointers.
(c) The default value of extern storage class is garbage value.
(d) A particular extern variable can be declared many times, but canbe initialized at only 1 time.

Q.16 Consider the following function:

```
Void Test (int arr[ ], int s, int e)
{
    int temp;
    if(s ≥ e)
        return;
    temp = arr[s + 1];
    arr[s + 1] = arr[e];
    arr[e] = temp;
    Test(arr, s + 1, e – 1)
}
```

Let arr is an array [0 to 4] which initially holds elements {1, 2, 3, 4, 5}. Test (arr[ ], 0, 3) is called. The number of elements that will maintain their initial position after the end of the code are _____.

Q.17 Consider the following C program segment:

```
# include <stdio.h>
main( )
{
    static char *s[ ] = {"madeeasy", "online",
    "test", "series"};
    char **ptr[ ] = {s + 3, s + 2, s + 1, s},
    *** p;
    p = ptr;
    ++p;
    printf("%s", * – – *++p + 3)
}
```

What will be printed by the program?

(a) line (b) ies
(c) test (d) eeasy

Q.18 What is the output of the following program?

```
#include <stdio.h>
#define MUL (a, b) a * b
#define pow (a) a * a
int main ( )
{
    int a = 3;
    int b = 2;
```

```
    printf ("%", MUL (MUL (a + 1, b),
    pow (b + 1)));
    return 0;
}
```

Q.19 Consider the following C-program:

```
#include <stdio.h>
int value (int *x) {
    static int count;
while (*x) {
    count = count + (*x & 1);
    *x >> = 1;
}
return count;
}
int main ( ) {
}
    int a[ ] = {3, 5, 6, 4};
    int y = 0, z = 0;
    for (; y < size of (a)/size of (int); y++)
        z = a[y] + value (&a[y]);
}
```

The value of z at the end of program is _____.

Q.20 Consider the following recursive program:

```
#include <stdio.h>
int main ( ) {
    code (4);
return 0;
}
    int code (int m) {
        if (m > 0) {
        int i = 1;
        for (; i < 3; i ++) {
                code (m – i);
                code (m – i – 1);
        printf("MadeEasy");
        }
    }
    return 0;
}
```

The number of times "MadeEasy" will be printed _____.

Q.21 Consider the following code:

```
int Do (char * gate)
{
    char * gate1 = gate;
    char * gate2 = gate + strlen (gate) – 1;
    while (gate1 < gate2)
    {
        if (* gate1 ++ ! =* gate2 - -)
        return 0;
    }
    return 1;
}
```

What is the functionality of above function Do ( )?

(a) Check whether string is odd palindrome
(b) Check whether the string is even palindrome
(c) Check whether the string is palindrome
(d) None of the above

Q.22 Consider the following program segment prints:

```
# include <stdio.h>
void fun (int *a, int *b)
{   a = b;
    *a+ = 2;
    a = b;
}
int p = 50, q = 20;
int main( )
{   fun (&p, &q);
    fun(&p, &p);
    fun(&q, &q);
    printf ("%d %d", p, q);
    return 0;
}
```

Which of the following represents the values printed by above program segments?

(a) 50, 22 (b) 52, 24
(c) 54, 20 (d) None of these

Q.23 Consider the following C program:

```
main ( )
{
    int p = –3, q = 2, r = 0, s, t;
    s = ++ p && ++ q || r ++;
    t = p + q + s ++ ;
    printf("\n % d% d", s, t);
}
```

Which of the following represents output of above program?

(a) 2 2 (b) 1 2
(c) 0 3 (d) 2 3

Q.24 Consider the following code:

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char *a = "MADEEASY";
    char *b = "GATECSIT2019";
    char *r = a;
    char *s = b;
    printf("%d", (int) strlen(b + 3[r] – 1[s]));
    return 0;
}
```

The output of the following program will be _____.

Q.25 Consider the given function magic ( ):

```
int magic (int n)
{
    static int r = 5;
    if (n <= 0) return 10;
    if (n > 3)
{
    r = 50;
    return (r + magic (n – 1));
}
    return (r – magic (n – 1));
}
```

The output corresponding to the function call magic (8) is _____.

Q.26 Consider a hypothetical machine which supports the following data types:
unsigned char: 1 Byte
unsigned short: 2 Bytes
int: 4 Bytes
Consider the following function red( )

```
int red(unsigned char a, unsigned short b)
{
    if (a = = 0) return b;
    else {
        a = a + 1;
        b = b * 2;
        return red(a, b); }
}
int main( ) {
    printf("%d", red((char) 240, 1));
    return 0;
}
```

What will be the output of the following program?

(a) The program terminates abnormally
(b) The program goes into infinite loop
(c) The program outputs 65536 (2^{16})
(d) None of these

Q.27 What will be the output of the following C program?

```
# include <stdio.h>
void print 1 (void)
{
    static int x = 10;
    x+ = 5;
    printf ("%d", x);
}
Void print 2 (void)
{
    static int x;
    x = 10;
    x+ = 5;
    printf("%d", x);
}
int main ( )
{
    print 1( ); print 1( ); print 2( ); print 2 ( );
    return 0;
}
```

(a) 15, 20, 25, 30 (b) 15, 20, 15, 20
(c) 15, 15, 15, 15 (d) None of these

Q.28 Consider the following program:

```
main( )
{
    int i, j;
    int A[m][n] = {{1, 2, 3} {4, 5, 6}}
    for(i = 0; i < n; i ++)
    for(j = 0; j < m; j ++)
    printf("%d", * (A[j] + i));
}
```

For the output printed by the above program.

(a) 1 2 3 4 5 6 (b) 1 4 2 5 3 6

(c) 4 5 6 7 8 9 (d) 4 5 6 4 5 6

Q.29 Consider the following C program:

```
int x = 10;
Void Part1(int *a) {
    *a + = x ++;
    printf("%d" *a);
}
Void Part2(int *b) {
    static x = 15;
    *b = *b × x;
    Part1(&x);
    printf("%d" *x);
}
Void main ( ) {
    Part2(&x);
    Part1(&x);
}
```

What will be the output using static scoping and dynamic scoping respectively?

(a) Static: 165, 303, 303

(b) Static: 165, 165, 165
Dynamic: 31, 301, 301
Dynamic: 31, 31, 31

(c) Static: 303, 303, 303

(d) Static: 165, 165, 303
Dynamic: 301, 301, 301
Dynamic: 31, 31, 301

Q.30 Consider the following C program?

```
# include <stdio.h>
int main( )
{
    short int c[5];
    int i = 0;
    for (i = 0; i < 5; i++)
    c[i] = 400+i ;
    char *b = (char *) c ;
    printf("%d", *(b + 8));
}
```

The output of above program, when 'short int take '2B' and character take 1 B to represent _____. (Assume memory storage follows Little Endian architecture).

Q.31 Consider the following two functions:

```
void call1 (int n)
{
    if (n == 1) return 1;
    printf ("%d", n);
    call2 (– –n);
    printf ("%d", n);
}
```

```
void call2 (int n)
{
    if (n == 1) return 1;
    printf ("%d", n);
    call1 (– –n);
    printf ("%d", n);
}
```

The output printed when call2 (6) is called is

(a) 6 5 4 3 2 1 2 3 4 6 (b) 6 5 4 3 2 1 5 4 3 2

(c) 6 5 4 3 2 1 2 3 4 5 (d) 6 6 5 5 4 4 3 3 2 2

Q.32 Consider the following C program segment:

```
#include <stdio.h>
main( )
{
    static char *s[ ] = {"galvin", "rossen",
    "navathe", "cormen"};
    char ** ptr[ ] = {s + 2, s + 1, s + 3, s}, *** p;
    p = ptr;
    ++p;
    printf("%s", * – – *++p + 3);
}
```

What will be printed by the program?

(a) sen (b) nen

(c) vin (d) athe

Q.33 Consider the following functions, googly( ), doosra( ) and teesra( ). Note that, a variable has a bool type if it holds a value in {true, false}. Also, $\log_2(n)$ computes the base 2 logarithm of the input number n.

FUNCTION 1

```
bool doosra(int n)
{
    return (ceil(log2(n)) == floor(log2(n)));
}
```

FUNCTION 2

```
bool googly(int n)
{
    if (n == 0) return false;
    while (n! = 1)
{
    if (n% 2! = 0) return false;
    n = n/2;
}
```

```
    return true;
}
```

FUNCTION 3

```
bool teesra(int x)
{
    return x && (! (x &(x – 1)));
}
```

Which of the above functions produce the same output for a given input?

(a) Googly, Doosra (b) Doosra, Teesra
(c) All 3 are equivalent (d) None of these

Q.34 Consider the following 3 programs:

Program P_1 :

```
int *g(void) {
    int x = 10;
    return (&x);
}
```

Program P_2 :

```
int *g(void) {
    int *px;
    *px = 10;
    return px;
}
```

Program P_3 :

```
int *g(void) {
    int *px;
    px = (int*) malloc (sizeof (int));
    *px = 10; free(px);
    return px;
}
```

Which of the above three functions are likely to cause problems with pointers?

(a) Only P_2 (b) Only P_1 and P_3
(c) Only P_1 and P_2 (d) P_1, P_2 and P_3

Q.35 Consider the following program:

```
void MadeEasy (int n)
{
    printf("*");
    if (n > 1)
    {
        MadeEasy (n/4);
        MadeEasy (n/4);
        MadeEasy (n/4);
        MadeEasy (n/4);
    }
}
```

Let X be the number of asterisks printed by the above function when $n = 1024$. Then the value of X will be _____. (Hint: Take n as a power of 4)

Q.36 Consider the following C-code:

```
main ( )
    {
        char p[ ] = "MADEEASYTEST";
        char *y = p;
        printf("%s", p + y [7] – y [10]);
    }
```

What will be the output of above C program?

(a) MADEEASYTEST
(b) SYTEST
(c) TEST
(d) EASYTEST

Q.37 Which of the following is correct output for the program code given below?

```
main ( )
{
    void fun ( );
    fun( );
    fun ( );
}
void fun ( );
{
    static int i = 1;
    auto int j = 5;
    printf("%d", (i++));
    printf("%d", (j++));
}
```

(a) 1 5 2 6 3 7
(b) 2 6 3 7 4 8
(c) 1 5 6 1 7 1
(d) 1 5 2 5 3 5

Q.38 A 4-ary i.e., either has 0 children or has 4 children tree has 20 leaf nodes. Then the total number of nodes in the tree are _____.

Q.39 The output of following program is _____.

```
# include <stdio.h>
int main( )
{
    static int a[ ] = {90, 98, 99, 96, 84, 70};
    static int *p[ ] = {a+2, a+1, a, a+3, a+4, a+5};
    static int **S = {p+4, p+5, p+1, p, p+2, p+3};
    int *** ptr; ptr = S + 2;
    printf("%d", ***(ptr + 3) – **(p + 1));
}
```

Answer Key:

1. (b)	**2.** (c)	**3.** (b)	**4.** (d)	**5.** (b)
6. (d)	**7.** (a)	**8.** (a)	**9.** (b)	**10.** (d)
11. (b)	**12.** (b)	**13.** (1024)	**14.** (a)	**15.** (d)
16. (3)	**17.** (d)	**18.** (10)	**19.** (11)	**20.** (22)
21. (c)	**22.** (b)	**23.** (a)	**24.** (8)	**25.** (290)
26. (d)	**27.** (d)	**28.** (b)	**29.** (d)	**30.** (148)
31. (c)	**32.** (d)	**33.** (c)	**34.** (d)	**35.** (1365)
36. (b)	**37.** (d)	**38.** (27)	**39.** (–2)	

Student's Assignments Explanations

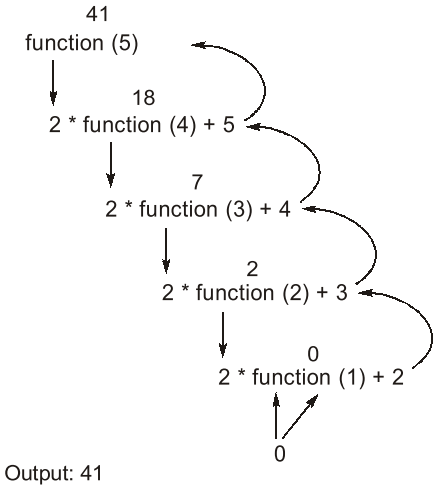
2. (c)

func(x) = x * x + x

x = 36 + 5 × 5 + 5 × 3 × 3 + 3

= 36 + 25 + 45 + 3 = 109

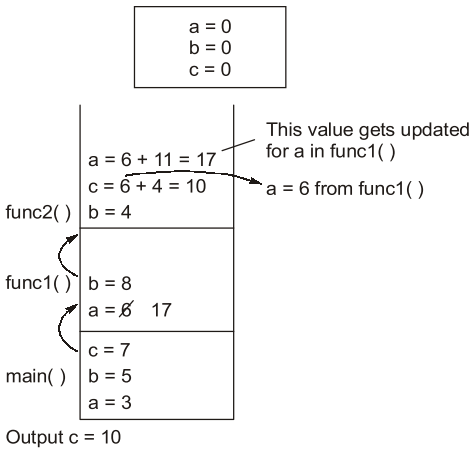
3. (b)



Output: 41

4. (d)

Global



Output c = 10

When func2( ) completes, it returns back to func1( )

In func1( ) → a = a + b + c = 17 + 8 + 7 = 32

Hence output → 10 32

5. (b)

It prints the first repeated character in the string.

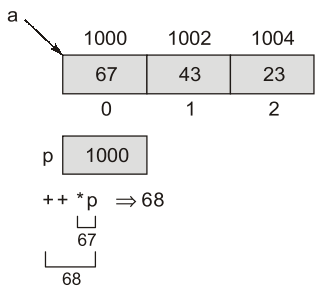
6. (d)

```
main( )
{
   int X=4.5;
   foo(4.5, 4.5);
}
   void foo(4, 4)
{
   X = 8
   Y = 16
}
```

Using call by value → 4 is returned.

Using call by reference → 16 is returned as both X and Y points to same value.

7. (a)



++ *p ⇒ 43

1000

1002

43

* p++ ⇒ 43

1002

1002 → Increment is pending

43

8. (a)

fun(10)

n = 20
i = 30
fun(20)

n = 50
i = 80
fun(50)

n = 130
i = 210
fun(130)

n = 340
i = 550
fun(340)

n + i = 340 + 550 = 890

i = ~~10~~ ~~30~~ ~~80~~ ~~210~~ 550

9. (b)

void fun(*p)

⇓

void fun(*&i)

⇓ * &i = i

void fun(i)

i = 0	p	1000	q	2000
1000		2000		3000

** q

2000

1000

0

= 0

10. (d)

1000	1001	1002	1003	1004	1005	1006	1007
G	r	a	d	u	a	t	e

p1 | 1000 | 2000

p2 | 1000 | 3000

* ++ p2

1000 ⇒ r

1001

r

sizeof(p2) → 8

11. (b)

main( )

b = 1
b = f(1)

print 1
returns 3

b = f(3)

print 3
returns 5

b = f(1 + f(b))

f(5)

print 5
returns 7

b = f(1 + 7) = f(8)

print (8)
returns 10

b = 10

Output: 1 3 5 8

b = ~~1~~ 3

12. (b)

A

1000	1002	1004
0	1	2
0	1	2

p | 1000 |

*(P + 2) ⇒ P[2] = 5

A

1000	1002	1004
0	1	5
0	1	2

P = A ++ = 1000

*P = 7 i.e., P[0] = 7

A

7	1	5