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

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
Eleventh Edition : 2025

Twelfth Edition : 2026

CONTENTS

Programming and Data Structures

CHAPTER 6

CHAPTER 7

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.

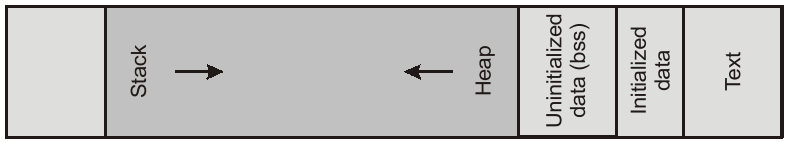
■■■■

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

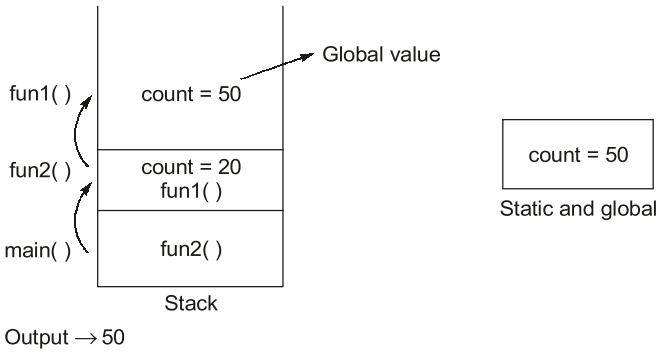
```
for(...){
    int i;
    {...
        {
            j = 5;
        }
    }
    ...
}
```

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:



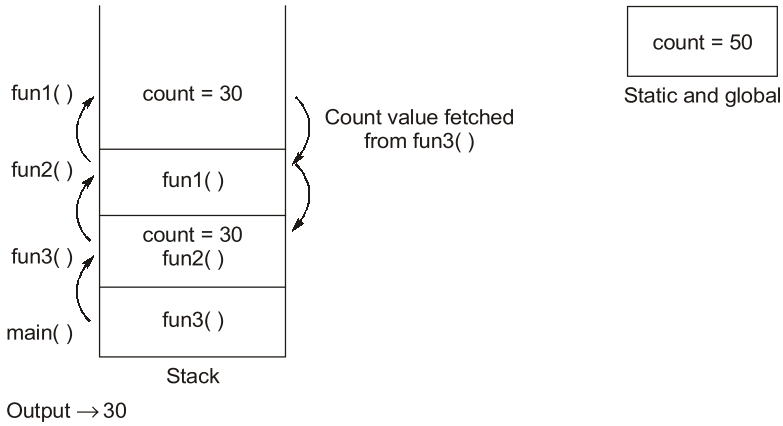
Dynamic Scoping: Dynamic scoping does not care how the code is written, but instead how it executes.

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 :



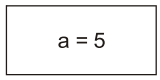
Example 1.7 What will be the output of following program using static and dynamic scoping?

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);
}
```

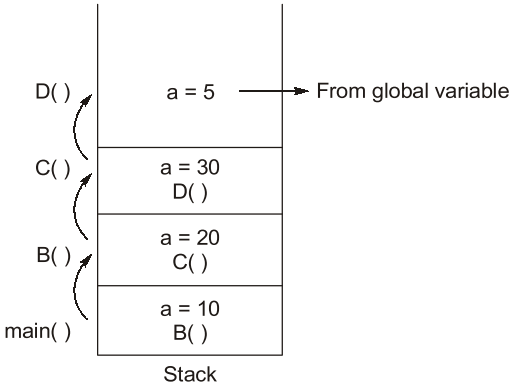
Solution :



Static and global

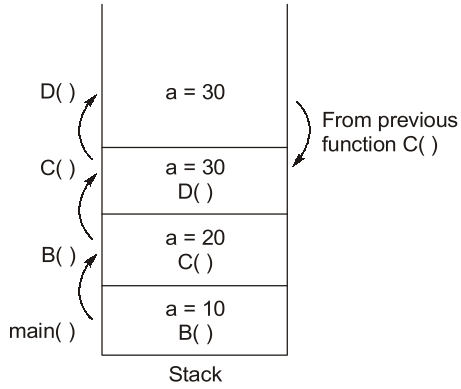
Static is global

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.

Rule 3: Name of identifier cannot be any keyword of C program.
Valid name: INT, FLOAT, etc. Invalid name: int, float, enum etc.

Rule 4: Name of function cannot be global identifier.
Valid name: __TOTAL__, __NAME__ , __TINY__etc.
Invalid name: __TIME__ ,__DATE__ , __FILE__ ,__LINE__ ,__STDC__, __TINY__, __SMALL__, __COMPACT__, __LARGE__, __HUHE__, __CDECL__, __PASCAL__, __MSDOS__, __TURBOC__

Rule 5: Name of identifier cannot be register Pseudo variables

Rule 6: Name of identifier cannot be exactly same as of name of function within the scope of the function.

Rule 7: Name of identifier is case sensitive i.e. num and Num are two different variables.

Rule 8: Only first 32 characters are significant of identifier name.
Example: abcdefghijklmnopqrstuvwxyz123456aaa.

Rule 9: Identifier name cannot be exactly same as constant name which have been declared in header file of C and you have included that header files.

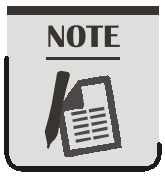
- Variable name cannot be exactly same as function name which have been declared any header file of C and we have included that header file in our program and used that function also in the program.
- Identifier name in C can be exactly same as data type which has been declared in any header of C.

Example 1.10 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int main()`
`{`
`  int float=5;`
`  printf("%d",float);`
`return 0;`
`}`
	Output: compilation error
	Here we have used float as variable name. So it will give compile time error.

Address of a Variable in C: Location in a memory where a variable stores its data or value is known as address of variable. To know address of any variable C has provided a special unary operator and which is known as dereference operator or address operator. It is only used with variables not with the constant.



- Visibility of variables explained in the following program.

```
#include<stdio.h>
int main(){
    int a=5;
    printf("Address of variable a is: %d",&a);
    return 0;
}
```

We cannot write: &&a, because: &&a=&(&a)=&(address of variable a)=&(a constant number) and we cannot use address operator with constant.

1.3.3 Important Points about Address of Variables in C

1. Address of any variable in C is an unsigned integer. It cannot be a negative number. So in printf statement we should use: %*u* instead of %*d*, to print the address of any variable. %*d*: It is used to. Print signed decimal number. %*u*: It is used to print unsigned decimal number.

2. Address of any variable must be within the range 0000 to FFFF in hexadecimal number format or 0 to 65535 i.e. range of unsigned short int in C. To print the address of any variable in hexadecimal number format by printf function we should use %x or %X.
   %x: To print a number in hexadecimal format using 0 to 9 and a, b, c, d, e, f.
   %X: To print a number in hexadecimal format using 0 to 9 and A, B, C, D, E, F.
3. A programmer cannot know at which address a variable will store the data. It is decided by compiler or operating system.
4. Any two variables in C cannot have same physical address.
5. Address of any variable in C is not integer type so to assign an address to a integral variable we have to type cast the address.

Example 1.11 What will be output of following program?

Code	Solution

`#include<stdio.h>`
`int main(){`
`  int a=100;`
`  unsigned int b=(unsigned)&a;`
`  printf("%u",b);`
`  return 0;`
`}`
	Output: Address of the variable a.

1.4 OPERATORS IN C

1.4.1 Bitwise Operators in C

In C bitwise operators work at bit-level. They are as follows:

(*i*) **& (bitwise AND):** '&' operator takes two numbers as input and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

(*ii*) **| (bitwise OR):** '|' operator takes two numbers as input and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

(*iii*) **∧ (bitwise XOR):** '∧' operator takes two numbers as input and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

(*iv*) **<< (left shift):** '<<' operator takes two numbers as input, left shifts the bits of first number, the second number decides the number of places to shift.

(*v*) **>> (right shift):** '>>' operator takes two numbers as input, right shifts the bits of first number, the second number decides the number of places to shift.

(*vi*) **~(bitwise NOT):** '~' operator takes one number as input and inverts all bits of it.

Example 1.12 What will be output of following C code?

```
#include<stdio.h>
int main()
{
   char a = 7, b = 11;
   printf("a = %d, b = %d\n", a, b);
```

cannot change value at present ptr. So this kind of string should only be used when we do not want to modify string at any point of time in the program.

(*ii*) **Dynamically allocated in heap segment:** Strings are stored like other dynamically allocated things in C and can be shared among functions.

```
char *ptr;
int length = 9;              /* one extra for '/0' */      [we assume character take one byte]
ptr = (char *malloc (sizeof (char) * length);
*(ptr + 0) = 'm';   *(ptr + 1) = 'a';   *(ptr + 2) = 'd';   *(ptr + 3) = 'e';   *(ptr + 4) = 'e';
*(ptr + 5) = 'a';   *(ptr + 6) = 's';   *(ptr + 7) = 'y';   *(ptr+8) = '/0';
```

Summary

- **'Auto' Storage Class:** The auto storage class is implicitly the default storage calls used simply specifies a normal local variable which is visible within its own code block only and which is created and destroyed automatically upon entry and exit respectively from the code block.
- **'Static' Storage Class:** The static storage class causes a local variable to become permanent within its own code block i.e. it retains its memory space and hence its value between function calls.
- **'Register' Storage Class:** The register storage class also specifies a normal local variable but it also requests that the compiler store a variable so that it may be accessed as quickly as possible, possible from a CPU register.
- **'Extern' Storage Class:** An extern class which informs the compiler for the existence of the global variable which enables it can be accessed in more that one source file.
- **Control Statements:** 1. **if** statement, 2. Nested **if** statement, 3. Conditional operator?, 4. **switch** statement, 5. **for** statement, 6. **while** statement, 7. **do-while** statement, 8. **break** statement and 9. **continue** statement.
- **Call by value:** "Actuals copied to formal", but not formals to actuals.
- **Call by reference:** "Actuals and formals uses same address space".
- **Call by value result (Restore):** "Actuals copied to formals and formals copied to actuals".
- **Call by Result:** "Actuals not copied to formals", but formals copied to actuals.
- **Call by Constant:** Formal values are never changed.
- **Call by Name:** The textual substitution of every occurrence of a formal parameter in the called routines body by the corresponding actual parameter.
- **Call by Text:** Same as call by name, but if variables are named same as local variables then it prefers local.
- **Call by Need:** Memory allocates only if formals are used in function.
- **Static Scoping:** The method of binding names to non local variables called static scoping. Scope of variable can be statically determined prior to execution.
- **Dynamic Scoping:** It is based on the calling sequence of subprograms. Scope can be determined at runtime.
- **Pointer:** A pointer is a variable that is used to store a memory address of another variable in memory. If one variable holds the address of another then it is said to the second variable.

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;
```

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

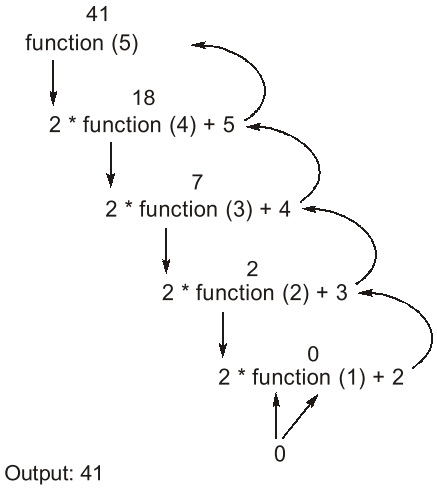
2. (c)

func(x) = x * x + x

x = 36 + 5 × 5 + 5 × 3 × 3 + 3

= 36 + 25 + 45 + 3 = 109

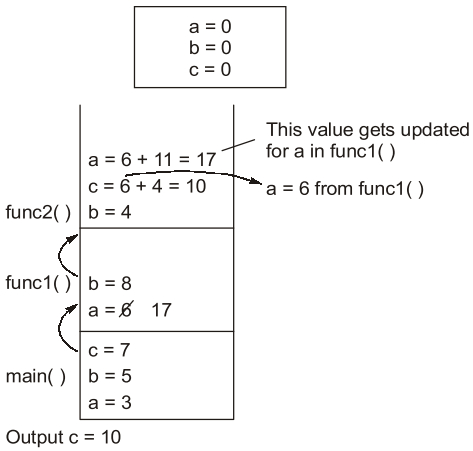
3. (b)



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)