APPENDIX A

Memory Allocation in C Programs

C supports three kinds of memory allocation through the variables in C programs:

Static allocation When we declare a static or global variable, static allocation is done for the variable. Each static or global variable is allocated a fixed size of memory space. The number of bytes reserved for the variable cannot change during execution of the program.

Automatic allocation When we declare an automatic variable, such as a function argument or a local variable, automatic memory allocation is done. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when it exits from a compound statement.

Dynamic allocation A third important kind of memory allocation is known as *dynamic allocation.* In the following sections we will read about dynamic memory allocation using pointers.

Memory Usage

Before jumping into dynamic memory allocation, let us first understand how memory is used. Conceptually, memory is divided into two parts—program memory and data memory (Fig. A1).

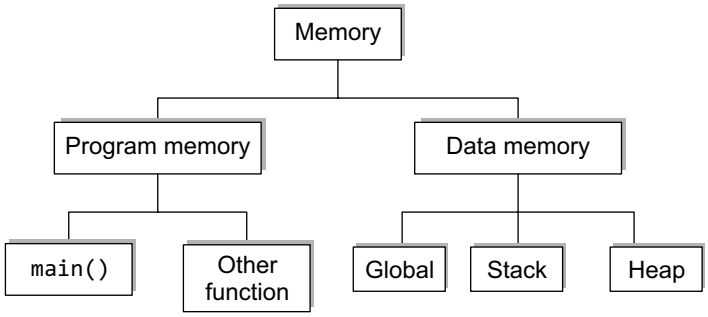


Figure A1 Memory usage

The program memory consists of memory used for the `main()` and other called functions in the program, whereas data memory consists of memory needed for permanent definitions such as global data, local data, constants, and dynamic memory data. The way in which `C` handles the memory requirements is a function of the operating system and the compiler.

When a program is being executed, its `main()` and all other functions are always kept in the memory. However, the local variables of the function are available in the memory only when they are active. When we studied recursive functions, we have seen that the system stack is used to store a single copy of the function and multiple copies of the local variables.

Apart from the stack, we also have a memory pool known as heap. Heap memory is unused memory allocated to the program and available to be assigned during its execution. When we dynamically allocate memory for variables, heap acts as a memory pool from which memory is allocated to those variables.

However, this is just a conceptual view of memory and implementation of the memory is entirely in the hands of system designers.

Dynamic Memory Allocation

The process of allocating memory to the variables during execution of the program or at run time is known as *dynamic memory allocation*. C language has four library routines which allow this function.

Till now whenever we needed an array we had declared a static array of fixed size as

```
int arr[100];
```

When this statement is executed, consecutive space for 100 integers is allocated. It is not uncommon that we may be using only 10% or 20% of the allocated space, thereby wasting rest of the space. To overcome this problem and to utilize the memory efficiently, C language provides a mechanism of dynamically allocating memory so that only the amount of memory that is actually required is reserved. We reserve space only at the run time for the variables that are actually required. Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance.

C provides four library routines to automatically allocate memory at the run time. These routines are shown in Table A1.

Table A1 Memory allocation/de-allocation functions

Function	Task
malloc()	Allocates memory and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
free()	Frees previously allocated memory
realloc()	Alters the size of previously allocated memory

When we have to dynamically allocate memory for variables in our programs then pointers are the only way to go. When we use malloc() for dynamic memory allocation, then you need to manage the memory allocated for variables yourself.

Memory Allocations Process

In computer science, the free memory region is called heap. The size of heap is not constant as it keeps changing when the program is executed. In the course of program execution, some new variables are created and some variables cease to exist when the block in which they were declared is exited. For this reason it is not uncommon to encounter memory overflow problems during dynamic allocation process. When an overflow condition occurs, the memory allocation functions mentioned above will return a null pointer.

Allocating a Block of Memory

Let us see how memory is allocated using the malloc()function. malloc is declared in <stdlib.h>, so we include this header file in any program that calls malloc. The malloc function reserves a block

of memory of specified size and returns a pointer of type `void`. This means that we can assign it to any type of pointer. The general syntax of `malloc()` is

```
ptr =(cast-type*)malloc(byte-size);
```

where `ptr` is a pointer of type `cast-type`. `malloc()` returns a pointer (of cast type) to an area of memory with size `byte-size`.

For example,

```
arr=(int*)malloc(10*sizeof(int));
```

This statement is used to dynamically allocate memory equivalent to 10 times the area of `int` bytes. On successful execution of the statement the space is reserved and the address of the first byte of memory allocated is assigned to the pointer `arr` of type `int`.

Programming Tip

To use dynamic memory allocation functions, you must include the header file stdlib.h.

`calloc()` function is another function that reserves memory at the run time. It is normally used to request multiple blocks of storage each of the same size and then sets all bytes to `zero`. `calloc()` stands for contiguous memory allocation and is primarily used to allocate memory for arrays. The syntax of `calloc()` can be given as:

```
ptr=(cast-type*) calloc(n,elem-size);
```

The above statement allocates contiguous space for `n` blocks each of size `elem-size` bytes. The only difference between `malloc()` and `calloc()` is that when we use `calloc()`, all bytes are initialized to zero. `calloc()` returns a pointer to the first byte of the allocated region.

When we allocate memory using `malloc()` or `calloc()`, a `NULL` pointer will be returned if there is not enough space in the system to allocate. A `NULL` pointer, points definitely nowhere. It is a *not a pointer* marker; therefore, it is not a pointer you can use. Thus, whenever you allocate memory using `malloc()` or `calloc()`, you must check the returned pointer before using it. If the program receives a `NULL` pointer, it should at the very least print an error message and exit, or perhaps figure out some way of proceeding without the memory it asked for. But in any case, the program cannot go on to use the `NULL` pointer it got back from `malloc()`/`calloc()`.

A call to `malloc`, with an error check, typically looks something like this:

```
int *ip = malloc(100 * sizeof(int));
if(ip == NULL)
{
    printf("\n Memory could not be allocated");
    return;
}
```

Write a program to read and display values of an integer array. Allocate space dynamically for the array.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, n;
    int *arr;
    printf("\n Enter the number of elements ");
    scanf("%d", &n);
    arr = (int*)malloc(n * sizeof(int));
    if(arr == NULL)
    {
        printf("   \n Memory Allocation Failed");
        exit(0);
    }
```

```
    for(i=0;i<n;i++)
    {
        printf("\n Enter the value %d of the array: ", i);
        scanf("%d",&arr[i]);
    }
    printf("\n The array contains \n");
    for(i=0;i<n;i++)
        printf("%d", arr[i]);
    return 0;
}
```

Now let us also see how we can allocate memory using the `calloc` function. The `calloc()` function accepts two parameters—`num` and `size`, where `num` is the number of elements to be allocated and `size` is the size of elements. The following program demonstrates the use of `calloc()` to dynamically allocate space for an integer array.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i,n;
    int *arr;
    printf ("\n Enter the number of elements: ");
    scanf("%d",&n);
    arr = (int*) calloc(n,sizeof(int));
    if (arr==NULL)
        exit (1);
    printf("\n Enter the %d values to be stored in the array", n);
    for (i = 0; i < n; i++)
     scanf ("%d",&arr[i]);
    printf ("\n You have entered: ");
    for(i = 0; i < n; i++)
        printf ("%d",arr[i]);
    free(arr);
    return 0;
}
```

Releasing the Used Space

When a variable is allocated space during the compile time, then the memory used by that variable is automatically released by the system in accordance with its storage class. But when we dynamically allocate memory then it is our responsibility to release the space when it is not required. This is even more important when the storage space is limited. Therefore, if we no longer need the data stored in a particular block of memory and we do not intend to use that block for storing any other information, then as a good programming practice we must release that block of memory for future use, using the `free` function. The general syntax of the `free()function` is,

```
free(ptr);
```

where `ptr` is a pointer that has been created by using `malloc()` or `calloc()`. When memory is de-allocated using `free()`, it is returned back to the free list within the heap.

To Alter the Size of Allocated Memory

At times the memory allocated by using `calloc()` or `malloc()` might be insufficient or in excess. In both the situations we can always use `realloc()` to change the memory size already allocated by `calloc()` and `malloc()`. This process is called *reallocation of memory*. The general syntax for `realloc()` can be given as,

```
ptr = realloc(ptr,newsize);
```

The function `realloc()` allocates new memory space of size specified by `newsize` to the pointer variable `ptr`. It returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region. Thus, we see that `realloc()` takes two arguments. The first is the pointer referencing the memory and the second is the total number of bytes you want to reallocate. If you pass zero as the second argument, it will be equivalent to calling `free()`. Like `malloc()` and `calloc()`, `realloc` returns a void pointer if successful, else a `NULL` pointer is returned.

If `realloc()` was able to make the old block of memory bigger, it returns the same pointer. Otherwise, if `realloc()` has to go elsewhere to get enough contiguous memory then it returns a pointer to the new memory, after copying your old data there. However, if `realloc()` cannot find enough memory to satisfy the new request at all, it returns a null pointer. So again you must check before using that the pointer returned by the `realloc()` is not a null pointer.

```
/*Example program for reallocation*/
#include < stdio.h>
#include < stdlib.h>
#define NULL 0
int main()
{
    char *str;
    str = (char *)malloc(10);
    if(str==NULL)
    {
        printf("\n Memory could not be allocated");
        exit(1);
}
    strcpy(str,"Hi");
    printf("\n STR = %s", str);
    /*Reallocation*/
    str = (char *)realloc(str,20);
    if(str==NULL)
    {
        printf("\n Memory could not be reallocated");
        exit(1);
}
printf("\n STR size modified\n");
printf("\n STR = %s\n", str);
strcpy(str,"Hi there");
printf("\n STR = %s", str);
/*freeing memory*/
free(str);
return 0;
}
```

Note With realloc(), you can allocate more bytes without losing your data.

Dynamically Allocating a 2-D Array

We have seen how `malloc()` can be used to allocate a block of memory which can simulate an array. Now we can extend our understanding further to do the same to simulate multidimensional arrays.

If we are not sure of the number of columns that the array will have, then we will first allocate memory for each row by calling `malloc`. Each row will then be represented by a pointer. Look at the code below which illustrates this concept.

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

```
int main()
{
    int **arr,i,j,ROWS,COLS;
    printf("\n Enter the number of rows and columns in the array: ");
    scanf("%d %d",ROWS,COLS);
    arr = (int **)malloc(ROWS * sizeof(int *));
    if(arr == NULL)
    {
        printf("\n Memory could not be allocated");
        exit(-1);
    }
    for(i=0; i<ROWS; i++)
    {
        arr[i] = (int *)malloc(COLS * sizeof(int));
        if(arr[i] == NULL)
        {
            printf("\n Memory Allocation Failed");
            exit(-1);
        }
    }
    printf("\n Enter the values of the array: ");
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
            scanf("%d",&arr[i][j]);
    }
    printf("\n The array is as follows: ");
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
            printf("%d",arr[i][j]);
    }
    for(i = 0; i < ROWS; i++)
        free(arr[i]);
    free(arr);
    return 0;
}
```

Here, `arr` is a pointer-to-pointer-to-`int`: at the first level as it points to a block of pointers, one for each row. We first allocate space for rows in the array. The space allocated to each row is big enough to hold a `pointer-to-int`, or `int *`. If we successfully allocate it, then this space will be filled with pointers to columns (number of `int`s). This can be better understood from Fig. A2.

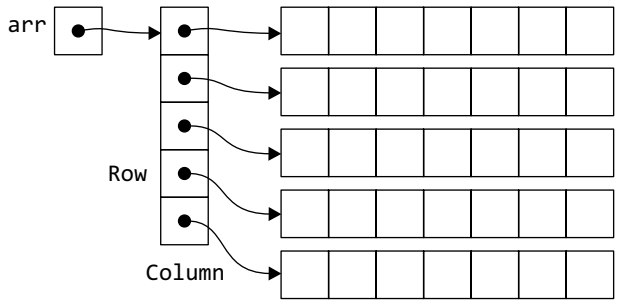


Figure A2 Memory allocation of two-dimensional array

Once the memory is allocated for the two-dimensional array, we can use subscripts to access its elements. When we write, `arr[i][j]`, it means we are looking for the `i`th pointer pointed to by `arr`, and then for the `j`th `int` pointed to by that inner pointer.

When we have to pass such an array to a function, then the prototype of the function will be written as

```
func(int **arr, int ROWS, int COLS);
```

In the above declaration, `func` accepts a `pointer-to-pointer-to-int` and the dimensions of the arrays as parameters, so that it will know how many rows and columns are there.

APPENDIX B

Garbage Collection

Garbage collection is a dynamic approach used for automatic memory management to reduce the memory leak problems. The garbage collection process identifies unused memory blocks and reallocates that storage for reuse. Garbage collection is implemented using the following approaches:

Mark-and-sweep In this approach when memory runs out, the garbage collection process locates all accessible memory and then reclaims the available memory.

Reference counting The garbage collection process here maintains a reference count of referencing number for each allocated object. When the memory count becomes zero, the object is marked as garbage and then destroyed by freeing its memory. The freed memory is finally returned to the memory heap.

Copy collection The garbage collection process maintains two memory partitions. When the first partition is full, the garbage collection process identifies all accessible data structures and copies them to the second partition. Then it compacts memory to allow continuous free memory.

Note Some programming languages like Java, C#, .NET, etc., have built-in garbage collection process to self-manage memory leak problem.

Advantages of Garbage Collection

Garbage collection frees the programmer from manually dealing with memory de-allocation, thereby eliminating or substantially reducing following types of programming bugs:

***Dangling pointer* bugs** are often encountered when the memory allocated to a variable (or an object) is freed but there are still pointers pointing to it. If such pointers are de-referenced especially when that memory is re-allocated to another variable or object, then results are simply unpredictable.

Double free bugs occur when the program tries to free a piece of memory that has already been freed. It may be a case that the freed memory has now been re-allocated to some other variable or object of the same or a different program. In such a case, the program will again give erroneous results.

Memory leaks occur when a program is unable to free memory occupied by objects that have become unreachable.

Garbage collection also helps in efficient implementation of persistent data structures.

Disadvantages of Garbage Collection

The typical disadvantages of garbage collection process include:

- Garbage collection consumes computing resources to decide which piece of memory must be freed. This information may already be available with the programmer.
- The time at which the garbage collection process will be executed is unpredictable which may lead to stalls scattered throughout a session. This is unacceptable in real-time environments, transaction processing, or in interactive programs. Although incremental, concurrent, and real-time garbage collectors solve this problem but with some or the other trade-off.
- Some of the bugs addressed by garbage collection can have certain security implications.

Requirements for Automatic Garbage Collection

An effective and efficient garbage collection process must have the following properties:

- Must identify garbage
- The object or variable identified as garbage must actually be garbage.
- Must have less overhead
- During garbage collection, the execution of the program is temporarily delayed. This delay must be minimum.

APPENDIX C

Backtracking

Backtracking is a general algorithm that finds all or some solutions to any computational problem that incrementally builds candidates to the solutions. At each step, while the valid candidates to the solution are extended, the other candidates are discarded. That is, each partial candidate *c* is immediately abandoned after it is determined that *c* cannot be a possible and valid solution to the problem. Hence, the name, backtrack.

The concept of backtracking is applicable only to problems that support partial candidate solutions and a relatively quick test to determine if the partial candidate solution can be completed to a valid solution. Backtracking is extensively used to solve constraint satisfaction problems such as following:

- Puzzles like Sudoku, eight queen's problem, crosswords, verbal arithmetic, Solitaire
- Combinational optimization problem like parsing and Knapsack problem
- Logic programming languages that internally use backtracking include Icon, Planner and Prolog
- *diff* which is a version comparing engine for the MediaWiki software

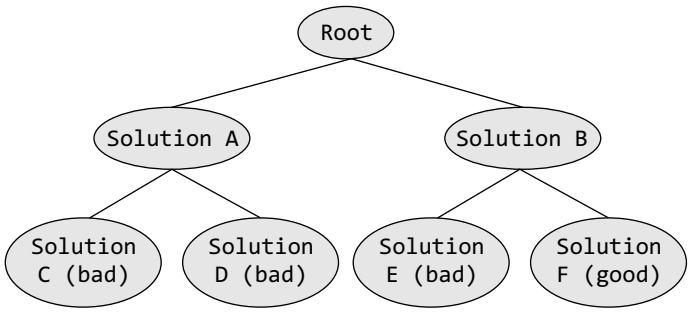
Backtracking is said to be a meta-heuristic algorithm (in contrast to a specific algorithm) that is guaranteed to find all solutions to a finite problem in a bounded amount of time. The term meta-heuristic implies that the algorithm works based on user-given procedures that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates.

Conceptualizing the backtracking process as potential search tree, the partial candidates of the solution can be viewed as the nodes of the tree. Each partial candidate has child nodes that represent the candidates that differ from it by a single extension step. Of course, the leaf nodes are the partial candidates that cannot be extended further.

The backtracking algorithm recursively traverses the search tree in depth-first order (starting from the root node). At each node *c*, the algorithm checks if *c* can be completed to a valid solution. If it cannot be completed, then the entire sub-tree rooted at *c* is pruned. However, if there is a possibility that *c* may lead to valid solution then the algorithm first checks if *c* itself is a valid solution. If yes, then the algorithm declares *c* as the valid solution otherwise it recursively enumerates all sub-trees of *c*.

Note The tests to determine the valid solution and children of each node are all defined by user-given procedures.

Look at the search tree given below.



Step 1 Start with the root node. The two available options are—Solution A and Solution B. Select Solution A.

Step 2 At Solution A there are again two options—Solution C and Solution D. Select Solution C.

Step 3 Since Solution C is not a good (valid and possible) candidate, backtrack to Solution A and now select Solution D.

Step 4 Since Solution D is not a good (valid and possible) candidate, backtrack to Solution A. now there are no more options available at Solution A, so select Solution B.

Step 5 At Solution B there are two options—Solution E and Solution F. Select Solution E.

Step 6 Since Solution E is not a good (valid and possible) candidate, backtrack to Solution B and now select Solution F.

Step 7 Solution F is a good (valid and possible) candidate, so the algorithm stops here.

Advantage: By terminating searches and not exploring candidates that cannot lead to valid possible solutions, the backtracking technique reduces the number of nodes examined. The algorithm can be used to solve exponential time problems in a *reasonable* amount of time.

Note Backtracking is not an optimization technique. It just reduces the search space.

APPENDIX D

Josephus Problem

In real-world applications, especially in the operating system, multiple activities have to be performed. The biggest issue is not just performing these activities but also completing them in minimum time. The Johnson's algorithm is used in such applications where an optimal order of execution of different activities has to be determined.

Consider a problem that consists of independent tasks $T_1, T_2, \ldots, T_n$ and two independent processes P_1 and P_2. If it is specified that P_1 must be completed before P_2, then Johnson's problem can be given as:

Step 1 Determine P_1 and P_2 times for each task.

Step 2 Make two queues, Q_1 and Q_2 where Q_1 is formed at the beginning of the schedule and Q_2 is formed at its end.

Step 3 For each task, analyse P_1 and P_2 times to determine the smallest time. If P_1 is the smallest time, then insert the corresponding task at the end of Q_1. Otherwise, insert the corresponding task at the beginning of Q_2. In case of a tie, take P_1 as the smallest time.

Note If there is a tie between multiple P1 or multiple P2 times, select the first task in the list.

Consider the table given below, which specifies the tasks and time it takes to complete processes P_1 and P_2.

TASK	TIME TO PERFORM P_1	TIME TO PERFORM P_2
0	17	6
1	24	12
2	5	8
3	14	10
4	11	8
5	14	11

Now, for each task, analyse P_1 and P_2 times to determine the smallest time. Tasks with P_1 time less than P_2 are assigned to the head of Q_1, other tasks are assigned to the tail of Q_2.

TASK	TIME TO PERFORM P_1	TIME TO PERFORM P_2	MINTIME	LOCATION
0	17	6	6	TAIL
1	24	12	12	TAIL

2	5	8	5	HEAD
3	14	10	10	TAIL
4	11	8	8	TAIL
5	14	11	11	TAIL

Sort the table using the MINTIME field.

TASK	TIME TO PERFORM P_1	TIME TO PERFORM P_2	MINTIME	LOCATION
2	5	8	5	HEAD
0	17	6	6	TAIL
4	11	8	8	TAIL
3	14	10	10	TAIL
5	14	11	11	TAIL
1	24	12	12	TAIL

There is an alternative implementation strategy which states that if LOCATION has the value TAIL then the task is added at the front of Q_2. Once all the tasks are assigned, HEAD and TAIL can be concatenated to create the final complete QUEUE.

When calculating the efficiency of the Johnson's algorithm, we see that the data is processed as one observation at a time, thereby taking `O(n)` time where `n` is the volume of the data. Then the data must be sorted which will again take at least `O(n × log n)` time. Since, `O(n × log n)` term dominates. Johnson's algorithm gives an optimal schedule in `O(n log n)` time.

APPENDIX E

File Handling in C

1. Write a program to store records of an employee in employee file. The data must be stored using binary file.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    typedef struct employee
    {
        int emp_code;
        char name[20];
    };
    FILE *fp;
    struct employee e[2];
    int i;
    fp = fopen("employee.txt", "wb");
    if(fp==NULL)
    {
        printf("\n Error opening file");
        exit(1);
    }
    printf("\n Enter the details employees");
    for(i = 0;i < 2;i++)
    {
        printf("\n\n Enter the employee code:");
        scanf(„%d", &e[i].emp_code);
        printf("\n\n Enter the name of the employee: ");
        scanf("%s", e[i].name);
        fwrite(&e[i], sizeof(e[i]), 1, fp);
    }
    fclose(fp);
    getch();
    return 0;
}
```

Output

```
Enter the details of employees
Enter the employee code: 01
Enter the name of the employee: Gargi
Enter the employee code: 02
Enter the name of the employee: Nikita
```

2. Write a program to read the records stored in ‘employee.txt’ file in binary mode.

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

```
int main()
{
    typedef struct employee
    {
        int emp_code;
        char name[20];
    };
    FILE *fp;
    struct employee e;
    int i;
    clrscr();
    fp = fopen("employee.txt", "rb");
    if(fp==NULL)
    {
        printf("\n Error opening file");
        exit(1);
    }
    printf("\n THE DETAILS OF THE EMPLOYEES ARE ");
    while(1)
    {
        fread(&e, sizeof(e), 1, fp);
        if(feof(fp))
        break;
        printf("\n\n Employee Code: %d", e.emp_code);
        printf("\n\n Name: %s", e.name);
    }
    fclose(fp);
    getch();
    return 0;
```

Output

```
Employee Code: 01
Name: Gargi
Employee Code: 02
Name: Nikita
```

APPENDIX F

Address Calculation Sort

Write a program to sort elements of an array using address calculation sort.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
struct node
{
    int data;
    struct node *next;
}*nodes[10]={NULL};

struct node *insert(struct node *start, int num)
{
    struct node *ptr,*new_node;
    ptr=start;
    new_node = (struct node*)malloc(sizeof(struct node));
    new_node->data=num;
    new_node->next=NULL;
    if(start==NULL)
        start = new_node;
    else
    {
        //insert the new node at its right position
        while((ptr->next->data<=num) && (ptr->next!=NULL))
            ptr=ptr->next;
        if(new_node->data < ptr->data)
        {
            new_node->next=ptr;
            start=new_node;
        }
        else
        {
            new_node->next=ptr->next;
            ptr->next=new_node;
        }
    }
    return start;
}
void addr_calc_sort(int arr[],int n)
{
    int i,j=0,pos;
    for(i=0;i<n;i++)
    {
        pos = arr[i] / 10;
```

```
        nodes[pos]=insert(nodes[pos],arr[i]);
    }
    for(i=0;i<10;i++)
    {
        while(nodes[i]!=NULL)
        {
            arr[j++]=nodes[i]->data;
            nodes[i]=nodes[i]->next;
        }
    }
    printf("\nSorted output is: ");
    for(i=0;i<n;i++)
        printf("%d\t",arr[i]);
    getch();
}
void main()
{
    int arr[MAX],i,n;
    printf("\n Enter the number of elements : ");
    scanf("%d",&n);
    printf("\n Enter the elements : ");
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    addr_calc_sort(arr,n);
}
```

Output

```
Enter the number of elements : 5
Enter the elements: 23  53  14  78  22
Sorted output is : 14  22  23  53  78
```

APPENDIX G

Answers

Chapter 1

Multiple-choice Questions

1. (b) 2. (c) 3. (d) 4. (d) 5. (b) 6. (b) 7. (d)
8. (c) 9. (c) 10. (d) 11. (c) 12. (b) 13. (c) 14. (a)
15. (b)

True or False

1. False 2. True 3. False 4. False 5. True 6. False 7. True
8. False 9. True 10. False 11. False 12. False 13. True 14. True
15. False 16. False 17. True 18. False 19. True 20. False 21. True
22. True 23. True 24. True 25. True

Fill in the Blanks

1. Dennis Ritchie
2. main()
3. ASCII codes
4. Operating system
5. Modulus operator (%)
6. Unary
7. Typecasting
8. Default case
9. printf()
10. Closing bracket
11. \n
12. const
13. sizeof
14. %hd
15. %x
16. –
17. Calling function
18. Calling function
19. Arguments/parameters
20. Function header and function body
21. Call by reference
22. I byte
23. NULL
24. Rvalue
25. *

Chapter 2

Multiple-choice Questions

1. (a) 2. (c) 3. (a) 4. (b) 5. (a) 6. (b) 7. (a)
8. (d) 9. (d) 10. (b) 11. (b) 12. (a) 13. (b)

True or False

1. False 2. True 3. False 4. False 5. False 6. True 7. False
8. False 9. False 10. False 11. False 12. True 13. False 14. False
15. False

Fill in the Blanks

1. Data structures
2. Functions
3. Arrays
4. Data type
5. Root = NULL
6. Considered apart from the detailed specifications or implementation
7. Input size
8. Amortized case
9. Index or subscript
10. Top
11. Peep or peek
12. An attempt is made to insert an element in an array, stack or queue that is already full.
13. Queue
14. Rear, front
15. Linear data structure
16. Program
17. Time complexity
18. Average case running time
19. O(1)
20. Modules
21. Top down
22. Worst
23. Omega
24. Non-asymptotically
25. Is in

Chapter 3

Multiple-choice Questions

1. (b)
2. (b)
3. (d)
4. (b)
5. (b)
6. (d)

True or False

1. True
2. True
3. True
4. False
5. False
6. True
7. True
8. True
9. False
10. True
11. True
12. False
13. False
14. False
15. True

Fill in the Blanks

1. Index or subscript
2. Consecutive
3. *n*
4. Pointer
5. Data type, name, and size
6. Base address
7. The number of elements stored in it
8. Array of arrays
9. Integral value
10. Fourth

Chapter 4

Multiple-choice Questions

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

True or False

1. False
2. False
3. True
4. True
5. False
6. False
7. True
8. False
9. True
10. False
11. True
12. False
13. True
14. False
15. False

Fill in the Blanks

1. A null-terminated character array
2. Null character
3. 5
4. zero
5. Consecutive
6. 99
7. scanf()
8. 65-97
9. Convert a character into upper case
10. When in dictionary order S1 will come after S2
11. strrev()
12. Morning
13. 15
14. Index operation
15. str2 is less than str1
16. strlen
17. puts

Chapter 5

Multiple-choice Questions

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

True or False

1. False 2. False 3. True 4. True 5. True 6. False 7. True
8. False 9. False 10. True 11. True 12. True 13. True 14. False

Fill in the Blanks

1. User defined
2. Structure declaration
3. Structure name
4. Typedef
5. Zero
6. Null character
7. Dot operator
8. Nested structure
9. Self-referential
10. We declare a variable of a structure
11. Union
12. Refer to the individual members of structure or union
13. Union

Chapter 6

Multiple-choice Questions

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

True or False

1. True 2. True 3. False 4. False 5. False 6. False 7. True
8. False 9. True 10. True

Fill in the Blanks

1. AVAIL 2. O(1) 3. O(n) 4. Two 5. One 6. Two 7. Two
8. One 9. Two 10. One 11. Node 12. START 13. Node
14. There is no memory that can be allocated for the new node to be inserted 15. First

Chapter 7

Multiple-choice Questions

1. (a) 2. (b) 3. (a) 4. (c)

True or False

1. False 2. True 3. True 4. True 5. False 6. True 7. False
8. True 9. False 10. True 11. False 12. False

Fill in the Blanks

1. stack
2. stack
3. O(n)
4. We try to delete a node from a stack that is empty
5. Left to right
6. Non-tail
7. Directly

Chapter 8

Multiple-choice Questions

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

True or False

1. False 2. False 3. False 4. True 5. False 6. True 7. True
8. True

Fill in the Blanks

1. Rear
2. Dequeue
3. O(1)
4. Input restricted dequeue
5. Circular array or a circular doubly linked list
6. Priority queues
7. Queues

Chapter 9

Multiple-choice Questions

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

True or False

1. True 2. True 3. True 4. False 5. True 6. False 7. True
8. False

Fill in the Blanks

1. Ascendant
2. Nodes
3. 2^{k-1}
4. Two
5. Siblings
6. Structure and contents
7. n and $\log_2(n+1)$.
8. Each node in the tree has either no child or exactly two children
9. Preorder
10. The estimated probability of occurrence for each possible value of the source character.

Chapter 10

Multiple-choice Questions

1. (a) 2. (b) 3. (b) 4. (d) 5. (a)

True or False

1. True 2. False 3. False 4. True 5. True 6. False 7. True
8. False 9. False 10. False

Fill in the Blanks

1. Two way threaded binary tree
2. Right sub tree
3. In-order successor
4. Subtracting the height of its right sub-tree from the height of the left sub-tree.
5. The left sub-tree of the tree is one level lower than that of the right sub-tree.
6. O(log n)
7. LL
8. Black and black
9. P (the parent of n) is the root of the splay tree.
10. Splay

Chapter 11

Multiple-choice Questions

1. (b) 2. (a) 3. (c) 4. (c) 5. (c)

True or False

1. True 2. True 3. False 4. False 5. False 6. True 7. True
8. False 9. False

Fill in the blanks

1. M and M-1
2. m and m-1
3. m/2
4. B tree
5. O(log N)

Chapter 12

Multiple-choice Questions

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

True or False

1. True 2. False 3. True 4. True 5. False 6. False 7. True
8. False 9. True 10. False 11. False

Fill in the blanks

1. $2i + 1$
2. Priority queues
3. Partially ordered trees
4. Min heap
5. Root
6. i
7. A set of binomial trees that satisfy binomial-heap properties
8. 2^i
9. O(1)
10. Collection of heap-ordered trees
11. Whether node *x* has lost a child since the last time *x* was made the child of another node

Chapter 13

Multiple-choice Questions

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

True or False

1. False 2. False 3. True 4. False 5. True 6. False 7. True
8. True 9. True 10. True

Fill in the Blanks

1. An isolated node
2. Terminate
3. Bit matrix or Boolean matrix
4. Cycle
5. Multi-graph
6. Tree vertices
7. Transitive closure
8. Articulation point
9. bi-connected
10. Bridge

Chapter 14

Multiple-choice Questions

1. (b) 2. (d) 3. (c) 4. (d) 5. (b) 6. (a) 7. (d)
8. (a) 9. (c) 10. (d) 11. (d)

True or False

1. False 2. False 3. False 4. False 5. True 6. False 7. True
8. False 9. True 10. True 11. True

Fill in the Blanks

1. Sorted array
2. O(n)
3. The process of arranging values in a predetermined order.
4. Merge sort / heap sort/ quick sort
5. External sorting
6. Bubble sort
7. $O(n^2)$
8. Merge sort/ quick sort
9. O(n log n)
10. O(n.k)
11. O(n log n)
12. Pivot element

Chapter 15

Multiple-choice Questions

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

True or False

1. False 2. False 3. False 4. False 5. False 6. True 7. False
8. True 9. True

Fill in the Blanks

1. Hash function
2. Hashing
3. Sentinel value and a data value
4. Collision
5. Probes
6. Quadratic probing

Chapter 16

Multiple-choice Questions

1. (d) 2. (c) 3. (a) 4. (a)

True or False

1. False 2. True 3. True 4. True 5. True 6. False 7. True
8. True

Fill in the Blanks

1. File
2. Type and size
3. Record
4. File position
5. File structure
6. Record number and address
7. Primary
8. Inverted
9. Database

INDEX

B

I

P

Q

T