SUMMARY

- All data and programs need to be placed in the primary memory for execution.
- Pointers are nothing but memory addresses. A pointer is a variable that contains the memory address of another variable.
- The '**&**' operator retrieves the lvalue (address) of the variable. We can dereference a pointer, i.e., refer to the value of the variable to which it points by using unary '*' operator.
- The address of a memory location is a pointer constant, therefore it cannot be changed in the program code.
- Unary increment and decrement operators have greater precedence than the dereference operator (*).
- Null pointer is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address. To declare a null pointer you may use the predefined constant NULL, You may also initialize a pointer as a null pointer by using a constant 0.
- A generic pointer is a pointer variable that has **void** as its data type. The generic pointer can be used to point to variables of any data type.
- When the memory for an array is allocated, its base address is fixed and it cannot be changed during program execution.
- When we dynamically allocate memory for variables, heap acts as a memory pool from which memory is allocated to those variables. The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation.
- `malloc()`, `calloc()`, and `realloc` returns a void pointer if successful, else a `NULL` pointer is returned.
- Memory leakage occurs when memory is allocated but not released when it is no longer required.

GLOSSARY

Alias A reference (usually in the form of a pointer) to an object which is also known via other references that may include its own name or other pointers.

Dereference To look up a value referred to. Usually, the 'value referred to' is the value pointed to by a pointer. Therefore, 'dereference a pointer' means to see what it points to. In C, a pointer is dereferenced either using the unary * operator or the array subscripting operator [].

Function pointer A pointer to any function type.

Lvalue An expression that appears on the left-hand sign of an assignment operator, hence, something that can perhaps be assigned to. An lvalue specifies something that has a location, as opposed to a transient value.

Null pointer A pointer value which is not the address of any object or function. A null pointer points to nothing.

Null pointer constant An integral constant expression with value 0 (or such an expression cast to void *), that represents a null pointer.

Pointer Variable that stores addresses

Rvalue An expression that appears on the right-hand sign of an assignment operator. Generally, rvalue can participate in an expression or be assigned to some other variable.

EXERCISES

Fill in the Blanks

1. Size of character pointer is _____.
2. Allocating memory at run time is known as _____.
3. A pointer to a pointer stores _____ of another _____ variable.
4. _____ pointer does not point to any valid memory address.
5. The size of memory allocated for a variable depends on its _____.
6. On 16-bit systems, integer variable is allocated _____ bytes.
7. The _____ appears on the right side of the assignment statement.
8. Pointers are nothing but _____.
9. _____ enable programmers to return multiple values from a function via function arguments.
10. The _____ operator informs the compiler that the variable is a pointer variable.
11. Data and programs need to be placed in the _____ for execution.
12. When compared with heaps, _____ is faster but also smaller and expensive.
13. All variables declared within main() are allocated space on the _____.
14. Shared libraries segment contains _____.
15. The function malloc() returns _____.
16. When memory is de-allocated using the free(), it is returned back to the _____ within the _____.
17. The functions malloc(), calloc(), and realloc return _____ if successful, else _____ is returned.
18. The function realloc() is used to _____.
19. A two-dimensional array is a _____.
20. An un-initialized memory contains _____.
21. Pointer to pointer stores _____.
22. The only integer value that can be assigned to a pointer variable is _____.
23. _____ can be used as parameter declaration to declare an array of integers passed to a function.
24. Dynamically allocated memory can be referred using _____.
25. _____ function returns memory to the heap.
26. Ragged array is implemented using an array of pointers to _____.

Multiple Choice Questions

1. The operator signifies a
   (a) referencing operator
   (b) dereferencing operator
   (c) address operator
   (d) none of these

2. (&num) is equivalent to writing
   (a) &num (b) *num
   (c) num (d) none of these
3. Pointers are used to create complex data structures like.
   (a) trees (b) linked list
   (c) stack (d) queue
   (e) all of these
4. While declaring pointer variables, which operator do we use?
   (a) address (b) arrow
   (c) indirection (d) dot
5. Which operator retrieves the lvalue of a variable?
   (a) & (b) *
   (c) -> (d) none of these
6. The code of the main() program is stored in memory in
   (a) stack (b) heap
   (c) global (d) bss
7. For dynamically allocated variables, memory is allocated from which memory area?
   (a) Stack (b) Heap
   (c) Global (d) BSS
8. The function `malloc()` is declared in `which header file`
   (a) stdio.h (b) stdlib.h
   (c) conio.h (d) iostream.h
9. Which function is used to request memory and set all aloocated bytes to zero?
   (a) malloc() (b) calloc()
   (c) realloc() (d) free()

State True or False

1. A pointer is a variable
2. The **&** operator retrieves the lvalue of the variable.
3. Array name can be used as a pointer.
4. Unary increment and decrement operators have greater precedence than the dereference operator.
5. The generic pointer can be pointed at variables of any data type.
6. A function pointer cannot be passed as a function's calling argument.
7. On 32-bit systems, integer variable is allocated 4 bytes.
8. Lvalue cannot be used on the left side of the assignment statement.
9. Pointers provide an alternate way to access individual elements of the array.
10. Pointer is a variable that represents the contents of a data item.
11. Unary increment and decrement operators have greater precedence than the dereference operator.
12. A fixed size of stack is allocated by the system and is filled as needed from the top to bottom.
13. All the parametres passed to the called function will be stored on the stack.
14. When the memory for an array is allocated, its base address is fixed and it cannot be changed during program execution.
15. An array can be assigned to another array.
16. Memory leakage occurs when memory is allocated but not released when it is no longer required.
17. mat[i][j] is equivalent to *(*(mat + i) + j).
18. Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer.
19. It is possible to add two pointer variables.
20. Pointer constants cannot be changed.
21. The value of a pointer is always an address.
22. *ptr++ will add 1 to the value pointed by ptr.
23. Pointers of different types can be assigned to each other without a cast.
24. Adding 1 to a pointer variable will make it point 1 byte ahead of the memory location to which it is currently pointing.
25. Any arithmetic operator can be used to modify the value of a pointer.
26. Only one call to free() is enough to release the entire array allocated using calloc().
27. Ragged arrays consumes less memory space.

Review Questions

1. Explain the difference between a null pointer and a void pointer.

EXERCISES

2. Define pointers.
3. Write a short note on pointers.
4. Compare pointer and array name.
5. Explain the result of the following code-

```
int num1 = 2, num2 = 3;
int *p = &num1, *q = &num2;
*p++ = *q++;
```

6. What do you understand by a null pointer?
7. What is an array of pointers? How is it different from a pointer to an array?
8. Write a short note on pointer arithmetic.
9. How are generic pointers different from other pointer variables?
10. What do you understand by the term pointer to a function?
11. Differentiate beween ptr++ and *ptr++.
12. How are arrays related to pointers?
13. Briefly explain array of pointers.
14. Write a program to sort 10 integers using array of pointers.
15. Write a program to illustrate the use of a pointer that points to a 2D array.
16. Give the advantages of using pointers.
17. Can we have an array of function pointers? If yes, illustrate with the help of a suitable example.
18. Explain the term dynamic memory allocation.
19. Differentiate between malloc(), calloc and realloc().
20. Write a short note on pointers to pointers.
21. Differentiate between a function returning pointer to int and a pointer to function returning int.
22. Write a program that illustrates passing of character arrays as an argument to a function (use pointers).
23. Differentiate between pointer to constants and constant to pointers.
24. What is a void pointer?
25. Define null pointer.
26. Explain the call by address technique of passing parameters to function.
27. How are pointers used on two dimensional arrays?
28. Write a program to print Hello world using pointers.
29. Write a program to enter a lowercase character. Print this character in uppercase and also display its ASCII value.
30. Write a program to subtract two integer values.
31. Write a program to calculate area of a circle.
32. Write a program to convert 3.14 into its integral equivalent.
33. Write a program to find smallest of three integer values.
34. Write a program to input a character and categorize it as a vowel or a consonant.
35. Write a program to input 10 values in an array. Categorize each value as positive, negative, or equal to zero.
36. Write a program to input a character. If it is in uppercase print in lowercase and vice versa.
37. Write a program to display the sum and average of numbers from 100–200.
38. Write a program to print all odd numbers from 100–200.
39. Write a program to input 10 values in an array. Categorise each value as prime or composite.
40. Write a program to subtract two floating point numbers using functions.
41. Write a program to calculate the area of a circle.
42. How can you declare a pointer variable?
43. Differentiate between a variable address and a variable's value. How can we access a variable's address and its value using pointers?
44. Give a brief of different memory areas available to the programmer.
45. What do you understand by dereferencing a pointer variable?
46. Write a short note on pointer expressions and pointer arithmetic.
47. What will *p++ = *q++ do?
48. Write a program to add two integers using functions. Use call by address technique of passing parameters.
49. Write a short note on pointer and a three dimensional array.
50. How can a pointer be used to access individual elements of an array? Illustrate with an example.

51. Can we assign a pointer variable to another pointer variable? Justify your answer with the help of an example.
52. What will happen if we add or subtract an integer to or from a pointer variable?
53. Is it possible to compare two pointer variables? Illustrate using an example.
54. Can we subtract two pointer variables?
55. With the help of an example explain how an array can be passed to a function? Is it possible to send just a single element of the array to a function?
56. Can array names appear on the left side of the assignment operator? Why?
57. Differentiate between an array name and an array pointer.
58. Using a program, explain how pointer variables can be used to access strings.
59. Write a program to print "Good Morning" using pointers.
60. Write a program to print the lowercase characters into uppercase and vice versa in the given string "gOOd mORninG".
61. Write a program to copy "University" from the given string "Oxford University Press" in another string.
62. Write a program to copy last five characters from the given string "Oxford University Press" in another string.
63. Write a menu-driven program to perform various string operations using pointers.
64. How can you have array of function pointers? Illustrate with an example.
65. Briefly discuss memory allocation schemes in C language.
66. Write a program to read and print a floating point array. The space for the array should be allocated at the run time.
67. Write a program to demonstrate working of calloc().
68. Write a short note on realloc(). Give a program to explain its usage.
69. With the help of an example, explain how pointers can be used to dynamically allocate space for two-dimensional and three-dimensional arrays.
70. Write a short note on wild pointers.
71. Give a briefing of memory leakage problem with the help of an example.
72. What is a dangling pointer?
73. Explain memory corruption with the help of suitable examples.
74. Differentiate between *(arr+i) and (arr+i).
75. Write a function to calculate roots of a quadartic equation. The function must accept arguments and return result using pointers.
76. Write a program using pointers to insert a value in an array.
77. Write a program using pointers to search a value from an array.
78. Write a function that accepts a string using pointers. In the function, delete all the occurrences of a given character and display the modified string on the screen.
79. Write a program to reverse a string using pointers.
80. Write a program to compare two arrays using pointers.
81. How can we access the value pointed by a pointer to a pointer?

Program output

Give the output of the following code.

1.
```
main()
{
   int arr[]={1,2,3,4,5};
   int *ptr, i;
   ptr = arr+4;
   for(i = 4; i >= 0;i--)
      printf("\n %d", *(ptr-i));
}
```

2.
```
main()
{
   int arr[]={1,2,3,4,5};
   int *ptr, i;
   ptr = arr+4;
   for(i = 0; i < 5; i++)
      printf("\n %d", *(ptr-i));
}
```

3.
```
#include <stdio.h>
main()
{
   int val=3;
   int *pval=&val;
   printf("%d %d", ++val, *ptr);
}
```

4.
```
#include <stdio.h>
main()
{
   int val=3;
   int *pval=&val;
   printf("%d %d", val, *ptr++);
}
```

5.
```
#include <stdio.h>
main()
{
   int val=3;
   int *pval=&val;
   printf("%d %d", val, ++*ptr);
}
```

6.
```
#include <stdio.h>
main()
{
   int arr[]={1,2,3,4,5};
   printf("%d", ++*arr);
}
```

7.
```
#include <stdio.h>
main()
{
   int arr[]={1,2,3,4,5};
   int *parr - arr+2;
   printf("%d %d", ++*parr-1, 1+*-parr);
}
```

8.
```
#include <stdio.h>
main()
{
   int num = 5, *ptr=&a, x=*ptr;
   printf("%d %d %d", ++num, x+2,
   *ptr--);
}
```

9.
```
#include <stdio.h>
main()
{
   int num=5, *ptr=&num;
   printf("\n %d", *&num);
   printf("\n %d", *&*&num);
   printf("\n %d", *&ptr);
   printf("\n %d", **&ptr);
   printf("\n %d", &**&ptr);
}
```

10.
```
main()
{
   int num=5, *ptr=&num;
   printf("%d %d", num, x+2, (*ptr)--);
}
```

11.
```
#include <stdio.h>
main()
{
   int arr[]={1,2,3,4,5}, i, k = 3;
   for(i=0;i<10;i++)
      *(arr+i)=i;
   printf("%d", *(arr[+k-1]);
}
```

12.
```
#include <stdio.h>
main()
{
   int arr[]={1,2,3,4,5};
   int i=1,j=2;
   printf("\n %d", *(arr+1+i));
   printf("\n %d", *(arr+*(arr+1)));
   printf("\n %d", *(arr+i)+*(arr+j);
   printf("\n %d", *(arr+j));
}
```

13.
```
main()
{
   char str[]="ABCDEFGH", *pstr=str;
   pstr++;
   while(*pstr!='H')
      printf("%c", *pstr++);
}
```

14.
```
main()
{
   char str[]="ABCDEFGH";
   printf("%d", (&str[3]-&str[0]);
}
```

15.
```
main()
{
   char *str="ABCDEFGH";
   (*str++);
   printf("%s", str);
}
```

16.
```
main()
{
   char *str="ABCDEFGH";
   str++;
   printf("%s", str);
}
```

17.
```
main()
{
   char *str="AbcDefGh";
   int i=0;
   while(*str)
   {
      if(isupper(*str++))
         i++;
   }
   printf("%d", i);
}
```

18.
```
main()
{
   printf("Hello World"+3);
}
```

19.
```
main()
{
   int arr[][2]={1,2,3,4,5,6,7,8,9};
   printf("%d", sizeof(arr));
}
```

20.
```
main()
{
   int arr[2][3]={1,2,3,4,5,6,7,8,9};
   printf("%d", sizeof(arr[1]));
}
```

21.
```
main()
{
   int arr[5], *parr=arr;
   while(parr < &arr[5])
   {
      *parr = parr-arr;
      printf("%d", *parr);
      parr++;
   }
}
```

22.
```
main()
{
   char *str = "Hello World";
   str[5]='!';
   printf("%s", str);
}
```

23.
```
main()
{
   char *str1 = "Hello World";
   char str2[20] = "Hello World";
   char str3[] = "Hello World";
   printf(" %d %d %d", sizeof(str1),
   sizeof(str2), sizeof(str3));
}
```

24.
```
main()
{
   register int num = 3, *ptr = &num';
   printf("%d",*ptr);
}
```

25.
```
#include <stdio.h>
void func(int (*parr)[3]);
main()
{
   int arr[2][3] = {1,2,3,4,5,6};
   func(arr);
   func(arr + 1);
}
void func(int (*parr)[3])
{
   int i;
   for(i = 0; i < 2; i++)
      printf("%d", (*parr)[i]);
}
```

Find errors if any in the following statements.

- `int ptr, *ptr;`
- `int num, *ptr=num;`
- `int *ptr=10;`
- `int num, **ptr=&num;`
- `int *ptr1, *ptr2, *ptr3=*ptr1+*ptr2;`
- `int *ptr; scanf("%d", &ptr);`

ANNEXURE 4

A4.1 DECIPHERING POINTER DECLARATIONS

The *right–left* rule is a widely used rule for creating as well as deciphering C declarations. Before starting, let us first understand the meaning of different symbols and the way in which they are read.

Symbol	Read As	Location
*	pointer to	placed on the left side
[]	array of	placed on the right side
()	function returning	placed on the right side

Following are the steps to decipher the declaration:

Step 1: Find the identifier and read as 'identifier is'.

Step 2: Read the symbols on the right of the identifier. For example, if you find '( )', then you know that it is a function declaration. So you can now say, 'identifier is function returning'. Or if you encounter a '[ ]', then read it as 'identifier is array of'.

Continue moving right until you either run out of symbols or you encounter a right parenthesis.

Step 3: Now, check the symbols to the left of the identifier. If it is not a symbol given in the table, then just say it. For example if you encounter int, then just say it as it is. Otherwise, translate it into English using the above table. Continue going left until you either run out of symbols or you encounter a left parenthesis.

Step 4: Repeat Steps 2 and 3 until the entire declaration is completely deciphered.

Consider some examples:

```
int *ptr[];
```

Step 1: Find the identifier. Here, the identifier is `ptr`. So say,

```
'ptr is'
```

Step 2: Identify the symbols on the right side of the identifier until you run out of symbols or encounter a left parenthesis. Here, the symbol is []. So say

```
'ptr is array of'
```

Step 3: Move to left of the identifier until you run out of symbols or encounter a left parenthesis. Here, the symbol is *. So say,

```
ptr is array of pointer to
```

Step 4: Continue moving left. Here, you find int. So say,

```
ptr is array of pointer to int.
```

Now decipher the following declaration

```
int *(*func())();
```

Step 1: Find the identifier. Here, the identifier is func. So say,

```
func is
```

Step 2: Identify the symbols on the right side of the identifier until you run out of symbols or encounter a left parenthesis. Here, the symbol is (). So say

```
func is function returning
```

Step 3: Move to left of the identifier until you run out of symbols or encounter a left parenthesis. Here, the symbol is *. So say,

```
func is function returning pointer to
```

Step 4: Can't move left anymore because of the left parenthesis, so now move to right. Here, you will encounter a symbol (). So say,

```
func is function returning pointer to
   function returning
```

Step 5: Can't move right anymore as all symbols have exhausted so move to left. Here you will encounter a '*'. So say,

```
func is function returning pointer to
   function returning pointer to
```

Step 6: Continue moving left. Here you will find int. So say,

```
func is function returning pointer to
   function returning pointer to int.
```

Some declarations also contain array size and function parametres. So if you encounter a symbol as '[10]', then read it as 'array (size 10)'. If you encounter, something like '(int *, char)', then read it as 'function expecting (int *, char) and returning...'. Now consider such an example and decipher the following decalaration:

```
int (*(*func)(int *, char))[3][5];
```

Use the steps illustrated below and check your answer.

```
func is pointer to function expecting (int
   *,char) and
returning pointer to array (size 3) of array
   (size 5) of int.
```

Some Illegal Declarations in C

It is quite possible that you end up with some illegal declarations using this rule. So, you must be very clear about what is legal in C and what is not allowed in the language. For example, consider a declaration as shown:

```
int *((*func)())[][];
```

This can be deciphered as, 'func is pointer to function returning array of array of pointer to int'. But did you notice that a function cannot return an array, but only a pointer to an array. Therefore, this declaration is illegal.

Let us look as some more illegal combinations in C language:

```
[]()–C does not permit an array of functions

()()–A function cannot return a function

()[]–A function cannot return an array
```

The table given below lists the declaration, meaning, and its remarks as valid or invalid.

Declaration	Meaning	Remarks
float num;	num is a float	Valid
char *ch;	ch is a pointer to char	valid
int arr[];	arr is an array of int	Valid
float func();	func is a function returning float	Valid
int **ptr;	ptr is a pointer to pointer to int	Valid
char (*ptr)[];	ptr is a pointer to array of char	valid
char (*ptr)();	ptr is a pointer to function returning char	valid
float *ptr[];	ptr is array of pointers to float	Valid
int mat[][];	mat is an array of array of int	Valid
int func[]()	func is an array of function returning int	Invalid
float *func();	func is a function returning pointer to float	Valid
float func()[];	func is a function returning array of float	Invalid
float func()();	func is a function returning function returning float	Invalid
int ***ptr;	ptr is a pointer to pointer to pointer to int	Valid
int (**ptr)[];	ptr is a pointer to pointer to array of int	Valid
float (**func)();	func is a pointer to pointer to a function returning float	Valid
char *(*ptr)[];	ptr is a pointer to array of pointer to char	Valid
float (*ptr)[][];	ptr is a pointer to array of array of float	Valid
float (*ptr)[]();	ptr is a pointer to array of function returning float	Invalid
char *(*(ptr)();	ptr is a pointer to function returning a pointer to char	Valid
char (*ptr)()[];	ptr is a pointer to function returning an array of char	Invalid
int (*ptr)()();	ptr is a pointer to function returning function returning int	Invalid
float **ptr[];	ptr is an array of pointer to pointer to int	Valid

(Contd)

(Contd)

Declaration	Meaning	Remarks
char (*ptr[])[];	ptr is array of pointer to array of char	Valid
char (*ptr[])();	ptr is array of pointer to function returning char	Valid
float *ptr[][];	ptr is array of array of pointer to float	Valid
int arr[][][];	arr is array of array of array of int	Valid
int arr[][]();	arr is array of array of function returning int	Invalid
float *arr[]();	arr is array of function returining pointer to float	Invalid
int arr[]()[];	arr is array of function returning array of int	Invalid
float **func();	func is a function returning pointer to float pointer (or pointer to float)	Valid
char *func()[];	func is a function returning array of char pointer	Invalid
float (*func())[];	func is a function returning pointer to array of float	Valid
float (*func())();	func is a function returning pointer to function returning float	Valid
char func()[][];	func is a function returning array of array of char	Invalid
char func()[]();	func is a function returning array of array of function returning char	Invalid
char *func()();	func is a function returning function returning char pointer	Invalid