```
        cin>>*(parr+i);
    for(i=0; i < n; i++)
        cout<<"\n arr["<<i<<"] = "<<*(parr+i);


}
```

OUTPUT

```
Enter the number of elements : 3
Enter the elements : 1 2 3
arr[0] = 1
arr[1] = 2
arr[2] = 3
```

Program 8.7 Write a program to find mean of n numbers using arrays.

```
using namespace std;
#include<iostream>
int main()
{   int i, n, arr[20], sum =0;
    int *pn = &n, *parr = arr, *psum = &sum;
    float mean = 0.0, *pmean = &mean;
    cout<<"\n Enter the number of elements in the array : ";
    cin>>*pn;
    cout<<"\n Enter the elements : ";
    for(i=0; i < n; i++)
        cin>>*(parr+i);
    for(i=0; i < *pn; i++)
        *psum += *(arr + i);
    *pmean = *psum / *pn;
    cout<<"\n The sum of these numbers is : "<<*psum;
    cout<<"\n The mean of these numbers is : "<<*pmean;
}
```

OUTPUT

```
Enter the number of elements : 5
1 2 3 4 5
The sum of these numbers is : 15
The mean of these numbers is : 3
```

Difference between reference variable and pointer variable

When we call a function using call-by-address, we are actually creating pointer variables. Though reference and pointer variables may sound similar but they are actually different in many ways, such as,

1. A reference variable means the same variable just with a different name. While a reference variable must refer to some variable (or object) and can never be NULL, a pointer variable on the other hand, may be NULL.
2. A pointer variable can be re-assigned while reference cannot.
3. A pointer variable can be assigned anywhere in the code but a reference variable must be assigned at initialization only.
4. A pointer variable can be assigned NULL directly, whereas reference variable cannot be.
5. Pointer variables can be made to iterate over an array by using the ++ operator (to go to the next item that a pointer is pointing to) but you cannot use a reference variable to do the same.
6. A pointer variable holds a memory address. But, a reference variable has the same memory address as the variable it references.
7. A pointer variable is dereferenced with * to use the value stored at that memory location, but a reference variable can be used directly.

While calling a function, prefer to use call-by-reference. Moreover, if you want to write a code that will work in C as well as C++, then you have no option but to use the call-by-address technique of passing parameters.

from one function to another. Hence, to use pointers for passing arguments to a function, the programmer must do the following:

- Declare the function parameters as pointers.
- Use the dereferenced pointers in the function body.
- Pass the addresses as the actual argument when the function is called.

Note It is an error to return a pointer to a local variable in the called function, because when the function terminates, its memory may be allocated to a different program.

Let us write some programs that pass pointer variables as parameters to functions.

Program 8.4 Write a program to add two integers using functions.

```
using namespace std;
#include<iostream>
void sum( int *a, int *b, int *t);
int main()
{       int num1, num2, total;
        cout<<"\n Enter the two numbers : ";
        cin>>num1>>num2;
        sum(&num1, &num2, &total);
        cout<<"\n Total = "<<total;
}
void sum ( int *a, int *b, int *t)
{       *t = *a + *b;
}
```

OUTPUT

```
Enter the two numbers : 2  3
Total = 5
```

Program 8.5 Write a program to calculate the area of a triangle.

```
using namespace std;
#include<iostream>
void read(float *b, float *h);
void calculate_area(float *b, float *h, float *a);
int main()
{       float base, height, area;
        read(&base, &height);
        calculate_area(&base, &height, &area);
        cout<<"\n Area of the triangle with base "<<base<<" and height
"<<height<<" = "<<area;
}
void read(float *b, float *h)
{       cout<<"\n Enter the base and height of the triangle : ";
        cin>>*b>>*h;
}
void calculate_area( float *b, float *h, float *a)
{       *a = 0.5 * (*b) * (*h);
}
```

OUTPUT

```
Enter the base and height of the triangle : 10  5
Area of the triangle with base 10.0 and height 5.0 = 25.00
```

Example 8.10 To display an array of given numbers from 1 to 9

```
using namespace std;
#include<iostream>
main()
{       int arr[]={1,2,3,4,5,6,7,8,9};
        int *ptr1, *ptr2;
        ptr1 = arr; // ptr1 pointing to the first element of the array
        ptr2 = &arr[8]; // ptr2 pointing to the last element of the array
        while(ptr1<=ptr2)
        {       cout<<"  "<<*ptr1;
                ptr1++;
        }
}
```

OUTPUT

```
1 2 3 4 5 6 7 8 9
```

Note An object is a named region of storage; an `lvalue` is an expression referring to an object

Table 8.1 Pointer arithmetic

Operation	Condition	Declaration	Description
Assignment	**Pointers must be of the same type.**	`int *ptr1, *ptr2;`	**If we write,** `*ptr1=*ptr2`, then both the pointers point to the same location.
Addition of an integer	..	`int i, *ptr;`	If we write, `ptr+i`, then `ptr` will point to ith object after its initial position.
Subtraction of an integer	..	`int i, *ptr;`	If we write, `ptr-i`, then `ptr` will point to ith object before its initial position.
Comparison of pointers	Pointers should point to elements of the same array.	`int *ptr1, *ptr2;`	If we write, `ptr1<ptr2`, then it would return 1 if ptr1 points to an element that is near to the front of the array.
Subtraction of pointers	Pointers should point to elements of the same array.	`int *ptr1, *ptr2;`	If we write `ptr1-ptr2`, then it returns the number of elements between `ptr1` and `ptr2` (provided `ptr1>ptr2`)
NOTE: Addition of pointers is illegal. Therefore, `ptr1+ptr2` is not valid. Moreover, multiplication and division are also not allowed. Therefore, `*ptr1 /= int_a;` or `*ptr1 /= *ptr2; *ptr1 *= int_a;` or `*ptr1 *= *ptr2;` are illegal statements.			

Program 8.6 Write a program to read and display an array of n integers.

```
using namespace std;
#include<iostream>
int main()
{       int i, n;
        int arr[10], *parr = arr;
        cout<<"\n Enter the number of elements : ";
        cin>>n;
        cout<<"\n Enter the elements : ";
        for(i=0; i < n; i++)
```

8.8 PASSING ARRAY TO FUNCTIONS

An array can be passed to a function using pointers. For this, a function that expects an array can declare the formal parameter in either of the two ways.

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

When we pass the name of the array to a function, the address of the zero element of the array is copied to the local pointer variable in the function. Observe the difference—unlike ordinary variables, the values of the elements are not copied; only the address of the first element is copied.

When a formal parameter is declared in a function header as an array, it is interpreted as a pointer a variable and not an array. With this pointer variable you can access all the elements of the array by using the expression, `array name + index`. Note to know how many elements are there in the array, you must pass the size of the array as another parameter to the function. So for a function that accepts an array as parameter, the declaration should be as follows.

```
func(int arr[], int n);        OR func(int *arr, int n);
```

Look at the following program which illustrates the use of pointers to pass array to a function.

Program 8.8 Write a program to read and print an array of n numbers, find out the smallest number, and print the position of the smallest number.

```
using namespace std;
#include<iostream>
void read_array(int *arr, int n);
void find_small(int *arr, int n, int *small, int *pos);
int main()
{       int num[10], n, smallest, pos;
        cout<<"\n Enter the size of the array : ";
        cin>>n;
        cout<<"\n Enter the elements of the array : ";
        read_array(num, n);
        find_small(num, n, &smallest, &pos);
        cout<<"\n The smallest number in the array is = "<<smallest<<" at position
"<<pos;
}
void read_array( int *arr, int n)
{       for(int i=0;i<n;i++)
                cin>>*(arr+i);
}
void find_small(int *arr, int n, int *small, int *pos)
{       *small = arr[0], *pos=0;
        for(int i=0;i<n;i++)
        {       if( *(arr+i)< *small)
                {       *small = *(arr+i);
                        *pos = i;
                }
        }
}
```

OUTPUT

```
Enter the size of the array : 5
Enter the elements of the array : 1 2 3 4 5
The smallest number in the array is = 1 at position 0
```

Note that it is not necessary that we have to pass the whole array to a function. We can also pass a part of the array also known as a sub-array. A pointer to a sub-array is also an array pointer. For example, if we want to send the array starting from the third element, then we can pass the address of the third element and the size

Simila... we are passing the same address, because we have written pstr = st...

Example 8.13 To read a string and then scan each character to count the number of upper and lower ca... characters entered

```
using namespace std;
#include<iostream>
int main()
{       char str[100], *pstr=str; // pstr is a pointer to string
        int upper = 0, lower = 0;
        cout<<"\n Enter the string : ";
        cin.getline(str,100);
        while(*pstr != '\0')
        // while loop will be executed until the null character is encountered
        {       if(*pstr >= 'A' && *pstr <= 'Z')
                        upper++;
                else if(*pstr >= 'a' && *pstr <= 'z')
                        lower++;
                pstr++;
        }
        cout<<"\n Total number of upper case characters = "<<upper;
        cout<<"\n Total number of lower case characters = "<<lower;
}
```

OUTPUT

```
Enter the string : Oxford University Press
Total number of upper case characters = 3
Total number of lower case characters = 18
```

Program 8.9 Write a program to read and print a text and count the number of characters, words, and lines in the text.

```
using namespace std;
#include<iostream>
int main()
{       char str[100], *pstr=str;
        int chars = 1, lines=1, words=1;
        cout<<"\n Enter the string : ";
        cin.getline(str,100);
        while(*pstr != '\0')
        {       if(*pstr == '\n')
                        lines++;
                if(*pstr == ' ' && *(pstr + 1) != ' ')
                        words++;
                chars++;
                pstr++;
        }
        cout<<"\n Number of characters = "<< chars;
        cout<<"\n Number of lines = "<< lines;
        cout<<"\n Number of words = "<< words;
}
```

OUTPUT

```
Enter the string : Oxford University Press
Number of characters = 24
Number of lines = 1
Number of words = 3
```

Program 8.10 Write a program to copy n characters of a character array from the mth position in another character array.

```
using namespace std;
#include<iostream>
#include<stdio.h>
int main()
{       char str[100], copy_str[100];
        char *pstr=str, *pcopy_str = copy_str;
        int m, n, i=0;
        cout<<"\n Enter the string : ";
        cin.getline(str,100);
        cout<<"\n Enter the position from which to start : ";
        cin>>m;
        cout<<"\n Enter the number of characters to be copied : ";
        cin>>n;
        pstr = pstr + m - 1;
        i=0;
        while(*pstr != '\0' && i <n)
        {       *pcopy_str = *pstr;
                pcopy_str++;
                pstr++;
                i++;
        }
        *pcopy_str = '\0';
        cout<<"\n The copied text is : ";
        puts(copy_str);
}
```

OUTPUT

```
Enter the string : Oxford University Press
Enter the position from which to start : 5
Enter the number of characters to be copied : 5
The copied text is : d Uni
```

8.11 ARRAY OF POINTERS

An array of pointers can be declared as

```
int *ptr[10];
```

This statement declares an array of 10 pointers where each pointer points to an integer variable. For example, look at the code given here.

```
int *ptr[10];
int p=1, q=2, r=3, s=4, t=5;
ptr[0]=&p;
ptr[1]=&q;
ptr[2]=&r;
ptr[3]=&s;
ptr[4]=&t
```

```
int(*ptr)[10];
```

Here, ptr is a pointer to an array of 10 elements. The parentheses are not optional. In the absence of these parentheses, ptr becomes an array of 10 pointers, not a pointer to an array of 10 ints.

Note Pointer to a one-dimensional array can be declared as follows:

```
int arr[]={1,2,3,4,5};
int *parr;
parr=arr;
```

Similarly, pointer to a two- dimensional array can be declared as,

```
int arr[2][2]={{1,2},{3,4}};
int (*parr)[2];
parr=arr;
```

Example 8.16 To illustrate the use of a pointer to a 2D array

```
using namespace std;
#include<iostream>
main()
{       int arr[2][2]={{1,2},{3,4}};
        int i, (*parr)[2];
        parr=arr;
        for(i=0;i<2;i++)
        {       for(int j=0;j<2;j++)
                        cout<<"  "<<(*(parr+i))[j];
        }
}
```

OUTPUT

```
1 2 3 4
```

The golden rule to access an element of a two-dimensional array can be given as

```
arr[i][j] = (*(arr+i))[j] = *((*arr+i))+j) = *(arr[i]+j)
```

Therefore,

```
arr[0][0] = *(arr)[0] = *((*arr)+0) = *(arr[0]+0)
arr[1][2] = (*(arr+1))[2] = *((*(arr+1))+2) = *(arr[1]+2)
```

Note

If we declare an array of pointer using

```
data_type *array_ name[SIZE];
```

Here SIZE represents the number of rows and the space for columns that can be dynamically allocated.

If we declare a pointer to an array using,

```
data_type (*array_ name)[SIZE];
```

Here SIZE represents the number of columns and the space for rows that may be dynamically allocated

Program 8.11 Write a program to read and display a 3 × 3 matrix.

```
using namespace std;
#include<iostream>
int main()
{       int i, j, mat[3][3];
        cout<<"\n Enter the elements of the matrix ";
        for(i=0;i<3;i++)
        {       for(j=0;j<3;j++)
                        cin>>mat[i][j];

        }
        cout<<"\n The elements of the matrix are ";
        for(i=0;i<3;i++)
        {       cout<<"\n";
                for(j=0;j<3;j++)
                        cout<<"\t"<<*(*(mat + i)+j);
        }
}
```

OR

```
using namespace std;
#include<iostream>
int main()
{
int i, j, mat[3][3];
cout<<"\n Enter elements of the matrix";
for(i=0;i<3;i++)
{ for(j=0;j<3;j++)
cout<<"\n mat["<<i<<"]["<<j<<"] = ";
cin>>((*(mat + i)+j));
}
}
cout<<"\n The elements of the matrix are ";
for(i=0;i<3;i++)
{ cout<<"\n";
for(j=0;j<3;j++)
cout<<"\t mat"<<i<<""<<j<<" ="<< *(*(mat + i)+j);
}
return 0;
}
```

OR

```
using namespace std;
#include<iostream>
void display(int (*)[3]);
int main()
{       int i, j, mat[3][3];
        cout<<"\n Enter the elements of the matrix ";
        for(i=0;i<3;i++)
        {       for(j=0;j<3;j++)
                        cin>>mat[i][j];
        }
       display(mat);
}
void display(int (*mat)[3])
{       int i, j;
        cout<<"\n The elements of the matrix are ";
```

```
        for(i=0;i<3;i++)
        {       cout<<"\n";
                for(j=0;j<3;j++)
                        cout<<"\t"<<*(*(mat + i)+j);

        }
}
```

OUTPUT

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix
1 2 3
4 5 6
7 8 9
```

8.13 POINTERS AND 3D ARRAYS

In this section, we will see how pointers can be used to access a three-dimensional array. We have seen that pointer to a one-dimensional array can be declared as

```
int arr[]={1,2,3,4,5};
int *parr;
parr = arr;
```

Similarly, pointer to a two-dimensional array can be declared as

```
int arr[2][2]={{1,2},{3,4}};
int (*parr)[2];
parr = arr;
```

A pointer to a three-dimensional array can be declared as

```
int arr[2][2][2]={1,2,3,4,5,6,7,8};
int (*parr)[2][2];
parr = arr;
```

We can access an element of a three-dimensional array by writing,

```
arr[i][j][k] = *(*(*(arr+i)+j)+k)
```

Let us look at the example given below.

Example 8.17 To illustrate the use of a pointer to a 3D array

```
using namespace std;
#include<iostream>
main()
{       int i,j,k;
        int arr[2][2][2];
        int (*parr)[2][2]=arr;
        cout<<"\n Enter the elements of a 2 x 2 x 2 array: ";
        for(i=0;i<2;i++)
        {       for(j=0;j<2;j++)
                {       for(k=0;k<2;k++)
```

```
                        cin>>arr[i][j][k];
        }
}
cout<<"\n The elements of the 2 x 2 x 2 array are: ";
for(i=0;i<2;i++)
{       for(j=0;j<2;j++)
        {       for(k=0;k<2;k++)
                        cout<< " " <<*(*(*(parr+i)+j)+k);
        }
}
return 0;
}
```

OUTPUT

```
Enter the elements of a 2 x 2 x 2 array: 1 2 3 4 5 6 7 8
The elements of the 2 x 2 x 2 array are: 1 2 3 4 5 6 7 8
```

Note In the cout statement, you could also have used *(*(*(arr+i)+j)+k) instead of *(*(*(parr+i)+j)+k).

8.14 POINTERS TO FUNCTIONS

C++ allows operations with pointers to functions. We have discussed earlier in this chapter that every function code along with all its variables is allocated some space in the memory. Therefore, every function has an address. Function pointers are pointer variables that point to the address of a function. Like other pointer variables, function pointers can be declared, assigned values, and are used to access the functions they point to.

This is a useful technique for passing a function as an argument to another function. In order to declare a pointer to a function, we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses `()` and an asterisk `(*)` is inserted before the name. The syntax of declaring a function pointer can be given as follows:

```
return_type(*function_pointer_name)(argument_list);
```

Look at the declaration here in which we declare a pointer to a function that returns an integer value and accepts two arguments—one of integer type and the other of `float` type.

```
int (*func)(int a, float b);
```

Considering the precedence rule, if you do not put the function name within parenthesis, you will end up in declaring a function returning a pointer:

/* function returning pointer to int */

```
int *func(int a, float b);
```

8.14.1 Initializing Function Pointer

As in case of other pointer variables, a function pointer must be initialized prior to use. If we have declared a pointer to the function, then that pointer can be assigned to the address of the right sort of function just by

8.15 ARRAY OF FUNCTION POINTERS

When an array of function pointers is made, the appropriate function is selected using an index. The code given below shows the way to define and use an array of function pointers in C++.

Step 1: Use typedef keyword so that 'fp' can be used as type

typedef int (*fp)(int, int);

Step 2: Define the array and initialize each element to NULL. This can be done in two ways
// with 10 pointers to functions which return an int and take two ints

1. fp funcArr[10] = {NULL};
2. int (*funcArr[10])(int, int) = {NULL};

Step 3: Assign the function's address—'Add' and 'Subtract' funcArr1[0] = funcArr2[1] = Add;

```
funcArr[0] = &Add;
funcArr[1] = &Subtract;
```

Step 4: Call the function using an index to address the function pointer

```
cout<<"\n"<<funcArr[1](2, 3); // short form
cout<<"\n"<<(*funcArr[0])(2, 3); // correct way
```

Program 8.12 Write a program to add, that uses an array of function pointers, subtract, multiply, or divide two given numbers.

```
using namespace std;
#include<iostream>
int sum(int a, int b);
int subtract(int a, int b);
int mul(int a, int b);
```

```
int div1(int a, int b);
int (*fp[4]) (int a, int b);
int main(void)
 {
   int result;
   int num1, num2, op;
   fp[0] = sum;  fp[1] = subtract; fp[2] = mul; fp[3] = div1;
   cout<<"\n Enter the numbers: ";
   cin>>num1>>num2;
   do{
   cout<<"\n 0: Add \n 1: Subtract \n 2: Multiply \n 3: Divide \n 4. EXIT\n";
   cout <<"\n\n Enter the operation: ";
   cin >> op;
   result = (*fp[op]) (num1, num2);
   cout<<"\n Result = "<<result;
   while(op!=4);
   return 0; }

   int sum(int a, int b)
{ return a + b; }
int subtract(int a, int b)
{ return a - b; }
int mul(int a, int b)
{ return a * b; }
int div1(int a, int b)
{ if(b) return a / b; else return 0; }
```

SAMPLE OUTPUT

```
Enter the numbers: 2 3
0: Add
1: Subtract
2: Multiply
3: Divide
4. EXIT
Enter the operation: 0
Result = 5
Enter the operation: 4
```

8.16 POINTERS TO POINTERS

In C++, you are also allowed to use pointers that point to pointers. The pointers, in turn, point to data (or even to other pointers). To declare pointers to pointers, just add an asterisk (*) for each level of reference.

For example, if we have a code as follows:

```
int x=10;
int *px;          //pointer to an integer
int **ppx;          // pointer to a pointer to an integer
px=&x;
ppx=&px;
```

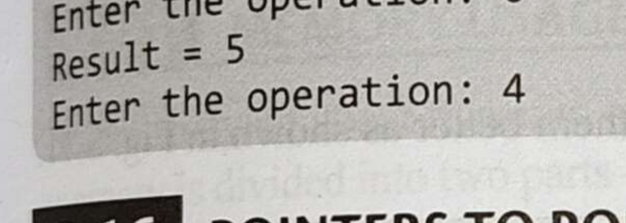

Figure 8.9 Pointer to pointer

Assume the memory location of these variables as shown in Fig. 8.9. Now if we write,

```
cout<<"\n "<< **ppx;
it would print 10, the value of x.
```

```
        cin>>arr[j++];
    cout<<"\n The array elements are : ";
    for(int i=0;i<10;i++)
        cout<<"   "<<arr[i];
    delete [] arr;

}
```

OUTPUT

```
Enter the array elements : 1  2  3  4  5
The array elements are : 1  2  3  4  5
LETS HAVE A BIGGER ARRAY ...
COPYING THE OLD ARRAY......
Enter new values : 6  7  8  9  10
The array elements are : 1  2  3  4  5  6  7  8  9  10
```

8.22.5 Advantages of `new/delete` Operators over `malloc()/free()`

While `malloc()` and `free()` are extensively used in C to allocate and release memory dynamically, they can be used in C++ as well but are generally not preferred due to the following reasons.

- The `new` operator calculates the size of the object. For this, it does not require the use of the `sizeof()` operator.
- `New` returns a pointer of relevant type. Therefore, it need not be typecasted as in case of `malloc()`.
- The `new` operator can initialize while allocation. This is not possible in case of `malloc()`.
- The syntax of `new` and `delete` is simple.
- `New` and `delete` operators can be overloaded (we will discuss this in the chapter on Operator Overloading)
- `malloc()` simply allocates memory and `free()` only releases it. However, new constructs objects and allocates memory as well. Similarly, `delete` not only deallocates memory but also destroys it.

Program 8.13 Write a program to add two vectors (arrays).

```
using namespace std;
#include<iostream>
#include <new>
#define MAX 5
void readVector(float *vector)
{     for(int i=0;i<MAX;i++)
            cin>>vector[i];

}
void addVectors(float *vec1, float *vec2, float *vec3)
{     for(int i=0;i<MAX;i++)
            vec3[i] = vec1[i] + vec2[i];

}
void displayVector(float *vector)
{     for(int i=0;i<MAX;i++)
            cout<<"   "<<vector[i];
}
main()
{     float *vec1 = new float [MAX];
```

```
delete[] pointer_variable;
```

for example, to release the space allocated to arr, you must write

```
delete[] arr;
arr = 0;      // reset the pointer to NULL
```

Note After deallocating the space, as a good programming practice, reset the pointer to null.

8.22.4 Alter the Size of Allocated Memory

At times, the memory allocated by using new might be insufficient or in excess. In both the situations, we can always change the memory size already allocated. This process is called reallocation of memory. In C++, memory can be reallocated in four simple steps.

- Create a new array of the appropriate type and of the new size.
- Copy the elements from the old array into the new array.
- Delete the old array.
- Rename the new array with the name of the initial array.

Example 8.22 To demonstrate the allocation, reallocation, and deallocation of memory

```
using namespace std;
#include<iostream>
#include<new>
#include<stdlib.h>
main()
{       int *arr = new int[5];
        if(arr == NULL)
        {       cout<<"\n Memory Allocation Failed";
                exit(1);
        }
        cout<<"\n Enter the array elements : ";
        for(int i=0;i<5;i++)
                cin>>arr[i];
        cout<<"\n The array elements are : ";
        for(int i=0;i<5;i++)
                cout<<"  "<<arr[i];
        cout<<"\n\n LETS HAVE A BIGGER ARRAY ...";
        int *temp = new int[10];
        if(temp == NULL)
        {       cout<<"\n\n\n Memory Allocation Failed";
                exit(1);
        }
        cout<<"\n\n\n COPYING THE OLD ARRAY..... ";
        for(int i=0;i<5;i++)
                temp[i] = arr[i];
        delete [] arr;
        arr = temp;
        cout<<"\n Enter new values : ";
        int j=5;
        for(int i=0;i<5;i++)
```

```
        cout<<"\n Enter the elements of the first vector : ";
        readVector(vec1);
        float *vec2 = new float [MAX];
        cout<<"\n Enter the elements of the second vector : ";
        readVector(vec2);
        float *vec3 = new float [MAX];
        addVectors(vec1, vec2, vec3);
        cout<<"\n The resultant vector is : ";
        displayVector(vec3);
        delete [] vec1;
        delete [] vec2;
        delete [] vec3;
}
```

OUTPUT

```
Enter the elements of the first vector : 1.1 2.2 3.3 4.4 5.5
Enter the elements of the second vector : 2.3 3.4 4.5 5.1 2.4
The resultant vector is : 3.4 5.6 7.8 9.5 7.9
```

Program 8.14 Write a program to sort an array.

```
using namespace std;
#include<iostream>
#include<new>
void sort(int *arr, int n)
{       for(int i=0;i<n;i++)
        {       for(int j=i+1;j<n;j++)
                {       if(arr[i]>arr[j])
                        {       int temp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = temp;
                        }
                }
        }
}
void display(int *arr, int n)
{       for(int i=0;i<n;i++)
                cout<<"  "<<arr[i];
}
void read(int *arr, int n)
{       for(int i=0;i<n;i++)
                cin>>arr[i];
}
main()
{       int n;
        cout<<"\n Enter the number of elements : ";
        cin>>n;
        int *arr = new int[n];
        if(arr!=NULL)
        {       read(arr, n);
                sort(arr, n);
        }
```

```
        cout<<"\n The sorted array is : ";
        display(arr,n);

}
```

OUTPUT

```
Enter the number of elements : 6
9 2 8 4 7
The sorted array is : 2 4 7 8 9
```

Program 8.15 Write a program to copy a given string into a new string. Memory for the new string must be allocated dynamically.

```
using namespace std;
#include<iostream>
#include<string.h>
void display(char *str)
{       cout<<"\t"<<str;        }
int length(char *str)
{       char *ptr = str;
        while(*ptr != '\0')
                ptr++;
        return ptr-str;

}
void copy_string(char *str1, char *str2)
{    while(*str1 != '\0')
      {                 *str2 = *str1;
                str2++, str1++;
      }
      *str2 = '\0';
}
main()
{       char *str = "Oxford";
        int len = length(str);
        char *str1 = new char [len+1];
        copy_string(str, str1);

cout<<"\n The copied String is : ";
        display(str1);
}
```

OUTPUT

```
The copied String is : Oxford
```

8.22.6 Dynamically Allocating 2D Arrays

We have seen how we can allocate a block of memory for an array. Now, we can extend our understanding further to extend the same mode to 2D arrays. A 2D array can be thought of as a collection of *n* number of 1D arrays, where each ID array represents a row. Figure 8.13 shows a 2D array `arr`. `arr` is a `pointer-to-pointer-to-int`: at the first level, 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).

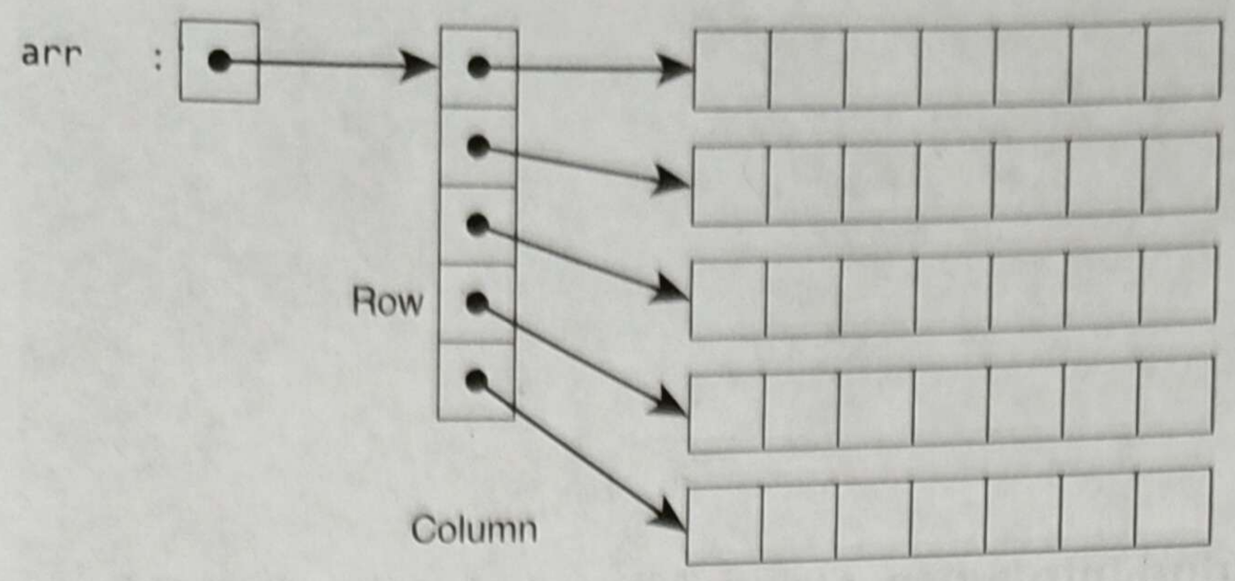

Figure 8.13 Two-dimensional array

A 2D array stored in memory in row major form can be allocated memory dynamically using the following steps:

- Define a pointer to pointer variable. For this, write `int **arr;`
- Allocate memory for storing pointers to rows. Therefore, write

```
arr = new int *[ROWS];
```

- Allocate memory for all columns. For this, write

```
for(int i=0;i<ROWS;i++)
        arr[i] = new int [COLS];
```

Once the memory is allocated for the 2D array, we can use subscripts to access its elements of `arr[i][j]`. 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);
```

Example 8.23 To illustrate the dynamic allocation of allocate space for a 2D array

```
using namespace std;
#include<iostream>
#include<stdlib.h>
int **AllocateMemory( int **arr, int ROWS, int COLS)
// dynamically allocates memory for the 2D array    .
{       arr = new int *[ROWS];
        if(arr == NULL) // to check if memory is allocated or not
        {               cout<<"\n Memory could not be allocated";
                exit(-1);
        }
        for(int i=0;i<ROWS;i++) // dynamically allocates memory for each row's
                                columns
                arr[i] =  new int[COLS];
        return arr;
}
void readMat( int **arr, int ROWS, int COLS) // read values in the 2D array
{       for(int i=0;i<ROWS;i++)
        {       for(int j=0;j<COLS;j++)
                        cin>>arr[i][j];
        }
}
void displayMat( int **arr, int ROWS, int COLS) // displays values of 2D array
{       for(int i=0;i<ROWS;i++)
                for(int j=0;j<COLS;j++)
                        cout<<"  "<<arr[i][j];
}
void deleteMat(int **arr, int ROWS) // free memory allocated
{       for(int i=0;i<ROWS;i++)
```

```
            delete arr[i]; // deletes each column and row
        delete [] arr; // deletes the pointer arr

}
main()
        int **arr, i, j, ROWS, COLS; // arr is a pointer to pointer to int
{       cout<<"\n Enter the number of rows and columns in the array : ";
        cin>>ROWS>>COLS;
        arr = AllocateMemory(arr, ROWS, COLS); // function call
        cout<<"\n Enter the values of the array : ";
        readMat(arr, ROWS,COLS); // function call
        cout<<"\n The array is a follows : ";
        displayMat(arr, ROWS, COLS); // function call
        deleteMat(arr, ROWS); // function call

}
```

OUTPUT

```
Enter the number of rows and columns in the array : 2 2
Enter the values of the array : 1 2 3 4
The array is a follows : 1  2  3  4
```

Nowadays, memory debuggers such as `Purify`, `Valgrind`, and `Insure++` are widely used for detecting memory corruption errors.

Points to Remember

- 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, that is 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 and 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 pointed at 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 allocate memory for variables dynamically, a 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.
- Memory leakage occurs when memory is allocated but not released when it is no longer required.

Glossary

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 using the unary * 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 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. In 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. When memory is de-allocated using delete, it is returned back to the _____.
16. `new` returns _____ if unsuccessful.
17. A two-dimensional integer array is a _____.
18. An un-initialized memory contains _____.
19. Pointer to pointer stores _____.
20. The only integer value that can be assigned to a pointer variable is _____.
21. _____ can be used as parameter declaration to declare an array of integers passed to a function.
22. Dynamically allocated memory can be referred using _____.
23. _____ function returns memory to the heap.
24. A ragged array is implemented using an array of pointers to _____.

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. In 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. A pointer is a variable that represents the contents of a data item.
11. A fixed size of stack is allocated by the system and is filled as needed from the top to bottom.
12. All parameters passed to the called function will be stored on the stack.
13. When the memory for an array is allocated, its base address is fixed and it cannot be changed during program execution.
14. An array can be assigned to another array.
15. Memory leakage occurs when memory is allocated but not released when it is no longer required.
16. mat[i][j] is equivalent to *(*(mat + i) + j).
17. It is possible to add two pointer variables.
18. Pointer constants cannot be changed.
19. The value of a pointer is always an address.
20. `*ptr++` will add 1 to the value pointed by `ptr`.
21. Pointers of different types can be assigned to each other without a cast.
22. Adding 1 to a pointer variable will make it point 1 byte ahead of the memory location to which it is currently pointing.
23. Any arithmetic operator can be used to modify the value of a pointer.
24. Ragged arrays consume less memory space.

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 such as
   (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 block code of the `main()` program is stored 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

Review Questions

1. Write a short note on pointers.
2. Give the advantages of using pointers.
3. Compare pointer and array name.
4. What do you understand by a null pointer?
5. What is a void pointer?
6. Explain the difference between a null pointer and a void 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 between `ptr++` and `*ptr++`.
12. How are arrays related to pointers?
13. Can we have an array of function pointers? If yes, illustrate with the help of a suitable example.
14. Explain the term dynamic memory allocation.
15. Differentiate between `malloc()` and `new`.
16. Write a short note on pointers to pointers.
17. Differentiate between a function returning pointer to `int` and a pointer to function returning `int`.
18. Differentiate between pointer to constants and constant to pointers.
19. Explain the call by address technique of passing parameters to function.
20. How are pointers used on two-dimensional arrays?
21. How can you declare a pointer variable?
22. Differentiate between a variable address and a variable's value. How can we access a variable's address and its value using pointers?
23. Explain the different memory areas available to the programmer briefly.
24. What do you understand by dereferencing a pointer variable?
25. Write a short note on pointer expressions and pointer arithmetic.
26. What will `*p++ = *q++` do?
27. Write a short note on pointer and 3D array.
28. How can a pointer be used to access individual elements of an array? Illustrate with an example.
29. Can we assign a pointer variable to another pointer variable? Justify your answer with the help of an example.
30. What will happen if we add or subtract an integer to or from a pointer variable?
31. Is it possible to compare two pointer variables? Illustrate using an example.
32. Can we subtract two pointer variables?
33. 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?
34. Can array names appear on the left side of the assignment operator? Why?
35. Differentiate between an array name and an array pointer.
36. How can you have array of function pointers? Illustrate with an example.
37. Briefly discuss memory allocation process in C++ language.
38. With the help of an example, explain how pointers can be used to dynamically allocate space for two-dimensional array.

39. Explain memory corruption with help of suitable examples.
39. Differentiate between `*(arr+i)` and `(arr+i)`.
40. How can we access the value pointed by a pointer to a pointer?

Programming Exercises

1. Explain the result of the following code:

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

2. Write a program that illustrates passing of character arrays as an argument to a function using pointers.
3. Write a program to print your name using pointers.
4. Write a program to print a character. Print its ASCII value and rewrite the character in upper case.
5. Write a program to find class average. The result should contain only two digits after the decimal.
6. Write a program to calculate area of a circle.
7. Write a program to convert a floating point number into an integer.
8. Write a program to find second largest of three numbers.
9. Write a program to enter a character and then determine whether it a vowel or not.
10. Write a program to test whether a number is positive, negative, or equal to zero.
11. Write a program to take input from the user and then check whether it is a number or a character. If it is a character, determine whether it is in upper case or lower case.
12. Write a program to display the sum and average of numbers from m to n.
13. Write a program to print all even numbers from m–n.
14. Write a program to read numbers until -1 is entered. Display whether the number is prime or composite.
15. Write a program to calculate the area of a triangle.
16. Write a program to add two integers using functions. Use call by address technique of passing parameters.
17. Write a program to take input from the user and then check whether it is a number or a character. If it is a character, determine whether it is in upper case or lower case.
18. Write a program to read a character until * is entered. If the character is in upper case, print it in lower case and vice versa. Count the number of upper and lower case characters entered.
19. Write a program to read and print a text. Count the number of characters, words, and lines in the text.
20. Write a program to copy a character array into another character array.
21. Write a program to copy n characters of a character array from the mth position into another character array.
22. Write a program to copy the last n characters of a character array into another character array. Convert the lower case letters into upper case letters while copying.
23. Write a program to read a text, delete all the semi-colons it has and finally replace all '.' with a ';'.
24. Write a program to reverse a string.
25. Write a program to concatenate two strings into a third string.
26. Write a program to read and display values of an integer array. Allocate space dynamically for the array.
27. Write a function to calculate the roots of a quadratic equation. The function must accept arguments and return result using pointers.
28. Write a program using pointers to insert a value in an array.
29. Write a program using pointers to search a value from an array.
30. 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.
31. Write a program to compare two arrays using pointers.

Find the errors (if any) in the following codes

```
1. int ptr, *ptr;
2. int num, *ptr=num;
3. int *ptr=10;
4. int num, **ptr=&num;
5. int *ptr1, *ptr2, *ptr3=*ptr1+*ptr2;
6. int *ptr; cin>>&ptr;
```

Scan this QR code to access additional questions and study material.

Fill in the Blanks Answers:

1. 1 byte
2. Dynamic memory allocation
3. Pointer
4. Null
5. Data type
6. 2
7. Value
8. Variables storing addresses
9. Pointers
10. - (asterisk)
11. Memory
12. Stack, heap
13. Stack
14. Code, data
15. Free store
16. Null
17. Array
18. Garbage value
19. Address
20. 0
21. Pointer
22. malloc or calloc
23. Free
24. Pointers

State True or False Answers:

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

19. False
20. True
21. True
22. True
23. True
24. True

Multiple Choice Questions Answers:

1. (b) Dereferencing operator
2. (b) *num
3. (d) All of these
4. (b) Indirection
5. (a) &
6. (c) Global
7. (b) Heap

39. Explain memory corruption with the help of a suitable example.

Memory corruption occurs when a program unintentionally modifies memory that it does not own, leading to unpredictable behavior, crashes, or security vulnerabilities. It can happen due to buffer overflows, invalid pointer dereferencing, or improper memory allocation.

Example: Buffer Overflow

c Copy Edit

```c
#include <stdio.h>
#include <string.h>

int main() {
    char arr[5];  // Array of size 5
    strcpy(arr, "HelloWorld"); // Exceeding the buffer size (causes memory corruption)
    printf("%s\n", arr);
    return 0;
}
```

Explanation:

Here, `"HelloWorld"` is a 10-character string, but the array `arr` is only 5 bytes long. This causes a buffer overflow, corrupting adjacent memory.

---

40. Differentiate between `(arr+1)` and `(arr+1)`.

This question seems to have a duplication mistake. You might be asking about `(arr + 1)` vs. `*(arr + 1)`.

1. `(arr + 1)`
   - This gives the address of the next element in the array.
   - If `arr` is an integer array, `arr + 1` moves forward by `sizeof(int)` bytes.
   - Example:

c Copy Edit

```c
int arr[3] = {10, 20, 30};
printf("%p\n", arr);       // Address of arr[0]
printf("%p\n", arr + 1);   // Address of arr[1]
```

2. `*(arr + 1)`
   - This gives the value stored at the next memory location.
   - Example:

c Copy Edit

```c
int arr[3] = {10, 20, 30};
printf("%d\n", *(arr + 1));  // Output: 20 (value at arr[1])
```

41. How can we access the value pointed to by a pointer to a pointer?

A pointer to a pointer stores the address of another pointer. We use `**` (double dereference) to access the value stored at the final memory location.

Example:

c Copy Edit

```c
#include <stdio.h>

int main() {
    int a = 10;
    int *ptr = &a;    // Pointer to integer
    int **ptr2 = &ptr; // Pointer to pointer

    printf("Value of a: %d\n", **ptr2); // Accessing the value using double dereferenc
    return 0;
}
```

Explanation:

- `ptr` stores the address of `a`.
- `ptr2` stores the address of `ptr`.
- `**ptr2` accesses the value stored at `a`.

1. Write a short note on pointers.

A **pointer** is a variable that stores the memory address of another variable. Instead of storing values directly, it holds the location where the data is stored.

Example:

```c
int a = 10;
int *p = &a; // p stores the address of a
```

Uses of Pointers:

- Dynamic memory allocation
- Efficient array handling
- Function pointers for callbacks

---

2. Give the advantages of using pointers.

- **Efficient memory usage** – Allows direct memory access.
- **Dynamic memory allocation** – Used in data structures like linked lists.
- **Function Pointers** – Useful for callbacks and event handling.
- **Array manipulation** – Enables flexible array operations.

---

3. Compare pointer and array name.

Feature	Pointer	Array Name
Storage	Stores an address	Represents the first element's address
Modifiable?	Yes	No
Arithmetic	`ptr++` moves to the next element	`arr` is fixed

Example:

```c
int arr[] = {10, 20, 30};
int *ptr = arr;
ptr++; // Moves to next element
```

---

4. What do you understand by a null pointer?

A **null pointer** is a pointer that doesn't point to any valid memory location. It is used for safety to avoid accessing garbage values.

Example:

```c
int *ptr = NULL;
```

5. What is a void pointer?

A **void pointer** (`void *`) is a generic pointer that can store the address of any data type.

Example:

c Copy Edit

```c
void *ptr;
int a = 10;
ptr = &a; // Valid
```

To access the value, explicit type casting is required.

---

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

Feature	Null Pointer	Void Pointer
Meaning	Points to nowhere	Can hold any type
Use	Avoids invalid memory access	Used for generic memory access

Example:

c Copy Edit

```c
int *ptr = NULL;  // Null pointer
void *ptr2;       // Void pointer
ptr2 = &a;        // Can hold any type
```

---

7. What is an array of pointers? How is it different from a pointer to an array?

- **Array of Pointers** – A collection of pointer variables.
- **Pointer to an Array** – A single pointer that points to an entire array.

Example:

c Copy Edit

```c
int *arr[3];  // Array of pointers
int (*ptr)[3];  // Pointer to an array
```

---

8. Write a short note on pointer arithmetic.

Pointer arithmetic includes:

- **Increment** (`ptr++`) – Moves to the next element.
- **Decrement** (`ptr--`) – Moves to the previous element.
- **Addition/Subtraction** (`ptr + n`) – Moves `n` elements forward/backward.

Example:

c Copy Edit

```c
int arr[] = {10, 20, 30};
int *ptr = arr;
ptr++; // Moves to next element
```

9. How are generic pointers different from other pointer variables?

A **generic pointer** (`void *`) can store addresses of different data types, while normal pointers store only one specific type.

Example:

c Copy Edit

```c
void *ptr;
int a = 10;
ptr = &a;  // Valid
```

---

10. What do you understand by the term pointer to a function?

A **function pointer** stores the address of a function and allows calling the function via the pointer.

Example:

c Copy Edit

```c
void display() { printf("Hello!"); }
void (*ptr)() = display;
ptr(); // Calls the function
```

---

11. Differentiate between `ptr++` and `*ptr++`.

Expression	Meaning
`ptr++`	Moves the pointer to the next memory location
`*ptr++`	Accesses the current value and then moves the pointer

Example:

c Copy Edit

```c
int arr[] = {10, 20, 30};
int *ptr = arr;
printf("%d", *ptr++); // Prints 10, then moves to 20
```

---

12. How are arrays related to pointers?

An array name acts as a pointer to its first element, making array manipulation easier using pointers.

Example:

c Copy Edit

```c
int arr[] = {10, 20, 30};
int *ptr = arr;
```

Both `arr[0]` and `*ptr` give the same value.

13. Can we have an array of function pointers? If yes, illustrate with the help of a suitable example.

Yes, we can have an array of function pointers to store multiple function addresses.

Example:

c Copy Edit

```c
#include <stdio.h>

void func1() { printf("Function 1\n"); }
void func2() { printf("Function 2\n"); }

int main() {
    void (*funcArr[2])() = {func1, func2}; // Array of function pointers
    funcArr[0](); // Calls func1
    funcArr[1](); // Calls func2
    return 0;
}
```

This allows calling different functions dynamically.

---

14. Explain the term dynamic memory allocation.

Dynamic memory allocation allows memory allocation at runtime using functions like `malloc()`, `calloc()`, `realloc()`, and `free()`.

Example:

c Copy Edit

```c
int *ptr = (int*)malloc(sizeof(int)); // Allocates memory for an integer
```

It is useful for data structures like linked lists and trees.

---

15. Differentiate between `malloc()` and `new`.

Feature	`malloc()` (C)	`new` (C++)
Memory type	Heap	Heap
Initialization	Does not initialize	Calls constructor (if object)
Returns	`void*` (needs casting)	Type-safe pointer
Freeing memory	`free(ptr);`	`delete ptr;`

Example:

c Copy Edit

```c
int *ptr = (int*)malloc(sizeof(int)); // C
int *ptr = new int; // C++
```

16. Write a short note on pointers to pointers.

A **pointer to pointer** stores the address of another pointer.

Example:

```c
int a = 10;
int *ptr = &a;
int **pptr = &ptr; // Pointer to pointer
printf("%d", **pptr); // Outputs 10
```

It is used in **multilevel indirection** and **dynamic memory management**.

---

17. Differentiate between a function returning pointer to int and a pointer to function returning int.

Concept	Function Returning Pointer	Pointer to Function
Meaning	Returns `int*`	Points to a function returning `int`
Example	`int* func();`	`int (*funcPtr)();`

Example:

```c
int* func() { return NULL; } // Function returning pointer
int func2() { return 10; }
int (*funcPtr)() = func2; // Pointer to function
```

---

18. Differentiate between pointer to constants and constant to pointers.

Concept	Pointer to Constant	Constant Pointer
Meaning	Cannot modify value	Cannot modify pointer address
Example	`const int *ptr;`	`int *const ptr;`

Example:

```c
const int *ptr;  // Value cannot be changed
int *const ptr2 = &a; // Address cannot be changed
```

---

19. Explain the call by address technique of passing parameters to a function.

In **call by address**, the function gets the address of arguments, allowing modification.

Example:

```c
void modify(int *p) { *p = 20; }
int main() {
    int a = 10;
    modify(&a); // Pass address
    printf("%d", a); // Outputs 20
}
```

20. How are pointers used on two-dimensional arrays?

A **2D array** is stored in contiguous memory, so pointers can access elements.

Example:

```c
int arr[2][2] = {{1,2},{3,4}};
int (*ptr)[2] = arr; // Pointer to array
printf("%d", ptr[1][1]); // Access element
```

---

21. How can you declare a pointer variable?

A pointer is declared using `*`.

```c
int *ptr; // Pointer to int
```

---

22. Differentiate between a variable address and a variable's value.

Concept	Address	Value
Meaning	Location in memory	Data stored
Example	`&a`	`a`

Example:

```c
int a = 10;
printf("%p", &a); // Address
printf("%d", a); // Value
```

---

23. Explain the different memory areas available to the programmer briefly.

- **Stack** – Stores local variables and function calls.
- **Heap** – Stores dynamically allocated memory.
- **Data Segment** – Stores global/static variables.
- **Code Segment** – Stores executable instructions.

---

24. What do you understand by dereferencing a pointer variable?

Dereferencing means accessing the value stored at the pointer's address.

```c
int a = 10, *ptr = &a;
printf("%d", *ptr); // Outputs 10
```

25. Write a short note on pointer expressions and pointer arithmetic.

Pointer arithmetic includes `ptr++`, `ptr--`, `ptr+n`, `ptr-n`.

Example:

```c
int arr[] = {10, 20, 30};
int *ptr = arr;
ptr++; // Moves to next element
```

---

26. What will `*p++ *q++` do?

It will **dereference** `p`, **increment** `p`, **dereference** `q`, and **increment** `q`, then multiply the values.

---

27. Write a short note on pointer and 3D array.

A **3D array** uses triple indexing, and pointers can access it.

```c
int arr[2][2][2] = {{{1,2},{3,4}},{{5,6},{7,8}}};
int (*ptr)[2][2] = arr;
printf("%d", ptr[1][1][1]); // Outputs 8
```

---

28. How can a pointer be used to access individual elements of an array?

Using `*ptr` and `ptr[index]`.

```c
int arr[] = {10, 20, 30};
int *ptr = arr;
printf("%d", *(ptr+1)); // Outputs 20
```

---

29. Can we assign a pointer variable to another pointer variable?

Yes, one pointer can store another's address.

```c
int a = 10;
int *p = &a;
int **q = &p;
```

---

30. What will happen if we add or subtract an integer to/from a pointer variable?

It will move the pointer in memory.

```c
int *ptr = arr;
ptr++; // Moves to next element
```

31. Is it possible to compare two pointer variables?

Yes, to check if they point to the same location.

```c
if (ptr1 == ptr2) { printf("Same location"); }
```

---

32. Can we subtract two pointer variables?

Yes, it gives the **difference in elements**.

```c
int diff = ptr2 - ptr1;
```

---

33. How can an array be passed to a function?

By passing its name, which acts as a pointer.

```c
void func(int *arr) { printf("%d", arr[0]); }
int main() { int arr[] = {1,2}; func(arr); }
```

---

34. Can array names appear on the left side of the assignment operator? Why?

No, because an array name is a **constant pointer**.

---

35. Differentiate between an array name and an array pointer.

An array name is **constant**, while a pointer is **modifiable**.

---

36. How can you have an array of function pointers?

```c
void (*funcArr[2])() = {func1, func2};
```

---

37. Briefly discuss memory allocation process in C++ language.

C++ uses **stack** for local variables and **heap** for `new`.

---

38. How can pointers dynamically allocate a 2D array?

```c
int **arr = (int**)malloc(rows * sizeof(int*));
for(int i=0; i<rows; i++)
    arr[i] = (int*)malloc(cols * sizeof(
```