ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3577

Program: PGD

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Programming in C/C++ Language (3577) • Level: PGD

Semester: Autumn 2026

Assignment No. 2

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant parts of the course book, understand every concept, type and run each program themselves, check its output, and then write the assignment in their own words. Copying this guide word for word is not allowed. The instructions printed in the booklet also state: you must answer the questions in your own words and copying from books is not allowed. All programs in this guide use standard C/C++ as taught at this level; sample inputs and outputs are illustrations, and students should test the programs with their own values as well.

USMAN NOTES www.usmannotes.com

Question 1: Arrays of Structures — Information of 5 Employees in C (20 Marks)

Question as printed: Explain the concept of arrays of structures. Write a C program to store and display information of 5 employees. (20)

Introduction

Real data rarely consists of single, separate values. An employee, for example, is described by several related items of different types at once: an employee number or name in characters, an age or years of service in whole numbers, and a salary in a decimal amount. An array alone cannot hold such a record, because all the elements of an array must be of one type; separate parallel arrays for the names, ages and salaries would be clumsy and easy to get out of step with one another. The structure solves the first half of the problem by grouping different types under one new type, and the array of structures solves the second half by holding many such records together. This question, which is set in C, brings the two ideas together: a structure for one employee, an array of five of them, and a program that stores and displays the information of five employees.

Structures and Arrays of Structures Explained

A structure is defined with the keyword struct, followed by a name and a list of members in braces, ending with a semicolon. For this question we may define: struct Employee { int id; char name[30]; char post[30]; float salary; }; This definition creates no variable; it only describes a new type, struct Employee, in the same way that int describes a type. A variable of the type holds all four members together in one block of memory, and each member is reached with the dot operator, as in emp.salary. An array of structures is then declared exactly like any other array: struct Employee staff[5]; This creates five employee records in a row, staff[0] to staff[4], each with its own id, name, post and salary, reached by combining the array index with the dot operator, for example staff[2].name means the name member of the third record. Because a

loop can visit the records one by one, input, display, searching and sorting all become short loop programs instead of five repeated blocks of code.

Table 1: Structure, array and array of structures compared

Concept	What it holds	Example in this question
Single variable	One value of one type	One salary: float salary;
Array	Many values, all of the same type	Five salaries: float salary[5];
Structure	One record of different types	One employee: id, name, post, salary
Array of structures	Many records, each of different-typed members	Five employees: struct Employee staff[5];

USMAN NOTES www.usmannotes.com

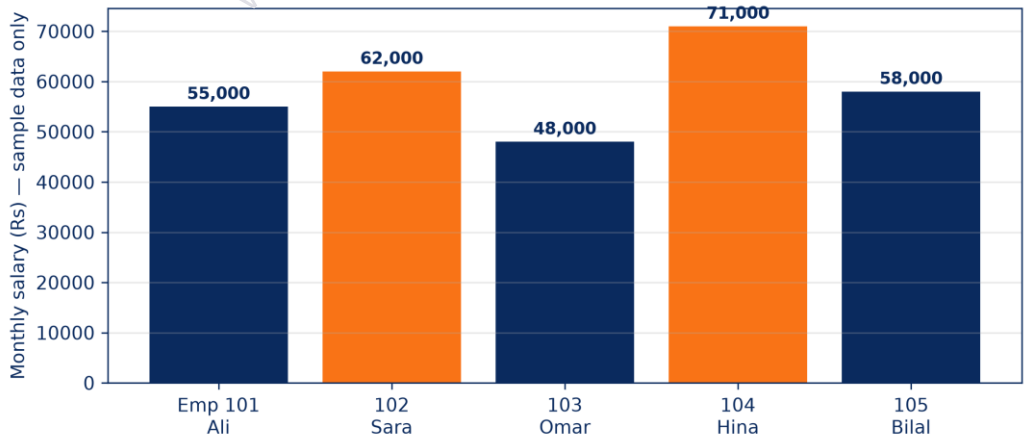


Figure 1: The five sample employee records used in the program, shown by their salaries (illustrative sample data only).

Program in C: Store and Display Information of 5 Employees

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

```
struct Employee
{
    int id;
    char name[30];
    char post[30];
    float salary;
};

int main()
{
    struct Employee staff[5];
    int i;

    printf("Enter information of 5 employees:\n");

    for (i = 0; i < 5; i++)
    {
        printf("\nEmployee %d\n", i + 1);

        printf("Enter ID: ");
        scanf("%d", &staff[i].id);

        printf("Enter name (one word): ");
        scanf("%s", staff[i].name);
```

USMAN NOTES www.usmannotes.com

```
        printf("Enter post (one word): ");
        scanf("%s", staff[i].post);

        printf("Enter salary: ");
        scanf("%f", &staff[i].salary);
    }

    printf("\n----- Employee Records -----\n");
    printf("ID\tName\tPost\tSalary\n");

    for (i = 0; i < n; i++)
    {
        printf("%d\t%s\t%s\t%.2f\n",
               staff[i].id, staff[i].name,
               staff[i].post, staff[i].salary);
    }

    return 0;
}
```

Explanation of the Program

The program is written in C, as the question requires, so it uses the C library stdio.h and the functions printf for output and scanf for input instead of C++ streams. The structure definition is placed before main so that the type is known throughout the program. Inside main, staff is

declared as an array of five structures. The first for loop, with i from 0 to 4, fills one record per repetition. Notice the details of scanf: the format %d reads an integer, %f reads a float, and %s reads a word into a character array. The ampersand & is required before ordinary variables such as &staff[i].id and &staff[i].salary because scanf must be given the address of the variable it is to fill; it is not written before staff[i].name, because the name of a character array already gives the address of its first element. The %.2f in the display loop prints the salary with two decimal places, the normal way to show money. The second loop then prints all five records in a table, using tab characters \t to line the columns up under the headings ID, Name, Post and Salary.

A sample set of records - ids 101 to 105, with names Ali, Sara, Omar, Hina and Bilal, posts such as Clerk, Teacher, Manager, Officer and Assistant, and salaries 55000, 62000, 48000, 71000 and 58000 — is used only to illustrate the input and output; students should enter their own sample values when they run the program. The important observation is that the record for each person stays together: Sara's salary can never accidentally be displayed beside Omar's name, as could happen with separate parallel arrays if one of them were sorted or edited and the others were not. This togetherness, called cohesion of data, is the real value of the structure, and it is the idea from which the class in object-oriented C++ later develops, where data and the functions working on it are grouped even more completely.

Uses, Advantages and Precautions

- Records of students, patients, books, products and customers are all naturally stored as arrays of structures, and files of such records form the basis of simple databases.

- Useful operations become easy loops: finding the highest-paid employee, searching for a record by id, counting staff above a salary limit, or sorting the array by name or salary.
- Names with spaces cannot be read by scanf with %s, which stops at the first space; in C the function fgets is used for a full line of text. For a course exercise, single-word names keep the program simple, but the limitation should be known.
- The size written in a character array must be respected, and input longer than the array can overflow into other memory — one of the classic dangers of C, and a reason to keep test data reasonable.
- A structure variable can be assigned to another of the same type in one statement, and structures can be passed to functions, which keeps programs that use records short and readable.

USMAN NOTES www.usmannotes.com

Conclusion

In conclusion, a structure groups the different-typed items of one record under a single new type, and an array of structures holds many such records in an indexed collection, each member of each record being reached by combining an index with the dot operator, as in staff[i].salary. The C program given defines an Employee structure with id, name, post and salary, fills an array of five such structures in a loop using scanf, and displays all five records in a table using printf. Keeping each employee's data together in one record is what makes the method safe and convenient, and the same idea underlies student records, stock lists and every other record-handling program the student will meet later.

Question 2: Pointers — Accessing and Modifying a Variable (20 Marks)

Question as printed: Write a C++ program that demonstrates the use of pointers by accessing and modifying the value of a variable using a pointer. (20)

Introduction

Every variable in a program occupies a place in the computer's memory, and every place in memory has a number, called its address. A pointer is a variable whose value is not an ordinary number, character or amount, but the address of another variable. Pointers are often described as the most powerful and the most dangerous feature inherited by C++ from C: powerful because they let a program work directly with memory, build linked lists and trees, handle arrays efficiently, and let a function change the caller's variables; dangerous because a pointer used carelessly can point to the wrong place and corrupt data. This answer explains the address operator and the dereference operator, and then gives the required program, in which the value of a variable is both read and changed through a pointer, with its address and value printed at each stage so the effect is fully visible.

The Two Pointer Operators

The address-of operator & gives the address of a variable: &number means 'the address at which number is stored'. A pointer is declared by writing the type of the variable it will point to, followed by an asterisk: int* p; declares a pointer that can hold the address of an int. The type matters because it tells the compiler how many bytes to read when the pointer is used and how far to move when pointer arithmetic is done. The dereference, or indirection, operator * goes in the opposite direction: if p holds the address of number, then *p means 'the value stored at

the address held in p', which is simply another name for number itself. Therefore *p = 50; does not change p; it changes the variable that p points to. Remembering that & gives an address from a variable and * gives a value from an address is the whole secret of pointers, and every diagram in the course book is really a picture of these two movements.

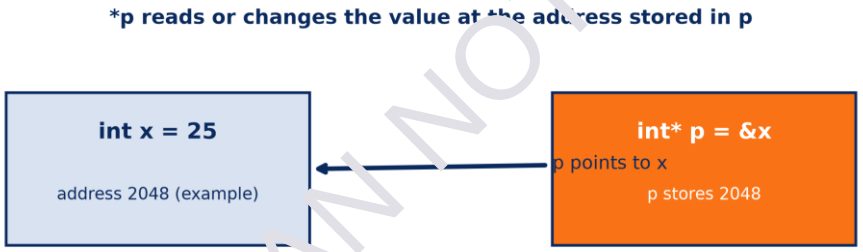


*Figure 1: A pointer stores the address of a variable. Reading or writing *p reads or writes that variable. Addresses shown are examples only.*

Program: Accessing and Modifying a Value Through a Pointer

```
#include <iostream>
using namespace std;


int main()
{
    int number = 25;
    int* p = &number;


    cout << "Value of number          = " << number << endl;
    cout << "Address of number (&number) = " << &number << endl;
```

```
    cout << "Value stored in p        = " << p << endl;
    cout << "Value pointed to (*p)    = " << *p << endl;

    *p = 50;

    cout << "\nAfter changing the value through the pointer:" << endl;
    cout << "Value of number          = " << number << endl;
    cout << "Value pointed to (*p)    = " << *p << endl;

    return 0;
}
```

USMAN NOTES www.usmannotes.com

Explanation of the Program

The program begins with int number = 25; and then declares the pointer and initialises it in the same statement: int* p = &number;. Initialising a pointer at declaration, or setting it to nullptr when it does not yet point anywhere, is an essential safety habit, because an uninitialised pointer contains a random address and using it can damage anything. The first group of output statements separates the three ideas that beginners most often confuse. number prints 25, the value. &number prints an address, shown by the system in hexadecimal form such as 0x61ff08 — the exact address differs on every run and every machine, so students must not expect a particular value. p prints the same address, because that is what was stored in it, and *p prints 25 again, proving that going through the pointer reaches exactly the same value. Then comes the key line, *p = 50;, which stores fifty in the memory location whose address is in p — that is,

in number. The final output shows that number is now 50 although the name number never appeared on the left of an assignment after its declaration, and *p is also 50, as it must be, since both names now describe the same memory location.

Table 1: The three expressions compared

Expression	What it means	Value in this program
number	The variable itself	25 at first, 50 after *p = 50
&number	The address of the variable	Example: 0x61ff08 (varies each run)
p	The pointer — an address stored as a value	The same address as &number
*p	The value at the address stored in p	25 at first, 50 after the change

USMAN NOTES www.usmannotes.com

Why Pointers Matter and How to Use Them Safely

Pointers are not an academic decoration. Arrays and pointers are closely related: the name of an array behaves like a pointer to its first element, which is why functions can process an array without copying it. Dynamic memory, obtained with the new operator, is always handled through pointers, allowing a program to request exactly as much memory as it needs while running and to release it with delete. Passing a pointer, or its safer modern relative, a reference, to a function allows that function to modify the caller's variables, which is how a swap function exchanges two values. Data structures such as linked lists, stacks, queues and trees are built entirely from structures joined by pointers. The safety rules are equally

important: always initialise a pointer, never use a pointer after its memory has been deleted, never let a pointer point to a variable that has gone out of scope, and set a finished pointer to nullptr so that a later accidental use fails visibly rather than silently. A pointer that points nowhere valid is called dangling or wild, and most mysterious program crashes in C and C++ are traced to such pointers.

It is also helpful to compare pointers with references, which this course introduces as the more convenient C++ alternative for many jobs. A reference, declared with &, is an alias for a variable: it must be initialised, it cannot be made to refer to a different variable later, and it is used with the same syntax as the variable itself, without *. A pointer can be reseated to point elsewhere, can be nullptr, and can be moved by arithmetic to walk through an array — flexibility that is needed for dynamic structures, but which is unnecessary, and therefore an avoidable risk, for simply referring to one variable. Choosing a reference for simple aliasing and a pointer when the address itself must be stored, changed or compared is good modern practice.

Conclusion

To conclude, a pointer is a variable that stores the address of another variable. The operator & obtains that address, the declaration int* p creates a pointer to an int, and the operator * follows the pointer to read or write the value at the stored address. The program demonstrates the complete idea in its clearest form: number starts at 25, p is set to &number, the value is read as *p, then *p = 50 changes number itself to 50, which the final output proves. Mastering this one small program — and the distinction among number, &number, p and *p shown in the

table — is the foundation for arrays, dynamic memory, functions that modify their arguments, and the linked data structures studied later in programming.

Question 3: The Role of Loops — Fibonacci Series of 15 Terms Using a while Loop (20 Marks)

Question as printed: Explain the role of loops in p ogramming. Write a C++ program to generate the Fibonacci series of the first 15 terms using a while loop. (20)

Introduction

Repetition is at the heart of computing. A computer's great strength is not that it performs any single calculation wonderfully, but that it performs millions of simple operations quickly, accurately and without tiring or losing concentration. Loops are the statements that organise this repetition. Without them, printing the marks of a hundred students would require a hundred print statements, and processing a file would be impossible because we would not know in advance how much code to write. This answer explains the role of loops in programming, compares the three loop statements of C++, and then gives the required program, which generates the first fifteen terms of the Fibonacci series — the series 0, 1, 1, 2, 3, 5, 8, ... in which each term is the sum of the two terms before it — using a while loop, as the question specifically demands.

The Role of Loops in Programming

Loops serve several distinct purposes. They repeat an action for every item of a collection, as when each element of an array is printed or totalled. They accumulate results, as in the sum

and average program of Assignment No. 1, where a running total is built up one value at a time. They search, examining items until the wanted one is found. They validate input, asking again and again until the user enters an acceptable value. They drive processes that continue while a condition holds, such as reading records from a file until its end, or running a game until the player chooses to quit. Good use of loops keeps programs short, general and easy to change: altering one number, the count, adapts the same loop to ten items or ten thousand. The two dangers are the infinite loop, in which the condition never becomes false because the variable controlling it is not updated, and the off-by-one error, in which the loop runs once too often or once too few; both are avoided by carefully tracing the first and last repetitions.

Table 1: The three loops of C++ compared

Loop	When the condition is tested	Best used when	Example use
for	Before each repetition	The number of repetitions is known	Reading exactly 10 numbers
while	Before each repetition	Repetitions continue while a condition holds; possibly zero times	Reading until a special value is entered
do-while	After each repetition	The body must run at least once	Showing a menu at least once before asking

The Fibonacci Series and the while Loop Plan

The Fibonacci series begins with 0 and 1, and every later term is the sum of the previous two: 0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, and so on. The series appears in nature, in art and in the analysis of algorithms, and generating it is a classic loop exercise because the next value depends on the two before it. The plan for the program keeps three variables: first holds the earlier term, second holds the later term, and next is calculated as first + second. After printing, the window slides forward: first takes the old value of second, second takes the value of next, and the loop repeats. The order of these two assignments matters — second must be copied into first before second is overwritten. A counter, count, starting from 2 because the first two terms are printed before the loop, controls the while loop so that exactly fifteen terms appear in total.

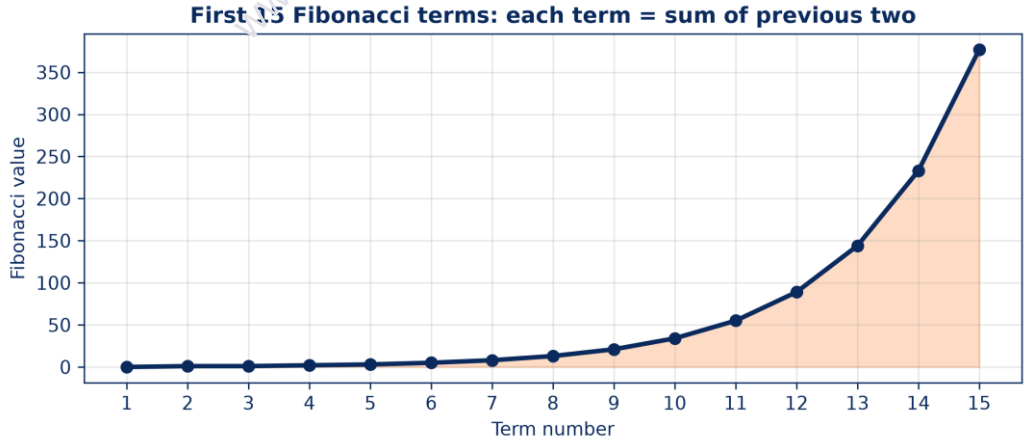


Figure 1: The first 15 Fibonacci terms: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377.

Program: First 15 Fibonacci Terms Using a while Loop

```
#include <iostream>

using namespace std;
```

```
int main()
{
    int first = 0;
    int second = 1;
    int next;
    int count = 2;

    cout << "Fibonacci series of the first 15 terms:" << endl;
    cout << first << "  " << second << "  ";

    while (count < 15)
    {
        next = first + second;
        cout << next << "  ";

        first = second;
        second = next;
        count++;
    }

    cout << endl;
    return 0;
}
```

USMAN NOTES www.usmannotes.com

Explanation of the Program and Its Trace

The first two terms, 0 and 1, cannot be produced by adding two previous terms, because there are not two previous terms; they are therefore printed before the loop and the counter is started at 2 to record that two terms are already shown. Each pass through the while loop produces exactly one new term. In the first pass, next = 0 + 1 = 1, which is printed; then first becomes 1 and second becomes 1, and count rises to 3. In the second pass, next = 1 + 1 = 2; the pair slides to first = 1, second = 2. The third pass gives 1 + 2 = 3, the fourth gives 2 + 3 = 5. The loop continues while count is less than 15, so it runs thirteen times, producing thirteen further terms, and stops when count reaches 15 — giving fifteen terms in all: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377. This count can be checked simply by counting the printed numbers, and the last few terms can be verified by hand: 89 + 144 = 233 and 144 + 233 = 377. If count++ were omitted, count would remain 2 forever, the condition would never fail, and the program would print numbers without end — a perfect miniature example of an infinite loop and of why the update step is part of the logic, not an afterthought.

Two extensions are worth noting. The same algorithm inside a for loop, with the counter in the loop heading, is equally correct; the question asks for a while loop because the exercise is designed to practise that form, in which initialisation, condition and update appear in three separate places and the student must keep all three consistent. Also, Fibonacci numbers grow quickly — the twentieth term is 6765 and the fiftieth exceeds twelve billion — so an int soon overflows and begins to print negative values; for long series a long long type is needed, which is a practical illustration of the data-type discussion in Assignment No. 1. Choosing the required

number of terms from the user, by reading n and using it in place of 15, turns this fixed program into a general one with almost no change, again showing the economy that loops give.

Conclusion

To conclude, loops are the mechanism by which programs repeat work: they process collections, accumulate totals, search, validate input and continue a process while a condition holds, and the while loop tests its condition before each repetition and is ideal when the repetitions depend on a changing condition. In the Fibonacci program the while loop slides a pair of variables forward, calculating each new term as the sum of the two before it, until a counter shows that fifteen terms have been printed. The trace confirms the output term by term, and the discussion of the counter shows both how the loop is guaranteed to stop and how easily it would run forever if the update were forgotten.

Question 4: Reading a File and Counting Characters, Words and Lines (20 Marks)

Question as printed: Write a C++ program that reads text from a file and counts the number of characters, words, and lines in the file. (20)

Introduction

Data kept only in variables disappears when a program ends. Files are how programs store information permanently on disk and work with information produced by other programs and by people. Word processors count words, editors report lines, and many utilities summarise text files, so counting the characters, words and lines in a file is a realistic task that combines

file handling with simple counting logic. In C++ file work is done with the streams of the header <fstream>: ifstream for an input file stream, which is read in much the same way as cin, and ofstream for output. This answer explains the method carefully — including the definitions of what counts as a character, a word and a line — gives the complete program, and works through a small example by hand so the counts can be verified.

How File Reading and Counting Work

The program opens a text file, by default named input.txt and placed in the same folder as the program, by constructing an ifstream object with the file name. The first duty is to check that the open succeeded, by testing the stream in a condition: if the file does not exist or cannot be opened, the program prints a clear error message and ends, instead of counting nothing and misleading the user. The file is then read one character at a time with the get function, which, unlike the extraction operator >>, does not skip white space — essential here, because spaces and new lines are exactly what the counting depends on. The definitions used are the standard simple ones. Every character read, including spaces and punctuation but handled as described below for the newline, adds one to the character count in the version explained by the trace table; a word is a sequence of characters separated by white space, so the program counts a new word whenever it moves from white space into a non-space character, or equivalently, as in the program below, counts words by reading word by word in a second simple method discussed after the code; and each newline character ends a line, with a final line that has no ending newline also deserving to be counted. Students must state their definitions, because different tools differ slightly — for example some counts include the newline as a character and some do not.

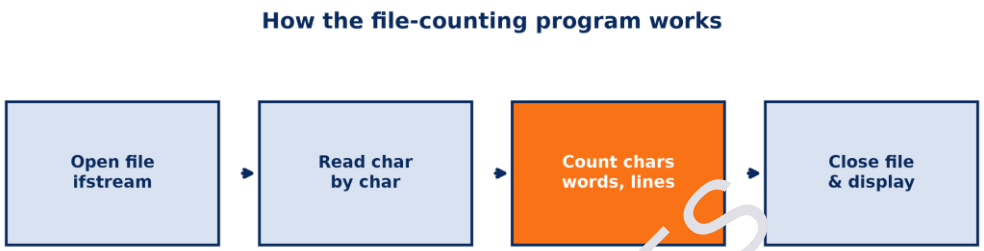


Figure 1: The stages of the file-counting program: open, read character by character, count, close and display.

Program: Counting Characters, Words and Lines in a File

```
#include <iostream>
#include <fstream>
using namespace std;


int main()
{
    ifstream fin("input.txt");
    char ch;
    int characters = 0;
    int words = 0;
    int lines = 0;
    bool inWord = false;
    bool endsWithNewline = false;


    if (!fin)
```

```
{
    cout << "Error: input.txt could not be opened." << endl;
    return 1;
}

while (fin.get(ch))
{
    characters++;
    endsWithNewline = (ch == '\n');

    if (ch == '\n')
        lines++;

    if (ch == ' ' || ch == '\n' || ch == '\t')
    {
        inWord = false;
    }
    else if (!inWord)
    {
        words++;
        inWord = true;
    }
}

/* If the file is not empty and does not end with a newline,
   the last line has no '\n', so count it as well. */
```

USMAN NOTES www.usmannotes.com

```
    if (characters > 0 && !endsWithNewline)
        lines = lines + 1;

    fin.close();

    cout << "Characters = " << characters << endl;
    cout << "Words      = " << words << endl;
    cout << "Lines      = " << lines << endl;

    return 0;
}
```

USMAN NOTES www.usmannotes.com

Explanation of the Program

The state variable inWord is the interesting part of the logic. It records whether the previous character was part of a word. When a non-space character is found and inWord is false, a new word has just started, so words increases by one and inWord becomes true; following letters of the same word find inWord already true and add nothing, so each word is counted exactly once, at its first letter. When a space, tab or newline is found, inWord is set back to false, ready for the next word. This technique is called a simple state machine, and it is far more reliable than counting spaces and adding one, which fails when a line begins or ends with spaces or when words are separated by several spaces. The character counter increases for every character returned by get, including spaces, tabs and newlines, and the line counter increases at each newline; after the loop, one more line is added only if the file is not empty and its last

character was not a newline, so a final unterminated line is counted as a line as well, while a file that properly ends with a newline is not over-counted. Stating this convention matters, because different tools differ slightly in such details. The file is closed with fin.close() when reading is finished; closing releases the file for other programs and is a habit that should never be omitted, even though the stream would close automatically when the program ends.

Table 1: Hand check on a small file whose last line ends without a newline

Content of input.txt (3 lines)	Count	Reason
Line 1: Programming in C++	—	18 characters including 2 spaces
Line 2: is interesting	—	14 characters including 1 space
Line 3: and useful	—	10 characters including 1 space
Total characters	44	42 visible characters + 2 newline characters between the three lines, as counted by get
Total words	7	Programming, in, C++, is, interesting, and, useful
Total lines	3	2 newline characters + 1 final line without a newline

The figures in the table are a hand-worked illustration on one specific small file, presented so that students can repeat the exercise on their own file and compare; the essential point is the method, not the particular numbers, and any student whose program disagrees with a careful hand count on the same file should first check the stated conventions about newlines before assuming the code is wrong. Testing should also include an empty file, which must report zero

characters, zero words and zero lines without crashing, and a file with several spaces between words, which must not inflate the word count. A missing file must produce the error message, not a false report of zeros — this is why the open check is tested separately. These four tests — normal, empty, widely spaced and missing — give good confidence in a file program.

Practical Uses and Variations

Counting programs are the ancestors of real text utilities. The Unix tool wc, used daily by programmers, reports exactly lines, words and characters for any file. Word limits in assignments — including the assignments in this course — are checked by similar logic. Once a file can be read character by character, many variations follow naturally: counting vowels and consonants, counting how many times a given word occurs, copying a file while converting it to capital letters, or reading student records from a file into an array of structures such as the one in Question 1 of this assignment. Reading the file word by word with fin >> word is a simpler alternative for the word count alone, because >> automatically skips white space, but it cannot by itself give a true character count, which is why the character-by-character method with get is taught as the main solution here.

Conclusion

To conclude, the program opens a text file with ifstream, checks that the open succeeded, and reads it character by character with get so that no white space is skipped. A character counter totals every character, a newline counter totals the lines with a stated convention for the final line, and a state variable that remembers whether we are inside a word counts each word exactly once at its first letter. The file is then closed and the three results displayed. The hand-

checked example and the tests for empty, widely spaced and missing files show how such a program is verified — an essential part of file handling, where the data is not typed in front of the programmer but read from a disk file that may be in any state.

Question 5: Arrays and the Sum of Elements in a 2D Array (Matrix) (20 Marks)

Question as printed: Explain the concept of arrays with an example. Write a C++ program to calculate the sum of elements in a 2D array (matrix). (20)

Introduction

An array is the first and most fundamental data structure a programmer meets. It stores a collection of values, all of the same type, in consecutive memory locations under a single name, and each value is accessed by a numbered position called an index. Whenever data naturally forms a list or a table — the marks of a class, the daily temperatures of a month, the entries of a timetable, or the numbers in a mathematical matrix — an array is the appropriate way to hold it. This answer explains the concept of arrays with an example, extends the idea from one dimension to two, and then gives the required program, which reads a two-dimensional array, or matrix, and calculates the sum of all its elements using nested loops.

The Concept of an Array, with an Example

The declaration int marks[5]; creates five integer locations named marks[0], marks[1], marks[2], marks[3] and marks[4]. The number in brackets in the declaration is the size, and it must be a positive whole number known at compile time in the basic form taught in this course. Indexing

begins at zero, a rule whose most frequent consequence is that the last index is size minus one, and stepping beyond it accesses memory outside the array. An array may be initialised at declaration, as in int marks[5] = {70, 85, 60, 90, 75};, and its great advantage is that a loop counter can be used as the index, so one short loop can read, print, search or total any number of elements. To illustrate: the sum of the five marks above is 70 + 85 + 60 + 90 + 75 = 380 and their average is 76 — calculated by exactly the accumulating loop of Assignment No. 1, now with numbers[i] in place of a single variable. The contrast with ordinary variables is clear: five separate variables mark1 to mark5 would each need its own statement for every operation, while the array is handled uniformly by loops, and the same code works unchanged for five hundred elements.

Two-Dimensional Arrays: Rows and Columns

A two-dimensional array adds a second index and is pictured as a table or matrix with rows and columns. The declaration int matrix[3][3]; creates nine integers arranged conceptually in three rows of three. The first index selects the row, the second the column, so matrix[1][2] is the element in row 1, column 2 — again counting from zero, so it is the item in the second row and third column as a person would count them. In memory the elements are actually stored row after row in one straight line, but the programmer should think and work in the table picture. Processing a 2D array requires nested loops: an outer loop that steps through the rows and an inner loop that, for each row, steps through the columns. This outer-inner pattern is the standard method for every matrix operation — printing, summing, finding the largest element, adding two matrices, and checking the diagonal elements where row and column indexes are equal.

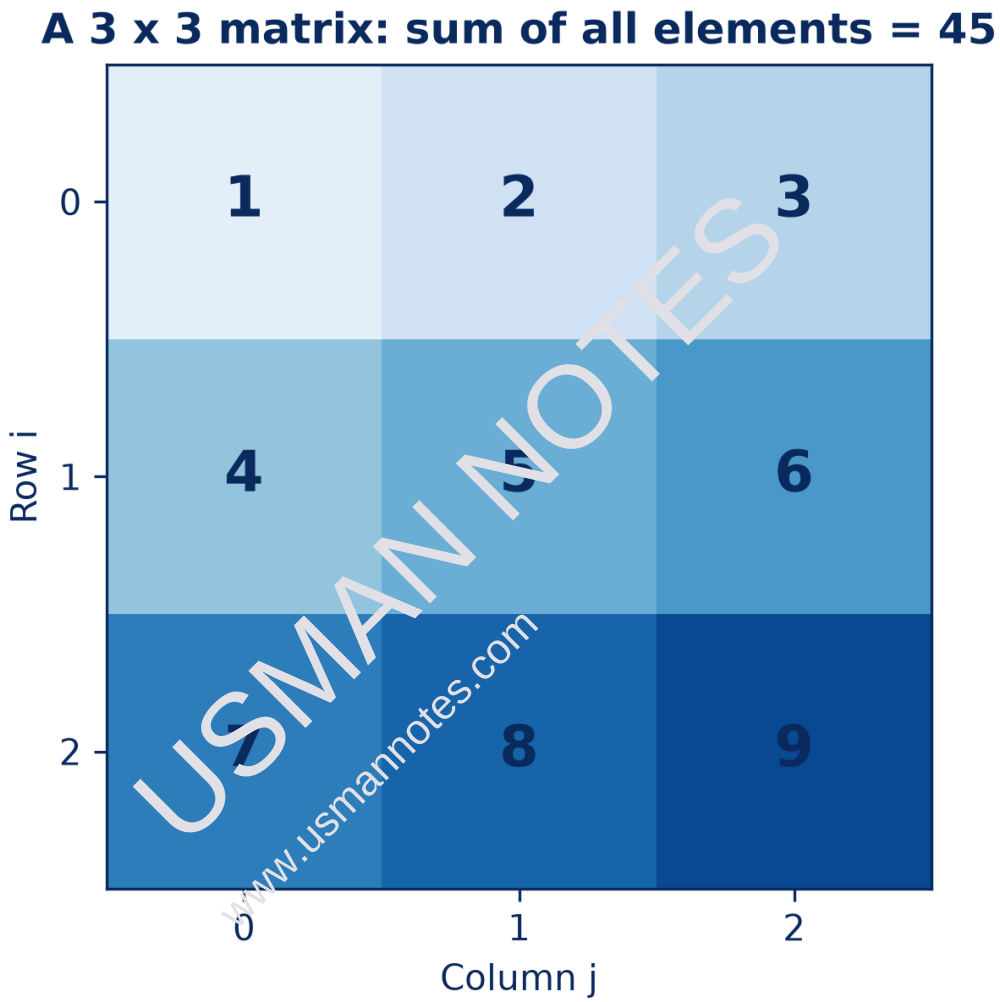


Figure 1: The example 3 x 3 matrix used in the explanation. The sum of its elements is 1+2+...+9 = 45.

Program: Sum of All Elements of a 2D Array

```
#include <iostream>
using namespace std;


int main()
{
    int matrix[3][3];
    int sum = 0;
```

```
cout << "Enter 9 elements of the 3 x 3 matrix:" << endl;
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        cout << "Enter element [" << i << "][" << j << "]: ";
        cin >> matrix[i][j];
    }
}

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        sum = sum + matrix[i][j];
    }
}

cout << "\nThe matrix is:" << endl;
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        cout << matrix[i][j] << "\t";
    }
    cout << endl;
```

USMAN NOTES www.usmannotes.com

```
        }

        cout << "\nSum of all elements = " << sum << endl;

        return 0;
    }
```

Explanation of the Program

The program declares a 3 by 3 integer matrix and an accumulator sum initialised to zero, following the rule already met in the sum program of Assignment No. 1: an accumulator that is not initialised gives a meaningless total. The first pair of nested loops fills the array: the outer loop fixes a row i, the inner loop runs j from 0 to 2 to fill the three columns of that row, and the prompt shows the indexes of the element being requested so that the user can follow the filling of the table. The second pair of nested loops visits the same nine elements and adds each to sum; the inner loop completes an entire row before the outer loop advances, so the elements are added row by row. The third pair prints the matrix in table form, using a tab after each element and endl after each row, which lets the user verify by eye that the stored matrix is the one intended before trusting the total. For the example matrix with rows 1 2 3, 4 5 6 and 7 8 9, the row sums are 6, 15 and 24, and the program's total is 6 + 15 + 24 = 45, a result easily checked by hand since the sum of the numbers from 1 to 9 is 9 x 10 / 2 = 45.

Table 1: The example matrix and its row sums

Row	Elements	Row sum

Row 0	1, 2, 3	6
Row 1	4, 5, 6	15
Row 2	7, 8, 9	24
Whole matrix	All nine elements	Total = 45

Discussion: Sizes, Variations and Errors

The fixed size 3 by 3 is used because the question and the course level call for a clear, concrete example, but the pattern generalises at once: declaring a larger array, for example int matrix[10][10];, and reading the actual numbers of rows and columns from the user, lets the same nested loops handle any matrix up to that capacity, because the loops are controlled by the row and column counts rather than by the constant 3. Variations that students should be able to produce from this program include the sum of each row separately, obtained by resetting a row accumulator inside the outer loop; the sum of the main diagonal, obtained by adding matrix[i][i]; and the largest element of the matrix, obtained by the assume-and-replace method of Assignment No. 1, Question 5, now applied in two dimensions. The common errors mirror those of one-dimensional arrays: confusing the order of the indexes, so that columns and rows are exchanged; using <= in the loop conditions and stepping one element too far in each direction; and forgetting that a two-dimensional array passed to a function must have its column size stated. Care with index bounds is the central discipline of array work in every dimension.

Arrays also have limits that motivate later topics. Their size is fixed once declared, they hold only one type, and inserting an element in the middle requires shifting others. Structures,

studied in Question 1, remove the one-type limit for a single record; dynamic memory and container classes such as vector remove the fixed-size limit. Seeing the array as the simple, fast, foundational member of this family — perfect for lists and tables whose size is known — helps the student choose the right tool in later programming rather than using arrays for everything or avoiding them entirely.

Conclusion

To conclude, an array is a fixed-size collection of same-type values stored consecutively and accessed by index, beginning at zero; a two-dimensional array adds a second index and represents a table of rows and columns. The program declares a 3 by 3 matrix, fills it and totals it with nested loops — an outer loop for the rows and an inner loop for the columns — adding every element to an accumulator initialised to zero, and prints the matrix so the result can be checked. For the example matrix the sum is 45, verified both by row sums and by the known total of the numbers 1 to 9, and the same nested-loop pattern extends directly to row sums, diagonals, and matrices of any size up to the declared capacity.