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. 1

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: Basic Input and Output Using cin and cout (20 Marks)

Question as printed: Write a C++ program to demonstrate basic input and output using cin and cout. Explain how these operators work in C++. (20)

Introduction

Every useful program communicates with the person using it. It receives data, called input, processes that data according to its instructions, and then presents results, called output. In

C++ this communication is organised as streams. A stream is simply a flow of characters: an input stream flows from a device such as the keyboard into the program, and an output stream flows from the program to a device such as the screen. The header file <iostream>, which stands for input-output stream, gives us the two objects that beginners meet first: cin, the standard input stream, normally connected to the keyboard, and cout, the standard output stream, normally connected to the monitor. Together with the stream insertion operator << and the stream extraction operator >>, they make input and output in C++ simple, safe and consistent across different types of data.

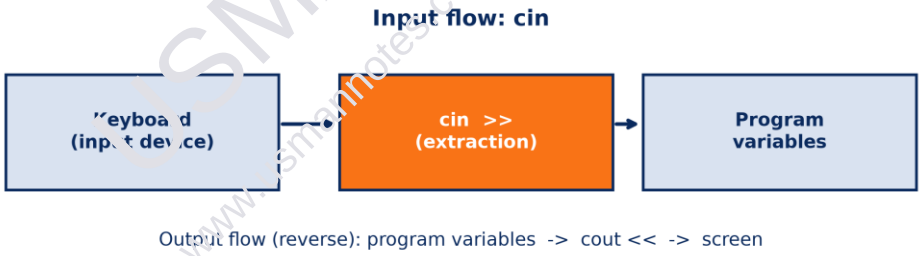


Figure 1: Input flows from the keyboard through cin into the program; output flows from the program through cout to the screen.

The cout Object and the Insertion Operator <<

cout is pronounced 'see-out'. It is used with the insertion operator <<, which inserts, or sends, the value written on its right side into the output stream on its left. For example, cout << "Hello"; sends the text Hello to the screen. Several insertions can be chained in one statement, as in cout << "Marks = " << marks << endl; because each insertion returns a reference to the same stream, so the next insertion can continue on it. The special manipulator endl ends the

line and flushes the stream, while the escape sequence '\n' also moves to a new line. cout automatically knows how to display each basic type: an int is printed as a whole number, a float or double as a decimal number, a char as a single character and a string as a sequence of characters. The programmer does not have to use different functions for different types, as was necessary in older languages; this is an example of operator overloading, where the same operator behaves sensibly for many types.

The cin Object and the Extraction Operator >>

cin is pronounced 'see-in'. It is used with the extraction operator >>, which extracts, or takes out, the next value from the input stream and stores it in the variable written on its right side, as in cin >> age;. The operator knows the type of that variable and converts the characters typed by the user accordingly: for an int it skips leading spaces, reads digits until a non-digit, and stores the whole number; for a float or double it also accepts a decimal point; for a char it normally skips white space and stores one character; and for a string it reads characters until the next white space, so a single word is read. Several extractions can also be chained, as in cin >> a >> b;, and the user may type the values separated by spaces or on separate lines. Input ends for one extraction as soon as enough characters for that type have been read, and the remaining characters stay in the stream for the next extraction.

Program: Demonstrating cin and cout

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


int main()
```

```
{
    char name[30];
    int age;
    float height;

    cout << "Enter your name (one word): ";
    cin >> name;

    cout << "Enter your age in years: ";
    cin >> age;

    cout << "Enter your height in metres: ";
    cin >> height;

    cout << "\n----- Your Information -----" << endl;
    cout << "Name   : " << name << endl;
    cout << "Age    : " << age << " years" << endl;
    cout << "Height : " << height << " metres" << endl;

    return 0;
}
```

USMAN NOTES www.usmannotes.com

Explanation of the Program, Step by Step

The first line, #include <iostream>, is a preprocessor directive that includes the declarations of cin, cout and the stream operators in the program; without it the compiler does not know these

names. The line using namespace std; makes the names of the standard library available without writing std:: before each one, although writing std::cout and std::cin is equally correct and is preferred in large projects. Inside main, three variables are declared to receive the input: a character array name for the name, an int age and a float height. Before every cin statement the program first prints a clear prompt with cout, because a waiting program gives the user no other sign of what to type; good prompting is an essential part of friendly input and output. Each cin >> statement then pauses the program until the user types a value and presses Enter. Finally the stored values are displayed in a neat, labelled form, with endl starting each item on a new line, so the output is easy to read and verify against what was typed.

Table 1: Sample run of the program

What appears on the screen	What the user types / what it means
Enter your name (one word):	Ali (stored in name)
Enter your age in years:	21 (converted and stored as the int 21)
Enter your height in metres:	1.75 (converted and stored as the float 1.75)
Name : Ali	Output produced by cout from the variable name
Age : 21 years	Output produced by cout from the variable age
Height : 1.75 metres	Output produced by cout from the variable height

Important Points About How cin and cout Work

- Direction of the arrows is a memory aid: << points towards cout, the destination of output, while >> points away from cin, towards the variable receiving the input.

- White space matters: pressing Space, Tab or Enter separates one input value from the next. For reading a full line including spaces, the function getline(cin, text) is used with a string variable instead of >>.
- Type safety: if the user types letters where a number is expected, the extraction fails and the stream enters a fail state; later inputs then do not work properly until the error is cleared. Checking input in this way is a more advanced topic, but beginners should at least test their programs with sensible data.
- Formatting can be controlled with manipulators from <iomanip>, such as setw for field width, fixed and setprecision for decimal places, which are often used when printing tables and amounts of money.
- Buffering: output collected in a buffer is displayed when the buffer is flushed, for example by endl or when input is requested; this is why a prompt should be printed before cin, so the user actually sees it in time.

A small but common beginner mistake is to reverse the operators, writing cin << or cout >>, which the compiler rejects because those combinations are not defined. Another is to forget the semicolon at the end of a statement, or to put the variable name in quotation marks in an output statement: cout << "age"; prints the word age, whereas cout << age; prints the value stored in the variable. Understanding this difference between a literal string and a variable is fundamental to all later programming.

Conclusion

To conclude, cin and cout are the standard stream objects of C++ for input and output. cout, with the insertion operator <<, sends values of any basic type to the screen, while cin, with the extraction operator >>, receives values from the keyboard and converts and stores them according to the type of the variable. Chaining, clear prompts, labelled output and endl for new lines make a program readable and user-friendly. The short program given above demonstrates all of these ideas together — declaring variables, prompting, reading three different types of values and displaying them neatly — and it forms the pattern on which almost every later program in this course is built.

Question 2: Data Types in C++ and a Program Using int, float and char (20 Marks)

Question as printed: Explain the use of different data types in C++. Write a program that takes integer, float, and character input from the user and displays them. (20)

Introduction

A program works with data, and data comes in different kinds: a whole number such as a student's age, a decimal number such as a temperature, a single character such as a grade, and a piece of text such as a name. In C++ every variable must be declared with a data type before it is used. The data type tells the compiler three things: how much memory to reserve for the variable, what kind of values it may store, and which operations are allowed on it. Choosing the correct type is not a formality. It prevents errors, saves memory and makes the intention of the

program clear to anyone who reads it. This answer explains the main data types of C++ and then gives the required program, which reads an integer, a float and a character from the user and displays them.

What a Data Type Does

When we write int marks; the compiler reserves on most modern systems, four bytes of memory, interprets the bits stored there as a whole number, and allows arithmetic such as addition and division with the rule that integer division discards the fractional part. When we write float price; the compiler reserves four bytes but interprets them in the floating-point format, so decimal fractions can be stored approximately. The same pattern of bits would mean something completely different under a different type, which is why type checking matters: the compiler stops us from accidentally treating a character as a number in a meaningless way, while still allowing deliberate conversions where they make sense. The sizeof operator tells us the size of any type or variable on our own system, and students should use it to verify the sizes quoted in books, because a few sizes, especially long, differ between systems.

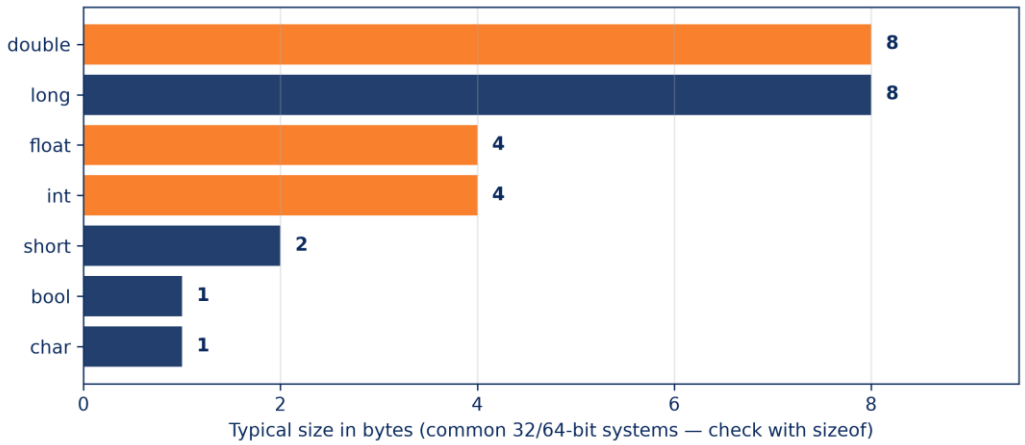


Figure 1: Typical sizes of common C++ data types. Exact sizes can vary by system and should be checked with sizeof.

Table 1: The main built-in data types of C++

Data type	Used for	Typical size	Example values
int	Whole numbers	4 bytes	25, -10, 0
float	Decimal numbers (single precision)	4 bytes	3.14, 98.6
double	Decimal numbers (double precision)	8 bytes	3.14159265
char	One character, stored as a code	1 byte	'A', '7', '+'
bool	True or false values	1 byte	true, false
short	Smaller whole numbers	2 bytes	-100, 32000
long / long long	Very large whole numbers	8 bytes	9876543210
void	No value (used for functions that return nothing)	—	—

USMAN NOTES www.usmannotes.com

Categories of Data Types

Built-in, or fundamental, types are the ones listed in the table: integer types (short, int, long, long long, and their unsigned forms, which store only zero and positive values but reach twice as far), floating-point types (float, double and long double, in increasing precision), the

character type char, whose values are really small integer codes — for example the code 65 stands for the letter A in the ASCII coding system — and the logical type bool, which stores only true or false and is used in conditions. Besides these, C++ provides derived types such as arrays, pointers, references and functions, which are built from the fundamental types, and user-defined types such as struct, class and enum, through which the programmer creates new types suited to a problem, for example a Student type combining a name, a roll number and marks. This question is concerned mainly with the fundamental types, but it is useful to see them as the foundation of the richer types studied later in the course.

Choosing the Right Type: Some Practical Rules

- Use int for counting, ages, marks and quantities that are naturally whole; use long long when the count may exceed about two billion.
- Use double in preference to float for general decimal work, because its extra precision reduces rounding errors; use float where memory is tight or the course question specifically asks for it, as this one does.
- Use char for a single character such as a grade or a Y/N reply, and a string or character array for words and names.
- Use bool for a condition or a flag, instead of using 0 and 1 in an int, because it states the intention clearly.
- Never store money in a floating-point type in a serious application without care: decimal fractions such as 0.1 cannot be represented exactly in binary, so small rounding errors can appear; beginners should simply be aware of the limitation.

Program: Reading and Displaying an Integer, a Float and a Character

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

int main()
{
    int number;
    float decimal;
    char symbol;

    cout << "Enter an integer: ";
    cin >> number;

    cout << "Enter a float value: ";
    cin >> decimal;

    cout << "Enter a character: ";
    cin >> symbol;

    cout << "\nYou entered:" << endl;
    cout << "Integer   = " << number << endl;
    cout << "Float     = " << decimal << endl;
    cout << "Character = " << symbol << endl;

    return 0;
```

USMAN NOTES www.usmannotes.com

```
}
```

Explanation of the Program

Three variables are declared, one of each required type: int number, float decimal and char symbol. The input statements follow the prompt-read-display pattern explained in Question 1. When the user types 42, the extraction operator converts the characters '4' and '2' into the integer value forty-two and stores it in number. When the user types 7.5, it stores an approximate binary representation of seven and a half in decimal. When the user types B, it stores the single character B, internally its character code, in symbol. The output statements then display each variable with a clear label, so the user can confirm that each value was stored in the intended variable and type. A sample run is: integer 42, float 7.5 and character B are entered, and the program prints Integer = 42, Float = 7.5 and Character = B, each on its own line.

Two details deserve attention. First, if a user types 7.9 for the integer, cin stores 7 in number and leaves .9 in the input stream, which would then be read as the next value; this shows that the type of the variable, not the user's typing, decides how input is interpreted. Second, a char variable holds exactly one character: typing a whole word for symbol stores only its first letter and leaves the rest in the stream. These behaviours are not faults; they are the defined rules of the extraction operator, and knowing them prevents much confusion when a program appears to skip an input — a problem usually caused by a leftover Enter key character, which can be handled by reading a character with cin.get or by using ws to skip white space.

Conclusion

In conclusion, data types are the foundation of correct programming in C++: each type fixes the memory used, the values allowed and the operations permitted. The fundamental types — int and its relatives for whole numbers, float and double for decimals, char for single characters and bool for true or false — cover the needs of most beginner programs, while arrays, structures and classes build richer types from them later. The program above shows the three types named in the question working together: one variable of each type is declared, filled with cin from the user's input and displayed with cout, proving in practical form that declaring the right type for the right job is what makes the stored values meaningful.

Question 3: Sum and Average of 10 Numbers Using a for Loop (20 Marks)

Question as printed: Write a C++ program to find the sum and average of 10 numbers using a for loop. (20)

Introduction

Adding a long list of numbers one by one, with ten separate input and addition statements, would make a program long, repetitive and hard to change. Loops exist precisely to repeat a set of steps under control. The for loop is the loop designed for the situation in which we know in advance exactly how many times the repetition must occur — and in this question we know it exactly: ten numbers are to be read, added to a running total, and finally divided by ten to obtain the average. This answer explains how the for loop works, gives the complete program,

traces it with a concrete set of numbers, and discusses the care needed with data types so that the average is not wrongly truncated.

How the for Loop Works

A for loop has the form: for (initialisation; condition; update) { statements }. The initialisation, such as int i = 1, runs once at the start and usually sets a counter. The condition, such as i <= 10, is tested before every repetition; the body runs only while it is true. The update, such as i++, runs after every repetition and moves the counter towards the end. When the condition becomes false, the loop stops and the program continues with the statement after the loop. The other essential idea in this program is the accumulator: a variable sum, started at zero before the loop, to which each new number is added by the statement sum = sum + number;. An accumulator must always be initialised; if it starts with an unknown value left in memory, the final total will be wrong by exactly that unknown amount.

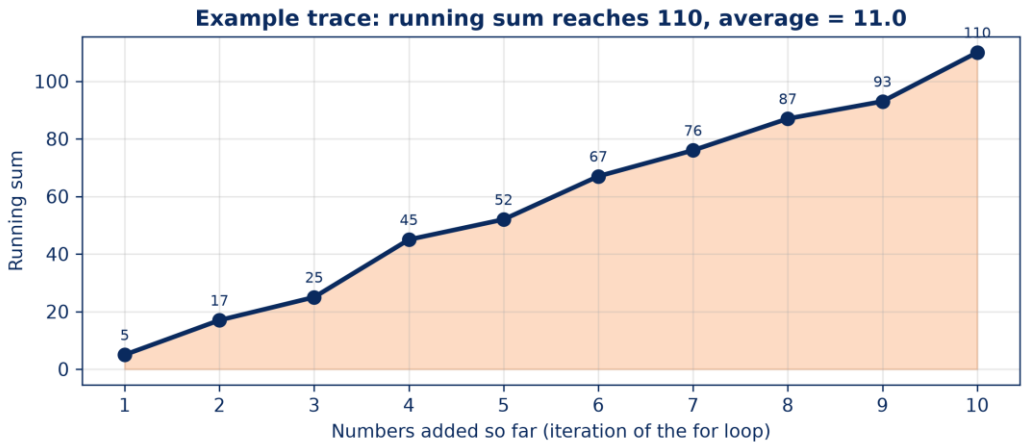


Figure 1: Example trace — ten numbers added one by one; the running sum reaches 110 and the average is 11.0.

Program: Sum and Average of 10 Numbers

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

int main()
{
    int number;
    int sum = 0;
    float average;

    cout << "Enter 10 numbers:" << endl;

    for (int i = 1; i <= 10; i++)
    {
        cout << "Enter number " << i << ": ";
        cin >> number;
        sum = sum + number;
    }

    average = sum / 10.0;

    cout << "\nSum of the 10 numbers     = " << sum << endl;
    cout << "Average of the 10 numbers = " << average << endl;

    return 0;
```

USMAN NOTES www.usmannotes.com

```
}
```

Explanation of the Program

The variables are number, which holds the value currently being read; sum, the accumulator, carefully initialised to zero; and average, declared as float so that it can hold a fractional result. Inside the loop, which runs with i taking the values 1, 2, 3 and so on up to 10, the program prompts for and reads one number and adds it to sum. Prompting with the counter, 'Enter number 3:', helps the user keep track of progress. After exactly ten repetitions the condition i <= 10 fails when i becomes 11, and the loop ends. Only then is the average calculated, outside the loop, because it is needed once, not ten times. The division is written as sum / 10.0, and this detail is vital: dividing an int by an int in C++ performs integer division and discards the fraction, so a sum of 105 would give 10 instead of 10.5. Writing 10.0, or converting with (float)sum / 10, forces real division and the correct average.

Table 1: Trace of the loop for the numbers 5, 12, 8, 20, 7, 15, 9, 11, 6, 17

Iteration i	Number read	sum after adding
1	5	5
2	12	17
3	8	25
4	20	45
5	7	52
6	15	67
7	9	76

8	11	87
9	6	93
10	17	110 (loop ends; average = 110 / 10.0 = 11.0)

Testing, Variations and Common Errors

A program should be tested with values whose answer is known in advance. Entering ten copies of the number 5 must give a sum of 50 and an average of 5; entering 1 to 10 gives a sum of 55 and an average of 5.5, which immediately exposes the integer-division mistake if it has been made, because the faulty version would print 5. Including a negative number in a test checks that signed values are handled. The most common errors in this program are: forgetting to initialise sum, writing the condition as i < 10, which reads only nine numbers, placing the average calculation inside the loop, and using the same name for the counter and another variable. Each error is easy to make and easy to find by a careful trace like the table above, in which the values of i, number and sum are written down step by step.

The same pattern extends naturally. Changing 10 to a variable n read from the user makes the program work for any count; storing the numbers in an array, as in Question 5, allows them to be used again after reading; and adding a largest-so-far or smallest-so-far test inside the loop turns it into the searching pattern. Recognising that counting, accumulating and searching are the three great uses of loops helps a student see that this short program is not an isolated exercise but a template used throughout computing — in totalling a shopping bill, averaging class marks and summarising sensor readings.

One further point about the counter deserves mention, because it affects how students should write the loop. Declaring the counter inside the loop heading, as in for (int i = 1; i <= 10; i++), keeps i confined to the loop: it exists only while the loop is running and cannot accidentally be used, or changed, elsewhere in the program. This is called the scope of the variable, and keeping the scope of every variable as small as the task allows is good practice that prevents a whole family of mistakes in longer programs. The loop would still work if i were declared before the loop, as older textbooks often did, but the modern, confined form is clearer. Similarly, the numbers themselves could be declared as float instead of int if decimal marks or measurements were to be averaged; the loop and accumulator logic would remain exactly the same, showing again that the pattern, once learned, transfers unchanged to many different kinds of data and many different problems.

USMAN NOTES www.usmannotes.com

Conclusion

To conclude, the for loop is the correct tool when the number of repetitions is known in advance, and an accumulator initialised to zero is the correct method for building a total. The program reads ten numbers inside a for loop controlled by a counter from 1 to 10, adds each to sum, and after the loop divides by 10.0 to obtain a true average, avoiding integer division. The trace table proves the working with concrete numbers whose sum, 110, and average, 11.0, can be checked by hand, and the testing notes show how known cases reveal the typical mistakes before the assignment is submitted.

Question 4: Conditional Statements — Positive, Negative or Zero (20 Marks)

Question as printed: Explain conditional statements in C++. Write a program that checks whether a given number is positive, negative, or zero. (20)

Introduction

A program that always followed the same straight path, statement after statement, could do very little. Real problems require decisions: if the marks are at least 50 the student passes, otherwise he fails; if the balance is sufficient the withdrawal is allowed, otherwise it is refused. C++ provides conditional statements for exactly this purpose. They test a condition — an expression that is either true or false — and execute different blocks of statements according to the result. This answer explains the forms of the if statement, the relational and logical operators used in conditions, and then gives the required program, which reads a number and reports whether it is positive, negative or zero: a problem with three outcomes, and therefore a natural example of the if – else if – else chain.

The Forms of the if Statement

The simplest form is the single if: if (condition) { statements } — the statements run only when the condition is true, and are skipped entirely when it is false. The if–else form adds an alternative block that runs when the condition is false, so exactly one of the two blocks always runs. For several mutually exclusive possibilities, the if – else if – else chain tests conditions in order from the top: the block belonging to the first true condition runs, all the rest are skipped,

and the final else, if present, catches every remaining case. A condition must be enclosed in parentheses, and a block of more than one statement must be enclosed in braces. Two other conditional tools complete the picture: the switch statement, which chooses among constant values of one integer or character expression, is neater than a long chain of equality tests; and the conditional operator ?: gives a small choice inside an expression, as in larger = (a > b) ? a : b;. For three ranges of a number, however, the else-if chain is the clearest and the appropriate choice here.

Table 1: Relational and logical operators used in conditions

Operator	Meaning	Example (result)
==	equal to	5 == 5 (true)
!=	not equal to	5 != 3 (true)
>	greater than	7 > 2 (true)
<	less than	2 < 7 (true)
>=	greater than or equal to	50 >= 50 (true)
<=	less than or equal to	40 <= 50 (true)
&&	AND — both sides true	marks >= 50 && attendance >= 75
||	OR — at least one side true	day == 'Sat' || day == 'Sun'
!	NOT — reverses true/false	!(age < 18)

USMAN NOTES www.usmannotes.com

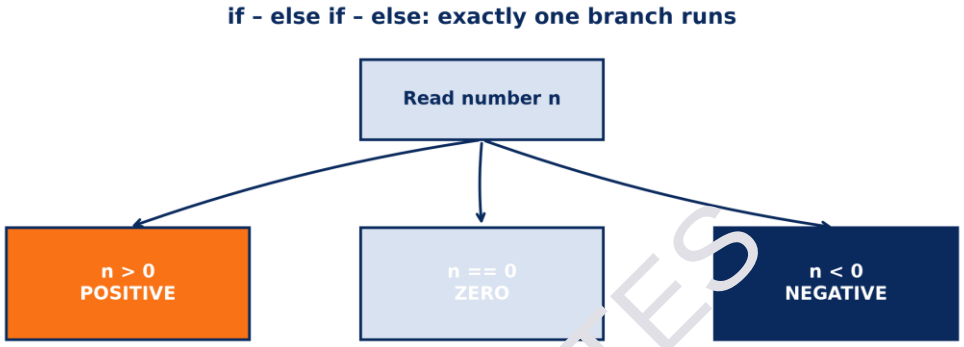


Figure 1: The three branches of the program — a number is positive, zero or negative, and exactly one branch is taken.

Program: Positive, Negative or Zero

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

int main()
{
    int number;

    cout << "Enter a number: ";
    cin >> number;

    if (number > 0)
    {
        cout << number << " is a POSITIVE number." << endl;
    }
    else if (number < 0)
```

```
    {
        cout << number << " is a NEGATIVE number." << endl;
    }
    else
    {
        cout << "The number is ZERO." << endl;
    }

    return 0;
}
```

USMAN NOTES www.usmannotes.com

Explanation of the Program

The program reads one integer into number and then works through the chain in a deliberately chosen order. The first test, number > 0, separates all the positive values. If it is true, the first message is printed and the rest of the chain is skipped completely — the computer does not even evaluate the later conditions. If it is false, the number is either negative or zero, and the second test, number < 0, separates the negative values. If that too is false, no test remains to be written: the number cannot be positive and cannot be negative, so it must be zero, and the final else prints that message. Notice the economy of the logic — only two tests are needed for three outcomes, because the last case is proved by elimination. Three sample runs confirm all branches: entering 8 prints that 8 is positive, entering -3 prints that -3 is negative, and entering 0 prints that the number is zero. Testing one value from every branch, plus the boundary value zero itself, is the minimum testing a conditional program should receive.

Common Mistakes with Conditions

- Writing = instead of == in a test: if (x = 5) assigns 5 to x and is treated as true, instead of comparing. Compilers often warn about this, and writing the constant first, if (5 == x), turns the mistake into an error that is caught.
- Forgetting braces: without them, only the very next statement belongs to the if, and following lines run unconditionally, which produces puzzling output.
- Placing a semicolon immediately after the condition, as in if (number > 0); — this makes the if control an empty statement, and the following block always runs.
- Overlapping or badly ordered conditions in a chain: because the first true condition wins, general conditions must not be placed before specific ones that they would hide.
- Testing floating-point values for exact equality: decimal values are stored approximately, so range tests such as value >= -0.0001 && value <= 0.0001 are safer than value == 0 for doubles. In this question an int is used, so exact testing for zero is perfectly correct.

Conditions are also frequently combined. A result program might test marks >= 50 && marks <= 100 before printing Pass, and nested if statements — an if inside another if — handle decisions that depend on an earlier decision, such as first checking that a number is positive and only then checking whether it is even. Nesting should be kept shallow, because deeply nested code is hard to read; an else-if chain or a switch is usually clearer when the choice depends on one value.

Conclusion

In conclusion, conditional statements give a program the power to decide. The if statement runs a block only when its condition is true, if–else chooses between two blocks, and the if – else if – else chain chooses exactly one block from several possibilities, with the final else covering all remaining cases. In the program given, two tests are enough to classify any integer as positive, negative or zero, because zero remains when the other possibilities are eliminated. Careful ordering of the tests, correct use of == rather than =, braces around every block, and testing with 8, -3 and 0 — one value for each branch — make the program both correct and demonstrably correct.

Question 5: Largest and Smallest Numbers in an Array of 15 Integers (20 Marks)

Question as printed: Write a C++ program to find the largest and smallest numbers in an array of 15 integers. (20)

Introduction

An array is a collection of values of the same type stored together under one name, each value being reached by its position, called an index. Arrays are the natural way to hold a list: the marks of a class, the temperatures of a month, or, as in this question, fifteen integers whose largest and smallest members must be found. Finding a maximum or a minimum is one of the most common tasks in computing, and it follows a pattern worth learning well: assume the first element is the answer, then examine every remaining element in turn, replacing the

assumption whenever a better one is found. This answer explains arrays and indexing briefly, gives the complete program, traces it, and notes the mistakes that most often make such programs fail.

Arrays and the Searching Pattern

The declaration int numbers[15]; creates fifteen int variables in a single block of memory, named numbers[0], numbers[1], and so on up to numbers[14]. C++ indexes arrays from zero, so the first element is at index 0 and the last at index size minus one; asking for numbers[15] would step outside the array into memory that belongs to something else, an error C++ does not automatically detect. Because the elements are numbered in a regular pattern, a for loop with a counter running from 0 to 14 visits each element exactly once — the same loop pattern used for input in Question 3. The searching pattern adds two variables: largest and smallest. Both must be initialised from a real element of the array, normally the first, numbers[0]. Initialising largest to zero is a classic fault: if all fifteen numbers happen to be negative, no element is greater than zero, and the program would wrongly report 0 as the largest value although 0 is not even in the array.

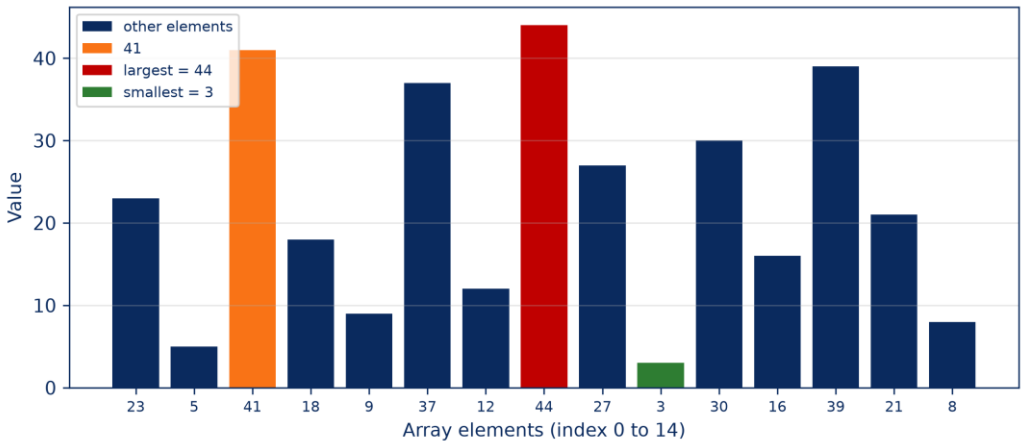


Figure 1: A sample array of 15 integers. Scanning it gives the largest value 44 and the smallest value 3.

Program: Largest and Smallest of 15 Integers

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


int main()
{
    int numbers[15];
    int largest, smallest;


    cout << "Enter 15 integers:" << endl;
    for (int i = 0; i < 15; i++)
    {
        cout << "Enter element " << i + 1 << ": ";
        cin >> numbers[i];
    }


    largest = numbers[0];
    smallest = numbers[0];


    for (int i = 1; i < 15; i++)
    {
        if (numbers[i] > largest)
            largest = numbers[i];
```

USMAN NOTES www.usmannotes.com

```
            if (numbers[i] < smallest)
                smallest = numbers[i];
        }

        cout << "\nLargest number  = " << largest << endl;
        cout << "Smallest number = " << smallest << endl;

        return 0;
    }
```

USMAN NOTES www.usmannotes.com

Explanation of the Program

The first loop fills the array: the counter i runs from 0 to 14, the prompt shows i + 1 so that the user sees friendly numbering from 1 to 15, and each value read by cin is stored at numbers[i]. Storing the values, rather than processing each and discarding it, is the point of using an array: the data remains available for the second pass and for any further processing. Before the second loop, both largest and smallest are set to numbers[0], the first genuine element. The second loop starts at i = 1, since index 0 has already been used as the starting assumption, and compares every remaining element twice: if it exceeds the current largest it becomes the new largest, and if it falls below the current smallest it becomes the new smallest. Both tests are independent if statements, not an else-if, because a value must be allowed to be tested against both — although in practice a single new value cannot be both the new largest and the new smallest unless all values are equal, using two plain ifs keeps the logic simple and always

correct. After the loop, the two variables hold the true extremes of the whole array and are displayed.

Table 1: Trace for the sample array 23, 5, 41, 18, 9, 37, 12, 44, 27, 3, 30, 16, 39, 21, 8

Element examined	Action taken	largest / smallest after it
23 (index 0)	Initial assumption	23 / 23
5	Smaller than smallest	23 / 5
41	Greater than largest	41 / 5
44	Greater than largest	44 / 5
3	Smaller than smallest	44 / 3
All other elements	Inside the range, no change	44 / 3
Final result	—	Largest = 44, Smallest = 3

USMAN NOTES www.usmannotes.com

Testing and Discussion

Four tests check this program thoroughly. First, the sample data above must give 44 and 3. Second, entering fifteen identical values, such as all 7s, must give largest and smallest both equal to 7, showing the program behaves sensibly when there is no variation. Third, entering only negative numbers, such as -5, -2, -9, tests the initialisation decision: the correct answers are -2 and -9, while a program that wrongly initialised largest to 0 would print 0 and expose its fault at once. Fourth, placing the extremes at the very ends — the largest as the first element and the smallest as the last — checks that no element, including the first and last, is missed by the loop bounds. The algorithm examines each element once, so it performs work proportional to the number of elements; for fifteen numbers this is trivial, but the same pattern remains

efficient for fifteen million. It is also worth noting that the two loops could be combined into one if the values were processed as they were read, but keeping input and processing in separate loops makes the program clearer and keeps the array available, which is better style at this level.

It is also worth recording the position, and not only the value, of an extreme when a later task requires it: alongside largest, a variable largestIndex can be updated in the same if statement whenever a new largest is found, so that after the loop the program can report both what the largest value is and where it occurred in the list. The same applies to the smallest value. This small extension shows the flexibility of the pattern — extra information that is discovered during the single scan can be kept with almost no extra work, provided the variables for it are declared and updated at exactly the moments when the extreme itself changes, and not elsewhere in the loop.

USMAN NOTES www.usmannotes.com

Conclusion

To conclude, the program declares an array of fifteen integers, fills it with a for loop whose counter runs over the indexes 0 to 14, and then finds the extremes by the assume-and-replace pattern: largest and smallest start from the first real element, and every later element is compared with both and replaces whichever it beats. The sample trace proves the method, ending with a largest of 44 and a smallest of 3, and the negative-numbers test shows why initialising from numbers[0], rather than from zero, is essential for a correct result with every possible set of input.