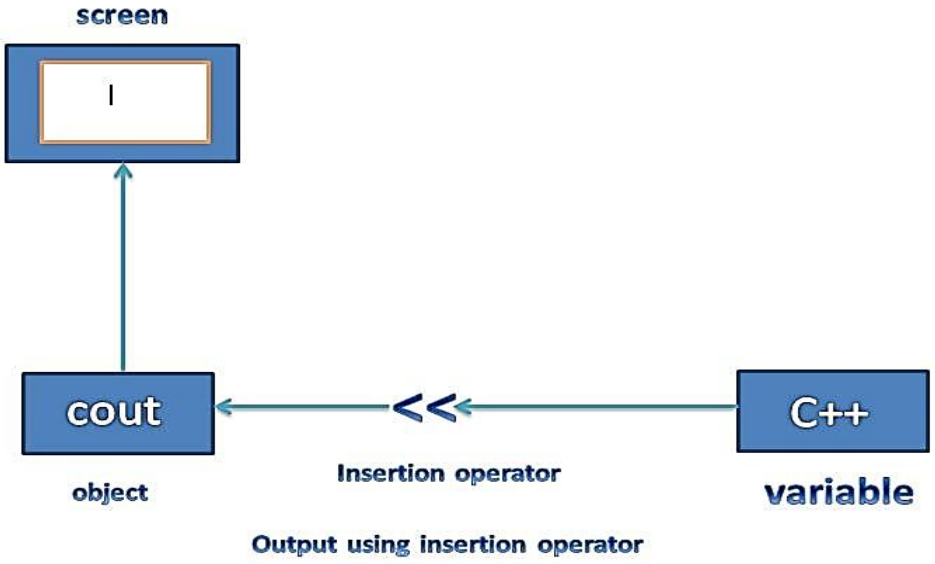
INTRODUCTION

UNIT 1

UNIT - I

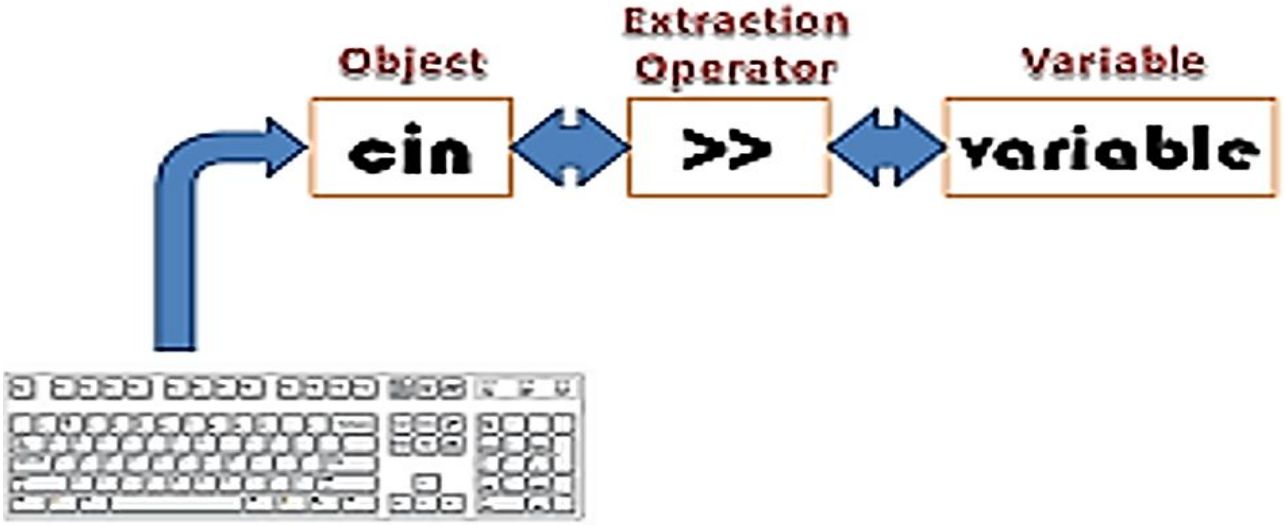
Introduction: Fundamentals of object oriented programming - procedure oriented programming vs. Object Oriented Programming (OOP), Characteristics of Object Oriented Programming. **C++ Programming Basics: Output using cout, directives, input with cin, type bool, Manipulators, type conversions.** Functions: Returning values from functions, reference arguments, overloaded functions, inline functions, default arguments, returning by reference

Input/output Operations



Program
↓
Memory (output buffer)
↓
Output stream (cout)
↓
Output device (monitor/screen)

Input/output Operations



Input/output Operations

Basic Input / Output in C++

In C++, data is read and written using streams, which are sequences of bytes.

Input stream: Data flows from a device (like the keyboard) to the computer's memory.

Output stream: Data flows from memory to an output device (like the screen).

These streams are defined in the <iostream> header file.

The most common stream objects are

cin : for taking input.

cout : for displaying output.

Standard Output Stream - cout

- cout is an instance of the ostream class used to display output on the screen.
- Data is sent to cout using the insertion operator <<.

Input/output Operations

Standard Input Stream - cin

- cin is an instance of the istream class used to read input from the keyboard.
- The extraction operator >> is used with cin to get data from the input stream and store it in a variable.

Operator	Name	Purpose
<<	Insertion operator	Sends data to output (cout)
>>	Extraction operator	Takes data from input (cin)

Input/output Operations

Cascading of I/O operators

- Cascading means using the same operator multiple times in one statement.
- Cascading of I/O operators in C++ is the technique of chaining multiple input (>>) or output (<<) operations in a single statement

```
#include <iostream>
int main()
 {
int age;
std::string name;
std::cout << "Enter your name and age: ";
std::cin >> name >> age;    //Cascading of Input operators
std::cout << "Hello, " << name << "! You are " << age << " years old." << std::endl; //Cascading of output
                                                                                        operators
 return 0;
}
```

Output Directives

cout → Writes to standard output stream (stdout).
cerr → Writes to standard error stream (stderr).
On the terminal, both appear on screen, but they are separate streams, so they can be redirected independently.
cout → For normal output or program results.
cerr → For errors, warnings, critical messages, or anything that should appear immediately.

Example:

```
#include <iostream>
using namespace std;
int main()
{
 cout << "Processing done successfully\n";   // normal output
  cerr << "Error: Invalid input detected\n";  // error output
  return 0;
}
```

Output Directives

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

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;

    if(age < 0) {
        cerr << "Error: Age cannot be negative\n";  // error
message
    } else {
        cout << "Your age is " << age << endl;      // normal
output
    }
    return 0;
}
```

Output Directives

Streams allow your program to read or write data without worrying about the exact device.

Examples:

cin → Used to take input from the keyboard

cout → output stream to screen (normal output)

cerr → output stream to screen (error messages)

cout → buffered → data is stored in memory first, then sent to the screen in chunks.

clog → buffered → used for logging messages. Keep a log of program execution, warnings, or debug messages.

cerr → unbuffered → data is sent immediately to the screen.

Output Directives

The C++ compiler mostly ignores whitespace such as spaces, tabs, and newlines. These characters are invisible to the compiler and do not affect program execution. Statements can be written on one line or spread across multiple lines, and the compiler treats them the same.

```
#include <iostream>
using
namespace std;
int main () { cout
<<
“Every age has a language of its own\n”
; return
0;}
```

Output:
Every age has a language of its own

```
#include <iostream>
using
namespace std;
int main () { cout
<<
“Every age has a language
 of its own\n”
; return
0;}
```

Output:
error: missing terminating " character

Output Directives

```
#include <iostream>
using namespace std;
int main () {
    cout<<“Every age has a
    language of its own\n”;
    return 0;
}    //ERROR
```

```
include <iostream>
using namespace std;
int main () {
    cout<<“Every age has a\
    language of its own\n”;
    return 0;}
//Every age has a    language of its
own
```

```
#include <iostream>
using namespace std;
int main() {
    cout << "Every age has a "
            "language of its own\n";
    return 0;
}
//Every age has a    language of its
own
```

In C++, string constants cannot be split across multiple lines because the compiler expects the string to end on the same line where it starts.
If a string is long, you can continue it by placing a backslash (\) at the line break, or by dividing it into two separate quoted strings, which the compiler automatically joins.

C++ Comments

Comment lines in C++Comments are used to explain code and are ignored by the compiler. They make programs easier to read and understand.

```
// This is a single-line comment

/* This is a
   multi-line comment */
```

Output Directives

A directive is an instruction to the compiler or preprocessor that tells it to perform some specific action before compilation.
The two lines that begin the FIRST program are directives. The first is a preprocessor directive, and the second is a using directive.

Preprocessor directives are instructions given to the preprocessor, which runs before compilation.
They are used to:

- Include files (#include)
- Define constants/macros (#define)
- Control compilation conditionally (#ifdef, #ifndef, #if, #endif)

include - Includes header files (library or user-defined).

```
#include <iostream>   // Standard library
#include "myfile.h"    // User-defined header
```

Output Directives

- <iostream> is a preprocessor directive that tells the preprocessor to copy the contents of the standard header <iostream> into source file before compilation, so that declarations for standard input/output stream objects like std::cin, std::cout, and std::cerr become available.

Output Directives

using directive is used to avoid typing full namespace names repeatedly.
For example, std::cout belongs to the std namespace.

- using namespace std; is a using directive that imports all names from the std namespace into the current scope, allowing you to write cout and cin instead of std::cout and std::cin

```
std::cout << "Hello" << std::endl;
cout << "Hello" << endl;
```

A namespace is a container for identifiers (like variables, functions, classes) that prevents naming conflicts. Standard C++ library uses the namespace std for all its functions and objects.

Output Directives

- #define is a preprocessor directive. Define constants or macros.
- It is used to define constants or macros before compilation. The preprocessor replaces every occurrence of the name with its value before the program is compiled.

```
#define PI 3.14159
#define SQUARE(x) ((x)*(x))
#undef PI   //Undefines a previously defined macro.
```

```
#include <iostream>
using namespace std;
#define PI 3.14159  // define constant PI
int main() {
    float radius = 5.0;
    float area = PI * radius * radius;
   // PI replaced by 3.14159
    cout << "Area: " << area << endl;
    return 0;
}
```

Output Directives

```
#include <iostream>
using namespace std;
// Define a macro
#define MAX 100
int main() {
    cout << "MAX before undef: " << MAX << endl;  // Prints 100

     #undef MAX  // Undefine the macro

     #define MAX 50
    cout << "MAX after undef: " << MAX << endl;   // Prints 50

    return 0;
}
```

Directives

```
#include <iostream>
#include <cmath>
#include <string>
#include <iomanip>
```

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

int main() {
    cout << sqrt(25) << endl;
    cout << pow(2, 4) << endl;
}
```

Output:
5
16

Manipulators

A manipulator in C++ is a special function that is used with input/output streams (cin, cout) to format the data.
It modifies how data is displayed or read, without changing the actual value.

Manipulators

endl
Inserts a newline and flushes the output buffer.
cout << "Hello" << endl;

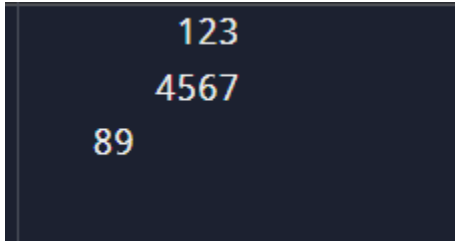
Flush
Flushes the output buffer without newline.
cout << "Data" << flush;

setw(n)
Sets the width of the next output field.
cout << setw(10) << 123;

Sets the width of the next output field for cout or cin.
The next output will take up at least n spaces.
If the content is shorter than n, it is padded with spaces.
If the content is longer than n, the field expands automatically.

Manipulators

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

int main() {
    cout << setw(10) << 123 << endl;   // right-aligned by default
    cout << setw(10) << 4567 << endl;
    cout << setw(5) << 89 << endl;
    return 0;
}
```



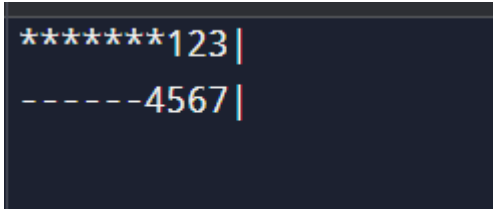
Manipulators

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

int main() {
    cout << left << setw(10) << 123 << "|" << endl;
    cout << left << setw(10) << 4567 << "|" << endl;
    cout << left << setw(10) << 89 << "|" << endl;
    return 0;
}
```

```
123       |
4567      |
89        |
```

Manipulators

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

int main() {
    cout << setfill('*') << setw(10) << 123 << "|" << endl;
    cout << setfill('-') << setw(10) << 4567 << "|" << endl;
    return 0;
}
```



Manipulators

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

int main() {
cout << left;
    cout << setfill('*') << setw(10) << 123 << "|" << endl;

cout << right;
    cout << setfill('-') << setw(10) << 4567 << "|" << endl;

    return 0;
}
```

```
123*******|
------4567|
```

Manipulators

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

int main() {
    char name[10];
    cout << "Enter your name : ";
    cin >> setw(5) >> name;
    cout << "You entered: " << name << endl;

    return 0;
}
```

```
Enter your name : welcome to sastra
You entered: welc
```

Manipulators

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

int main() {
    string name;
    cout << "Enter your name : ";
    cin >> setw(5) >> name;
    cout << "You entered: " << name << endl;

    return 0;
}
```

```
Enter your name : welcome to sastra
You entered: welco
```

Manipulators

left and right (from <iomanip>)
Align output in a field.

```
cout << left << setw(10) << "Hi";  // left-align
cout << right << setw(10) << "Hi"; // right-align
```

showpoint / noshowpoint (from <iomanip>)
Forces or removes decimal point.

```
 cout << showpoint << 3.0<<endl; // 3.00000
cout << noshowpoint<<3.0000;   //3
```

hex / dec / oct (from <iostream>)
Print integer in hexadecimal, decimal, or octal.

```
cout << hex << 255; // ff
cout << dec << 255; // 255
cout << oct << 255; // 377
```

Manipulators

showbase / noshowbase
Show or hide numeric base prefix (0x, 0).

```
cout <<showbase << hex << 255<<endl;
cout <<showbase << dec << 255<<endl;
cout << showbase <<oct << 255<<endl;
```

0xff
255
0377

boolalpha / noboolalpha
Prints boolean values as true/false or 1/0.
Int a=true;

```
cout << boolalpha <<a;  // true
cout << noboolalpha << a; // 1
```

Manipulators

fixed – Fixed-point notation
Shows floating-point numbers in normal decimal form, not scientific.
Works with setprecision(n) to control decimal places.

```
double x = 123.456789;
cout << fixed << setprecision(2) << x << endl; // 123.46
```

```
double x = 123.456789;
cout << scientific << setprecision(2) << x << endl;  //1.23e+02
```

Manipulators

```
#include <iostream>
#include<iomanip>
using namespace std;
int main() {
float a=1234.1234567812;
 float b=12.45678;
 cout<<a<<endl;
 cout<<setprecision(4)<<a<<endl;
cout<<setprecision(2)<<a<<endl;
cout<<fixed<<setprecision(2)<<a<<endl;
cout<<setprecision(3)<<a<<endl;
cout<<a<<endl;
 cout<<defaultfloat<<a<<' ' <<b<<endl;
cout<<setprecision(6)<<a<<endl;
  return 0;}
```

```
1234.12
1234
1.2e+03
1234.12
1234.123
1234.123
1.23e+03 12.5
1234.12
```

Manipulators

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 12.34567;
    int a = -25;
  int c = 40;
   cout << fixed << setprecision(2) << x << endl;
    cout << internal << setw(6) << a << endl;
    cout << hex << 255 << endl;
    cout << uppercase <<  255 << endl;
cout<<254<<endl;
cout<<nouppercase<<hex<<255<<endl;
cout<<showpos<<c<<endl;
cout<<dec<<showpos<<c;
 return 0;
}
```

```
12.35
-   25
ff
FF
FE
ff
28
+40
```

Manipulators

```
#include <iostream>
using namespace std;
int main()
{
int age;
 string name;
cin>>name;
cout<<name;
return 0;
}  // hello world  hello
```

```
#include <iostream>
using namespace std;
int main()
{
 int age;
 string name;
getline(cin,name);
cout<<name;
return 0;}  //hello world
```

```
#include <iostream>
using namespace std;
int main()
{
 int age;
 string name;
cin>>age;
getline(cin,name);
cout<<name;
cout<<age
return 0;}  //12   12
```

```
#include <iostream>
using namespace std;
int main()
{
 int age;
 string name;
cin>>age;
cin>>ws;
getline(cin,name);
cout<<name<<endl;
cout<<age
return 0;}  //hello world 12
```

Manipulators

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    bool b;
    int x;
    string name;
    // Input boolean as true/false
    cout << "Enter true or false: ";
    cin >> boolalpha >> b;
    // Input hexadecimal number
    cout << "Enter hex number (like FF): ";
    cin >> hex >> x;
    // Input full name (with spaces)
    cout << "Enter your name: ";
    getline(cin >> ws, name);

    // Output
    cout << "\nResults:\n";
    cout << "Boolean: " << boolalpha << b << endl;
    cout << "Number (Decimal): " << dec << x << endl;
    cout << "Name: " << name << endl;

    return 0;
}
```

Basic

Constant

1. **const** keyword creates a typed constant variable whose value cannot be changed.

```
const float PI = 3.14159;  // constant float
const int MAX = 100;       // constant integer
```

2. **#define** directive creates a macro that acts as a constant before compilation.

```
#include <iostream>
using namespace std;
#define PI 3.14159
#define MAX 100
int main() {
    cout << "PI = " << PI << endl;
    cout << "MAX = " << MAX << endl;
    return 0;}
```

Bool

```
#include <iostream>
#include <iomanip>  // for boolalpha
using namespace std;

int main() {
    int alpha = 5;
    int beta = 10;

    bool result = (alpha < beta);
    cout << "Is alpha less than beta? " << result << endl;

    cout << boolalpha;  // enables printing bool as true/false
    cout << "Is alpha less than beta? " << result << endl;

    cout << noboolalpha << "Is alpha less than beta? " << result << endl;

    return 0;
}
```

```
Is alpha less than beta? 1
Is alpha less than beta? True
Is alpha less than beta? 1
```

Compound Assignment Operators

```
#include <iostream>
using namespace std;
int main()
{
    int ans = 27;
    ans += 10;  // same as: ans = ans + 10;
    cout << ans << ", ";
    ans -= 7;   // same as: ans = ans - 7;
    cout << ans << ", ";
    ans *= 2;   // same as: ans = ans * 2;
    cout << ans << ", ";
    ans /= 3;   // same as: ans = ans / 3;
    cout << ans << ", ";
    ans %= 3;   // same as: ans = ans % 3;
    cout << ans << endl;

    return 0;}
```

Arithmetic Operators


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

int main() {
    int ans = 20;

    cout << ans + 10 << ", "
                << ans - 7 << ", "
                << ans * 2 << ", "
                << ans / 3 << ", "
                << ans % 3 << endl;
    return 0;
}
```

Increment Operators

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

int main()
{
    int count = 10;

    cout << "count = " << count << endl;     // displays 10
    cout << "count = " << ++count << endl;   // displays 11 (prefix increment)
    cout << "count = " << count << endl;     // displays 11
    cout << "count = " << count++ << endl;   // displays 11 (postfix increment)
    cout << "count = " << count << endl;     // displays 12

    return 0;
}
```

Prefix ++count: increment happens first, then the value is used.
Postfix count++: value is used first, then increment happens.

Decrement Operators


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

int main() {
    int count = 10;

    cout << "count = " << count << endl;
cout << "count = " << --count << endl;
cout << "count = " << count << endl;
    cout << "count = " << count-- << endl;
cout << "count = " << count << endl;
    return 0;
}
```

Prefix --count: Decrement happens first, then the value is used.
Postfix count--: value is used first, then Decrement happens.

Escape Sequence

An escape sequence is a sequence of characters that starts with a backslash (\) and is used to represent special characters such as newline, tab, backslash, etc., inside strings and character constants.

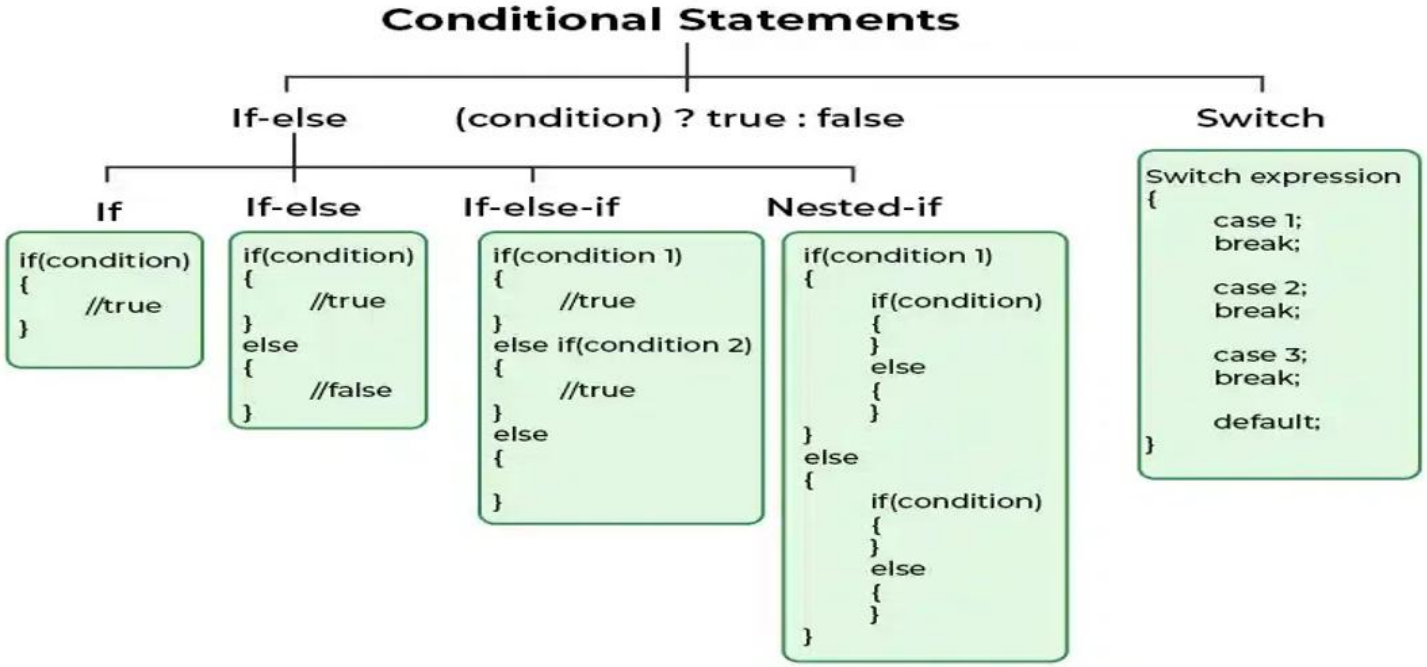
Escape Sequence

Escape Sequence	Character / Meaning
`\a`	Bell (beep sound)
`\b`	Backspace
`\f`	Form feed
`\n`	Newline (moves cursor to next line)
`\r`	Carriage return
`\t`	Horizontal tab
`\\`	Backslash (\ )
`\'`	Single quotation mark (')
`\"`	Double quotation mark (")
`\xdd`	Character represented by hexadecimal value `dd`

Escape Sequence

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Bell\a";               // Bell sound (beep) Produces a beep sound
    cout << "\nBack\bspace";         // Backspace Moves the cursor back Deletes previous character
    cout << "\nForm\fFeed";          // Form feed usually extra spacing; page break in old printers
    cout << "\nNew\nLine";           // New line Moves to new line
    cout << "\rReturn";              // Carriage return  Moves cursor to start of line
    cout << "\nTab\tSpace";           // Tab space Horizontal tab
    cout << "\nBackslash \\";         // Backslash Prints backslash
    cout << "\nSingle Quote \'A\'";    // Print Single quotation mark
    cout << "\nDouble Quote \"A\"";    // Print Double quotation mark
    cout << "\nHexadecimal \x41";      // Hex value (41 = 'A')
    cout << "\n octal \101“;    // octal A
    return 0;
}
```

Decision making statements



Decision making statements

Ternary Operator

Syntaxcondition ? expression1 : expression2;

If the condition is true → expression1 is executed

If the condition is false → expression2 is executed

Example

```
#include<iostream>
using namespace std;
int main()
{
int a = 10, b = 20;
int max = (a > b) ? a : b;
cout << max;
return 0;
}
```

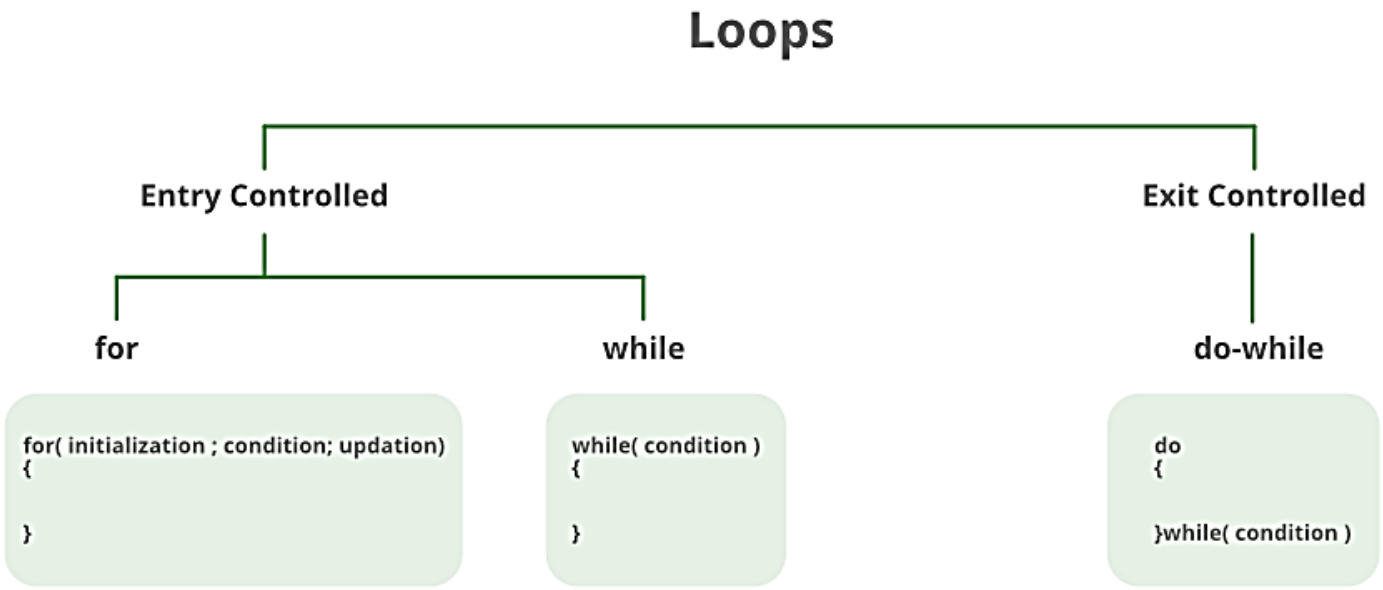
Output:20

Decision making statements

condition1 ? result1 : condition2 ? result2 : result3;

```
#include <iostream>
using namespace std;
int main() {
    int a, b, c, biggest;

    cout << "Enter three numbers: ";
    cin >> a >> b >> c;

    biggest = (a > b)
              ? ((a > c) ? a : c)
              : ((b > c) ? b : c);

    cout << "Biggest number is: " << biggest;
    return 0;
}
```

Iterative statements



Iterative statements

for

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

int main() {
    cout << "Using for loop:" <<
endl;
    for (int i = 1; i <= 5; i++)
{
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

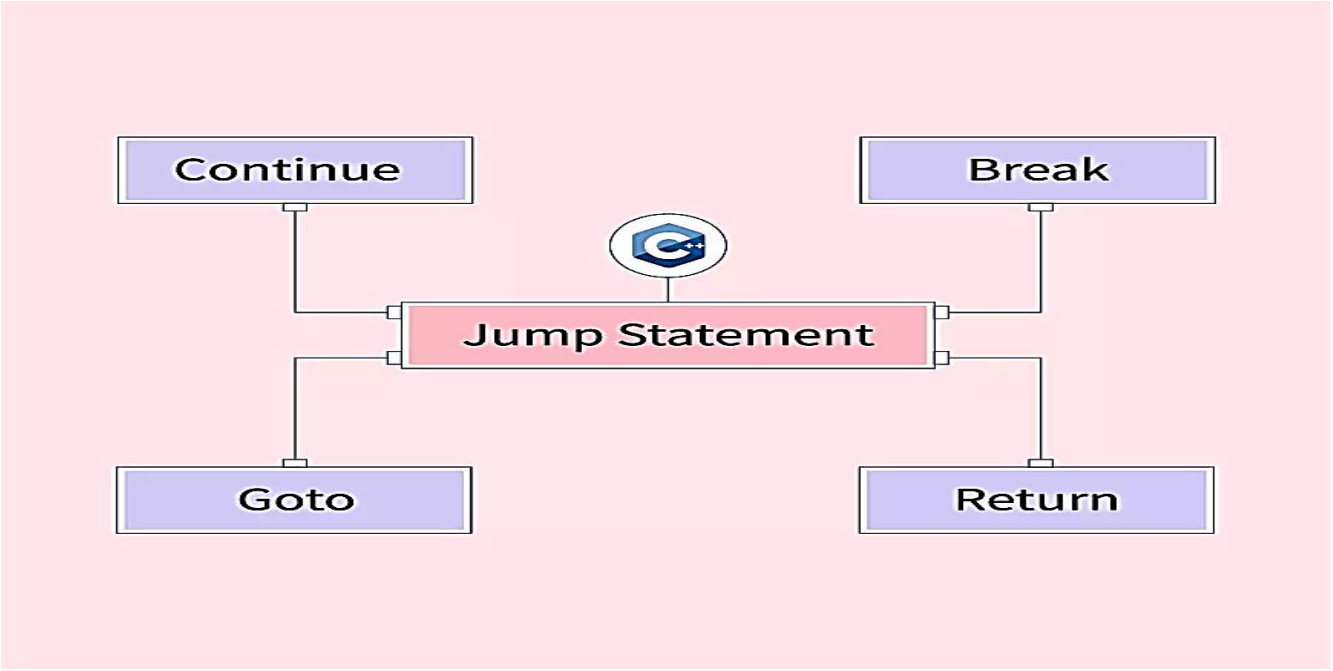
while

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

int main() {
    cout << "Using while loop:"
<< endl;
    int i = 1;
    while (i <= 5) {
        cout << i << " ";
        i++;
    }
    cout << endl;
    return 0;
}
```

do while

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

int main() {
    cout << "Using do-while loop:"
<< endl;
    int i = 1;
    do {
        cout << i << " ";
        i++;
    } while (i <= 5);
    cout << endl;
    return 0;
}
```

Jumping statements



Jumping statements

continue → skips the rest of the current loop iteration and goes to the next iteration
break → exits the loop or switch immediately
goto → jumps to a labeled statement anywhere in the function
return → exits from the function immediately

Jumping statements

Program prints numbers from 1 to 10 but skips even numbers using continue.

```
#include <iostream>
using namespace std;
int main()
{
 for (int i = 1; i <= 10; i++)
 {
    if (i % 2 == 0)
{           continue; // skip even numbers
     }
 cout << i << " "; // only odd numbers will be printed
  }
  cout << endl;
return 0;
}
```

Output:1 3 5 7 9

Jumping statements

Program prints numbers from 1 to 10 but stops when it reaches 5 using break.

```
#include <iostream>
using namespace std;
int main()
{    for (int i = 1; i <= 10; i++)
 {
  if (i == 5)
 {
break; // stop the loop
   }
 cout << i << " ";
 }
cout << endl;
    return 0;
}
```

Output:

1 2 3 4

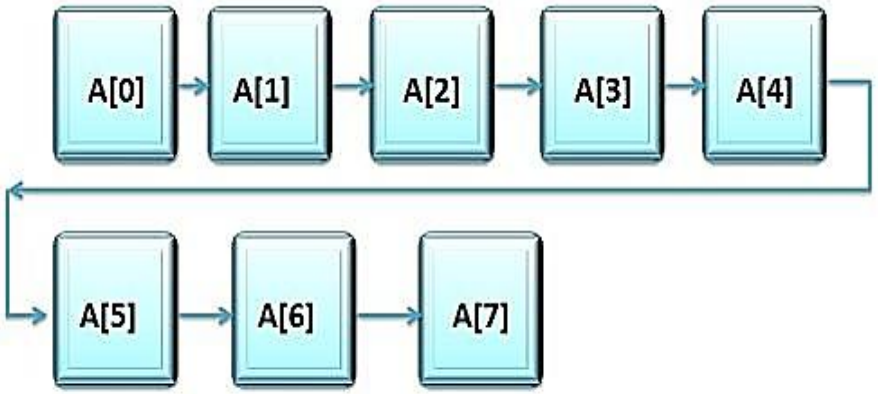
Jumping statements

This program prints numbers from 1 to 5 using a label and goto.

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

int main() {
    int i = 1;
start:      // label
    if (i > 5) {
        goto end; // exit the loop
    }
    cout << i << " ";
    i++;
    goto start; // jump back to start
end:
    cout << endl;
    return 0;
}
```

Output:
1 2 3 4 5

Arrays



One-Dimensional Array A[8]

	Column 0	Column 1	Column 2
Row 0	X[0][0]	X[0][1]	X[0][2]
Row 1	X[1][0]	X[1][1]	X[1][2]
Row 2	X[2][0]	X[2][1]	X[2][2]

Arrays


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

int main() {
    // 1D arrays declaration and initialization
    int arrInt[5] = {1, 2, 3, 4, 5};
    float arrFloat[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
    char arrChar[5] = {'a', 'b', 'c', 'd', 'e'};

    // Print integer array
    cout << "Integer array: ";
    for (int i = 0; i < 5; i++) {
        cout << arrInt[i] << " ";
    }
    cout << endl;

    // Print float array
    cout << "Float array: ";
    for (int i = 0; i < 5; i++) {
        cout << arrFloat[i] << " ";
    }
    cout << endl;

    // Print char array
    cout << "Char array: ";
    for (int i = 0; i < 5; i++) {
        cout << arrChar[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Arrays


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

int main() {
    int arr[2][3] = { {1, 2, 3}, {4, 5, 6} }; // 2 rows, 3
columns

    cout << "2D int array:" << endl;
    for (int i = 0; i < 2; i++) {          // loop for rows
        for (int j = 0; j < 3; j++) {      // loop for columns
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

```
2D int array:
1 2 3
4 5 6
```

Arrays

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

int main() {
     char names[2][11] = { "Alice", "Bob" };

    cout << "Names in the 2D char array:" << endl;
    for (int i = 0; i < 2; i++) {
        cout << names[i] << endl;
    }

    return 0;
}
```

```
Names in the 2D char array:
Alice
Bob
```

Recursion


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

int fact(int n)
{
    if(n == 0)        // Base case
        return 1;
    else
        return n * fact(n-1);   // Recursive call
}

int main()
{
    int n = 4;
    cout << fact(n);
}
```

fact(4)
= 4 * fact(3)
fact(3)
= 3 * fact(2)
fact(2)
= 2 * fact(1)
fact(1)
= 1 * fact(0)
fact(0)
= 1 ← Base case reached

fact(1) = 1 * 1 = 1
fact(2) = 2 * 1 = 2
fact(3) = 3 * 2 = 6
fact(4) = 4 * 6 = 24

Recursion


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

int fib(int n)
{
    if(n == 0)        // Base case
        return 0;
    else if(n == 1)   // Base case
        return 1;
    else
        return fib(n-1) + fib(n-2);  // Recursive calls
}

int main()
{
    int n = 5;
    cout << fib(n);
}
```

fib(5)
= fib(4) + fib(3)

fib(4)
= fib(3) + fib(2)

fib(3)
= fib(2) + fib(1)

fib(2)
= fib(1) + fib(0)

fib(2) = 1 + 0 = 1
fib(3) = 1 + 1 = 2
fib(4) = 2 + 1 = 3
fib(5) = 3 + 2 = 5

Type Conversion

Implicit Type Conversion (Automatic Conversion)
Explicit Type Conversion (Type Casting)

Implicit Type Conversion (Automatic)

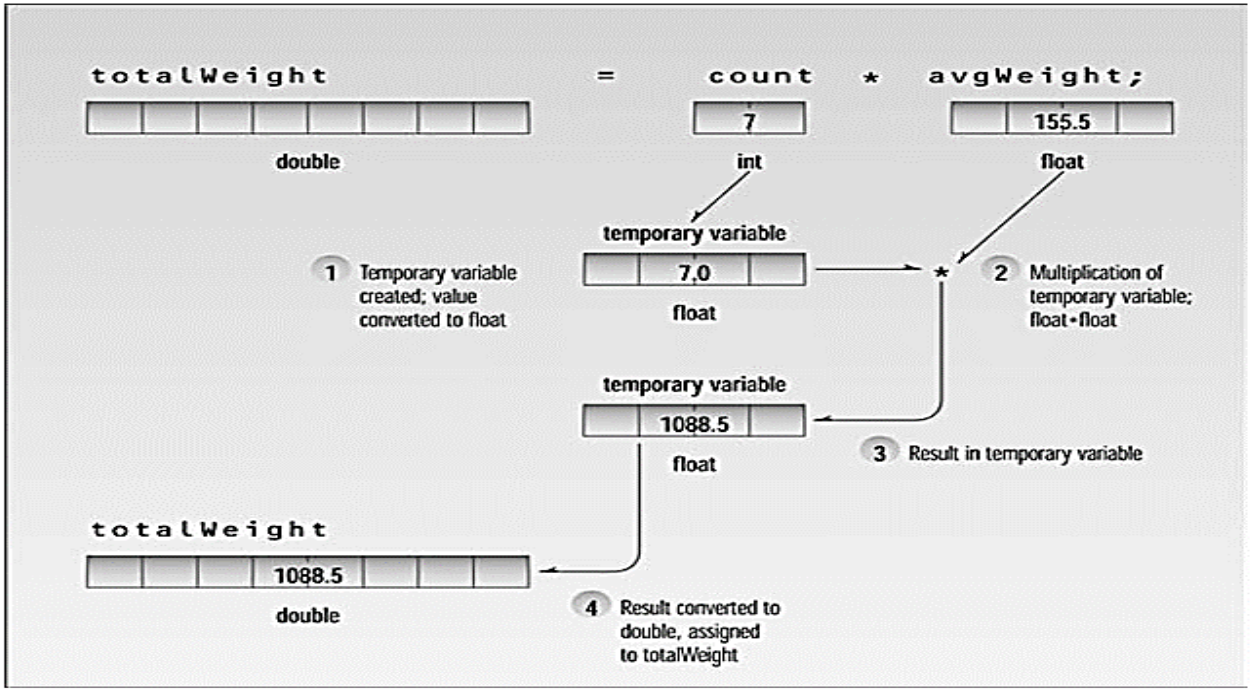
- Done automatically by the compiler.
- Usually happens when different types are used in expressions.
- The smaller type is converted to a larger type automatically.

Type Conversion

TABLE 2.4 Order of Data Types

Data Type	*Order*
`long double`	Highest
`double`	
`float`	
`long`	
`int`	
`short`	
`char`	Lowest

Type Conversion



Type Conversion


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

int main() {
    int a = 10;
    double b = 3.5;
    double result;

    result = a + b;  // int a is automatically converted to double

    cout << "Result: " << result << endl;
    return 0;
}
```

Type Casting

Explicit Type Conversion (Type Casting)
Done **manually by the programmer** using **type cast operators**.
You can convert data types **anywhere you want**.

```
int a = 10;
double b = 3.5;
int sum;

sum = a + (int)b;  // convert b to int
cout << sum << endl;  // Output: 13
```

Type Casting

Before Standard C++, casts were handled using quite a different format.

Instead of aCharVar = static_cast<char>(anIntVar);

you could say ,

aCharVar = (char)anIntVar;

or alternatively

aCharVar = char(anIntVar)

Type Casting


```
#include <iostream>
using namespace std;
int main() {
    int anIntVar = 65;   // ASCII value of 'A'
    char aCharVar1, aCharVar2, aCharVar3;
    // 1. Old C-style cast
    aCharVar1 = (char)anIntVar;
    // 2. Function-style cast
    aCharVar2 = char(anIntVar);
    // 3. Modern C++ cast (Recommended)
    aCharVar3 = static_cast<char>(anIntVar);
    cout << "C-style cast      : " << aCharVar1 << endl;
    cout << "Function-style    : " << aCharVar2 << endl;
    cout << "static_cast style : " << aCharVar3 << endl;
    return 0;
}
```

```
C-style cast      : A
Function-style    : A
static_cast style : A
```

Type Casting

static_cast<type>(value) – Normal type conversion. Compile time conversion. It converts one data type into another.
dynamic_cast<type>(value) → for classes and inheritance. Run time.
const_cast<type>(value) → remove const qualifier from a variable without changing its data type.
reinterpret_cast<type>(value) → low-level type reinterpretation. It is used to convert one pointer or data type into another by reinterpreting the underlying memory representation.

```
double b = 3.5;
int a = static_cast<int>(b);  // converts 3.5 to 3
cout << a << endl;
```

Type Casting


```
#include <iostream>
using namespace std;
int main() {
    int a = 7, b = 2;
    double result1, result2;
    // Implicit conversion
    result1 = a / b;
    cout << "Implicit result: " << result1 << endl;

    // Explicit conversion
    result2 = static_cast<double>(a) / b; // convert a to double
    cout << "Explicit result: " << result2 << endl;

    return 0;
}
```

Scope and Stroage Class

The scope of a variable deter mines which parts of the program can access it, and its storage class determines how long it stays in existence.

- Variables with local scope are visible only within a block.
- Variables with file scope are visible throughout a file.

A block is basically the code between an opening brace and a closing brace. Thus a function body is a block.

There are two storage classes: automatic and static.

• Variables with storage class automatic exist during the lifetime of the function in which they’re defined.

• Variables with storage class static exist for the lifetime of the program

Scope and Stroage Class

TABLE 5.2 Storage Types

	Local	*Static Local*	*Global*
Visibility	function	function	file
Lifetime	function	program	program
Initialized value	not initialized	0	0
Storage	stack	heap	heap
Purpose	Variables used by a single function	Same as local, but retains value when function terminates	Variables used by several functions

Scope and Stroage Class


```
#include <iostream>
using namespace std;
// Global variable
int g = 10;

void demo() {
    // Local variable
    int l = 5;
    // Static local variable
    static int s = 0;
    l = l + 1;   // New every time
    s = s + 1;   // Keeps value
    cout << "Local l = " << l << endl;
    cout << "Static s = " << s << endl;
}

int main() {
    cout << "Global g = " << g << endl << endl;

    demo();
    cout << endl;

    demo();
    cout << endl;

    demo();

    return 0;
}
```

Scope and Stroage Class


```
#include <iostream>
using namespace std;
// Global variable
int g = 10;

void demo() {
    // Local variable
    int l = 5;
    // Static local variable
    static int s = 0;
    l = l + 1;   // New every time
    s = s + 1;   // Keeps value
    cout << "Local l = " << l << endl;
    cout << "Static s = " << s << endl;
}

int main() {
    cout << "Global g = " << g << endl << endl;

    demo();
    cout << endl;

    demo();
    cout << endl;

    demo();

    return 0;
}
```

Scope and Stroage Class

Global g = 10

Local l = 6
Static s = 1

Local l = 6
Static s = 2

Local l = 6
Static s = 3

Scope resolution operator to define member function outside the class


```
#include <iostream>
using namespace std;
// Class declaration
class Student {
private:
    int roll;
public:
    void setRoll(int r);   // Function declaration
    void showRoll();       // Function declaration
};

// Function definitions using scope resolution operator
void Student::setRoll(int r) {
    roll = r;
}
void Student::showRoll() {
    cout << "Roll Number = " << roll << endl;
}

int main() {
    // Create object
    Student s1;

    // Access functions using object
    s1.setRoll(101);
    s1.showRoll();

    return 0;
}
```

exit()


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

int main() {
    cout << "Before exit\n";
    exit(0);   // Program stops here
    cout << "After exit\n"; // This will NOT run
}  // output before exit
```