Modern Programming Principles & Practice

Feb-May 2025
sruthi.padathara@dorset.ie

Object Oriented Programming Principles & Introduction C++

Contents

- Need of OOPs, difference b/w structured & OOPs
- What are objects, features
- OOPs Features
- Classes & objects
- Encapsulation
- Inheritance
- Polymorphism
- Data Abstraction
- History, Features
- Rules of C++ programming
- Structure of C++ program
- C++ Tokens(Identifiers, Keywords, Constants, Operators, Special characters)
- C++ Data types(Basic, Derived, User defined)
- Installation IDE VSCode
- How to Write and Run Your First Program in C++

Overview of **Programming Paradigms**

Two main approaches:

Structured Programming: A programming paradigm that follows a **top-down** approach, dividing programs into smaller functions for easy execution.

```
Start ATM_Withdrawal
    Authenticate_User()
    IF authentication_success THEN
        Check_Balance()
        IF sufficient_balance THEN
            Dispense_Cash()
            Print_Receipt()
        ELSE
            Display "Insufficient Balance"
        ENDIF
    ELSE
        Display "Authentication Failed"
    ENDIF
End ATM_Withdrawal
```

Object-Oriented Programming (OOP): A programming paradigm that organizes code using **objects** that contain **data (attributes)** and **functions (methods)**.

```
Class User
    Attributes: account_number, pin, balance
    Methods: authenticate(), check_balance(), withdraw_amount()

Class ATM
    Method: start()
        user = Authenticate_User()
        IF user is authenticated THEN
            IF user.check_balance(requested_amount) THEN
                user.withdraw_amount(requested_amount)
                Print_Receipt()
            ELSE
                Display "Insufficient Balance"
            ENDIF
        ELSE
            Display "Authentication Failed"
        ENDIF

    Method: Print_Receipt()
        Display "Transaction Successful"
        Display "Remaining Balance: ", user.balance

Start ATM
```

Structured Programming

- Uses **functions and procedures**
- Focuses on **logic and process flow**
- **Sequential execution** (Step-by-step)
- **Less flexible** for complex applications
- **Data is global** and shared among functions
- **Example:** C, Pascal, Fortran

Object-Oriented Programming (OOP)

- **Encapsulation** (Data hiding for security)
- **Inheritance** (Code reuse and extension)
- **Polymorphism** (One interface, multiple implementations)
- **Abstraction** (Hides complex details)
- Uses a **bottom-up** approach
- **Example:** Java, Python, C++, C#

Need of OOPs

Why do we need object-oriented programming

Object-oriented programming (OOP) is essential because it helps in structuring code in a way that enhances readability, maintainability, and scalability.

Easier Development & Maintenance – OOP organizes code into objects, making large projects more manageable and reducing complexity.
Data Hiding & Security – Encapsulation allows restricting direct access to certain parts of the code, preventing accidental modifications and improving security.
Real-World Problem Solving – OOP models real-world entities using classes and objects, making software development more intuitive.
Code Reusability – Inheritance allows existing code to be reused, reducing redundancy and effort in writing code from scratch.
Generic Code & Flexibility – Polymorphism enables writing generic code that works with different data types, reducing duplication and improving efficiency.

Difference between Structured Programming & OOPs

Feature	Structured Programming	Object-Oriented Programming (OOPs)
Definition	A type of procedural programming that focuses on a sequence of instructions.	Consists of objects that have properties (data) and methods (functions).
Program Structure	A program consists of small functions and procedures.	A program is built using objects and entities.
Code Readability & Reusability	Code is **readable**, and some components can be reused.	Objects are created, and each object contains multiple functions and data.
Code Quality	Programs are clear and maintain high quality.	Aims to make development **easier and more productive**.
Focus	Focuses on **functions and processes** to manipulate data.	Focuses on **objects and data**, with methods defining their behavior.
Modularity	Divides a program into functions, making it easier to modify and manage.	Divides a system into **small modules (objects)**, combining data and processes.
Execution Flow	Code executes **sequentially** from top to bottom.	Methods work **dynamically** and are called as needed.
Approach	Uses a **top-down approach** (breaks a large problem into smaller parts).	Uses a **bottom-up approach** (starts with small objects and builds a system).
Flexibility	**Less flexible**, as modifying one function may require changes in other parts of the program.	**More flexible**, as objects operate independently and can be modified easily.
Data Security	**Less secure**, as data is often stored globally.	**More secure**, as data is encapsulated within objects.
Code Importance	Focuses more on **code and logic**.	Focuses more on **data and its manipulation**.
Functionality	The **main function** calls other functions for processing.	Objects communicate with each other and **pass messages** to perform actions.

OOPs Features

- Classes
- Objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Classes

A **class** is a blueprint or template for creating objects in **Object-Oriented Programming (OOP)**. It defines **attributes (data)** and **methods (functions)** that an object will have.

```
Class User
   Attributes:
      account_number
      pin
      balance
   Methods:
      authenticate()
      check_balance()
      withdraw_amount()
```

A **User class** defines common properties for all users.
When we create a **new user object**, it will have its own **account number, PIN, and balance**.

Objects: Instances of a Class

```
user1 = User() // Creating an object
user1.account_number = 12345
user1.balance = 5000
```

What are objects, features

An **object** is a fundamental building block in **object-oriented programming (OOP)**. It is an instance of a **class**, which acts as a blueprint for creating objects.

Key Features of Objects

1. **State (Attributes or Properties)**
   - An object has attributes that represent its **state** or **data**.
   - These attributes are defined by **variables** inside the object.
   - **Example:** In an **E-commerce Product object**, attributes might include `productName`, `price`, and `stockQuantity`.
2. **Behavior (Methods or Functions)**
   - Objects have **methods** (or functions) that define their **behavior** or **actions**.
   - Methods allow the object to **perform operations** on its own data or interact with other objects.
   - **Example:** A **Product object** might have a method called `applyDiscount()` to reduce its price.
3. **Identity**
   - Every object has a unique **identity** that distinguishes it from other objects. This identity is defined when the object is created.
   - **Example:** Two different **Product objects** (e.g., one for a laptop and one for a phone) might have the same attributes, but each will have a unique identity in memory.

Encapsulation

Encapsulation is the **process of restricting direct access** to certain data and methods within a class. It protects sensitive information and ensures that data can only be modified through controlled mechanisms.

Key Features of Encapsulation:

- **Data Hiding** → Prevents direct access to sensitive information.
- **Security** → Protects data from accidental modifications.
- **Modularity** → Organizes code into logical units.
- **Controlled Access** → Data is accessed through methods (getters and setters).
- **Flexibility** → Internal implementation can be changed without affecting other parts of the program.

Example:

Bank Account System

- **Data (Private)**: Account number, balance, PIN
- **Methods (Public)**: `deposit()`, `withdraw()`, `get_balance()`
- Users **cannot directly change balance**; they must use `withdraw()` or `deposit()`.

Data Abstraction

Data abstraction is the process of **hiding complex implementation details** and exposing only the necessary functionality to the user. It allows users to interact with a system without knowing the underlying logic.

Key Features of Data Abstraction:

- **Hides Implementation Details** → Only essential features are visible to the user.
- **Simplifies Complex Systems** → Users don’t need to understand the internal workings.
- **Improves Code Maintainability** → Changes in implementation do not affect the user.
- **Enhances Security** → Prevents unauthorized access to internal data.

Mobile Phone Interface

- **Visible to User:** Call, send messages, install apps.
- **Hidden Details:** Signal processing, OS operations, app execution.

Inheritance

Inheritance is a feature in Object-Oriented Programming (OOP) that allows a **new class (child class)** to **acquire the properties and behaviors** of an **existing class (parent class)**. It promotes **code reusability** and **hierarchical relationships**.

Key Features of Inheritance:

- **Code Reusability** → Avoids rewriting the same code in multiple classes.
- **Hierarchy Structure** → Establishes a parent-child relationship between classes.
- **Extensibility** → New features can be added without modifying existing code.
- **Improves Maintainability** → Changes in the parent class automatically reflect in child classes.

Types of Inheritance:

Single Inheritance → One class inherits from another (Parent → Child).

(One parent, one child.)

A **Car** inherits properties from a **Vehicle**.

- **Parent Class:** Vehicle (wheels, engine, fuel type).
- **Child Class:** Car (adds air conditioning, music system).

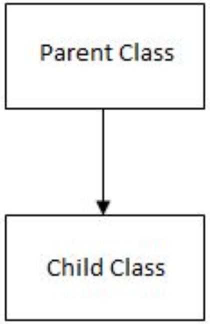


Fig: Single inheritance

Multiple Inheritance → A child class inherits from multiple parent classes.

(One child, multiple parents.)

A **Smartphone** inherits from both **Camera** and **Computer**.

- **Parent Class 1:** Camera (captures photos, records videos).
- **Parent Class 2:** Computer (runs apps, connects to the internet).
- **Child Class:** Smartphone (combines both features)

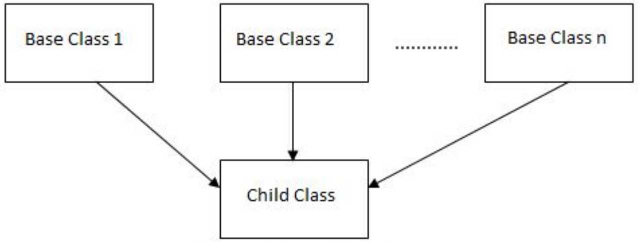


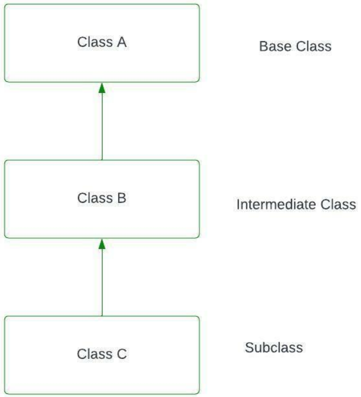
Fig: Multiple Inheritance

Multilevel Inheritance → A class inherits from a derived class (Grandparent → Parent → Child).

(One child, multiple parents.)

A **SportsCar** inherits features from **Car**, which in turn inherits from **Vehicle**.

- **Grandparent Class:** Vehicle (engine, wheels).
- **Parent Class:** Car (adds seats, music system).
- **Child Class:** SportsCar (adds turbo engine, spoiler).



Hierarchical Inheritance → Multiple child classes inherit from a single parent class.

(One parent, many children.)

A **Dog** and **Cat** both inherit from **Animal**.

- **Parent Class:** Animal (breathes, eats, moves).
- **Child Class 1:** Dog (barks, wags tail).
- **Child Class 2:** Cat (meows, climbs trees).

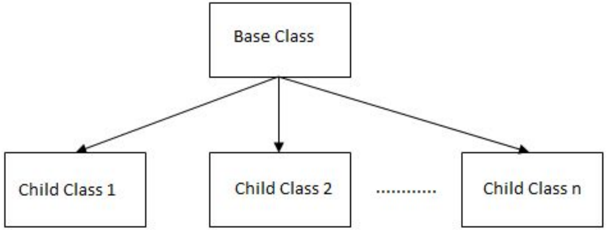


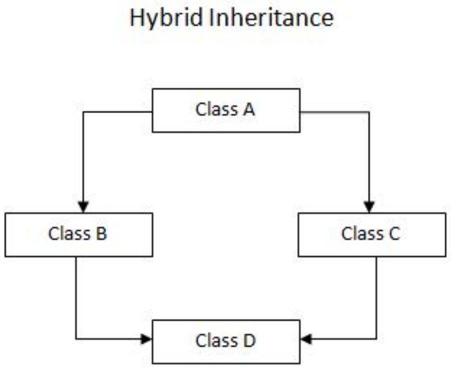
Fig: Hierarchical Inheritance

Hybrid Inheritance → Combination of two or more types of inheritance.

(Combination of multiple types.)

Consider a **University Management System** where:

- **"Person"** is the base class (common for all roles).
- **"Student"** and **"Professor"** inherit from "Person" (Hierarchical Inheritance).
- **"Teaching Assistant" (TA)** inherits from **both "Student" and "Professor"** (Multiple Inheritance).



Polymorphism

Polymorphism means **"many forms"** and allows a single function, method, or operator to behave differently based on the object calling it. It enhances **flexibility and reusability** in Object-Oriented Programming (OOP).

Key Features of Polymorphism:

- One interface, multiple implementations.
- Increases flexibility and scalability.
- Reduces code duplication and improves maintainability.
- Enhances reada**bility by using the same method name for different tasks.**

Example:

E-commerce Payment System
Different payment methods use the same `pay()` function.
`pay()` in **CreditCardPayment** → Deducts from a credit card.
`pay()` in **PayPalPayment** → Uses PayPal balance.

History of C++

Developed by: Bjarne Stroustrup at Bell Labs (now Nokia Bell Labs).
Year: Started in **1979**, officially named **C++ in 1983**.
Purpose: To enhance **C language** by adding Object-Oriented Programming (OOP) features.
Inspired by: C Language (efficiency & speed) + **Simula 67** (OOP concepts).
First compiler: Cfront, which translated C++ code into C.

Evolution of C++ Standards:

- **1979:** Bjarne Stroustrup started developing "C with Classes" at Bell Labs.
- **1983:** Renamed to **C++**, inspired by the **++ (increment) operator** in C.
- **1985:** First official **C++ Programming Language** book was published.
- **1990: Annotated C++ Manual** was released.
- **1998:** First **ISO Standard C++ (C++98)** was released.
- **2003:** Minor updates released as **C++03**.
- **2011: C++11** introduced modern features like **auto, lambdas, smart pointers, and multithreading**.
- **2014: C++14** added small improvements to C++11.
- **2017: C++17** introduced features like **structured bindings, parallel algorithms, and constexpr if**.
- **2020: C++20** introduced modules, coroutines, and improved concurrency support.
- **2023: C++23** (latest version) brought further improvements in safety, performance, and expressiveness.

Features of C++

Key Features of C++

- **Middle-Level Language** → Combines high-level and low-level programming.
- **Object-Oriented** → Supports **Encapsulation, Inheritance, Polymorphism, and Abstraction**.
- **Fast & Efficient** → Offers high performance due to direct memory manipulation.
- **Portable** → Can run on multiple operating systems with minimal changes.
- **Extensible & Scalable** → Allows the creation of large, modular applications.
- **Memory Management** → Provides **manual memory control** using `new` and `delete`.
- **Standard Template Library (STL)** → Supports reusable **algorithms, containers, and iterators**.
- **Multi-Paradigm Support** → Supports **procedural, object-oriented, and generic programming**.
- **Concurrency & Multithreading** → Built-in support for parallel programming.
- **Highly Used** → Widely used in **game development, finance, operating systems, and embedded systems**.

Why is C++ Called a Middle-Level Language?

Combines both Low-Level & High-Level Programming:

- Low-Level → In C++, you can use pointers to access and manipulate memory, similar to low-level languages.
- High-Level → C++ supports classes, objects, and inheritance to create reusable and modular applications.

Rules of C++ programming

- C++ follows a structured set of rules for writing programs.
- These rules ensure **correct execution, readability, and maintainability**.

1. **Every C++ Program Must Have `main()`**
   Rule: Execution starts from the `main()` function.
2. **Statements End with a Semicolon (`;`)**
   Rule: Every command must end with `;`.
3. **C++ is Case-Sensitive**
   Uppercase and lowercase letters are treated differently.
4. **Use of `{ }` (Curly Braces) for Code Blocks**
   Functions, loops, and conditions must be enclosed in `{ }`.
5. **Comments for Readability (`//` and `/* */`)**
   Comments explain code but are ignored during execution
6. **Every Variable Must Have a Data Type**
   Variables must be defined with a type (e.g., `int`, `float`, `char`).
7. **Parentheses `()` are Mandatory for Functions & Conditions**
   Functions and conditions require `()`.
8. **Code Must Be Inside a Function or Class**
   Standalone statements are not allowed.
9. **Double Quotes `" "` for Strings & Single Quotes `' '` for Characters**
   Strings use `" "` while single characters use `' '`.

Structure of C++ program

- **Documentation Section**: Explains the program's purpose.
- **Linking Section**: Includes necessary headers and namespaces.
- **Definition Section**: Defines constants and data type aliases.
- **Global Declaration Section**: Declares global variables.
- **Function Declaration Section**: Declares functions used in `main()`.
- **Main Function**: Execution starts here.
- **Function Definition Section**: Defines functions like `greet()`.

C++ Tokens

Tokens are the smallest building blocks of C++ programs.
The compiler divides the program code into tokens for further processing.

- Identifiers
- Keywords
- Constants
- Operators
- Special characters

Identifiers

In C++, **identifiers** are the names given to various program elements like variables, functions, classes, structs, etc. These names are used to uniquely identify the entities within the program.

```
// Creating a variable
int val = 10;
// Creating a function
void func() {}
```

`val` and **`func`** are identifiers in the above code.

Rules for Naming Identifiers in C++

Characters Allowed:

- Identifiers can contain letters (A-Z or a-z), digits (0-9), and underscores (`_`).
- Special characters and spaces are not allowed.

Start with a Letter or Underscore:

- Identifiers must start with a letter (A-Z or a-z) or an underscore (`_`).
- **Invalid**: `123name`, `name!`

Cannot Be C++ Reserved Keywords:

- Reserved words like `int`, `return`, `class`, etc., cannot be used as identifiers.
- **Example**: `int` cannot be used as an identifier.

Unique in Its Scope:

- An identifier must be unique within its scope (e.g., within a function or class).

Case-Sensitive:

- C++ is case-sensitive, so `Num` and `num` are different identifiers.

Valid Identifiers:

- `firstName`, `_age`, `totalAmount`, `x1`

Invalid Identifiers:

- `1stValue` (starts with a digit)
- `class` (reserved keyword)
- `total amount` (contains a space)

Naming Conventions in C++

Naming conventions are not mandatory rules but community best practices for clearer, more understandable code.

For Variables:

- Use **camelCase** (e.g., `studentName`, `totalAmount`).
- Start with a lowercase letter.
- Use descriptive names to explain the variable’s purpose.

For Functions:

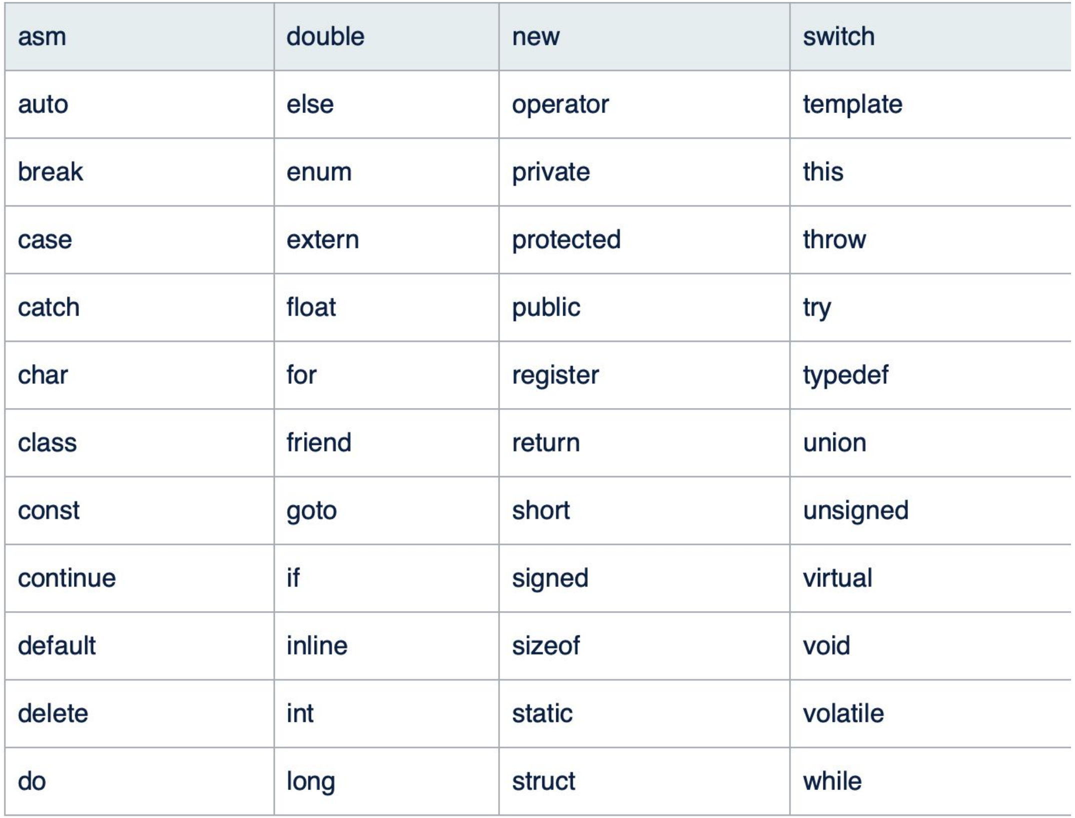
- Use **camelCase**.
- Function names should generally represent **actions** (e.g., `getName()`, `calculateTotal()`).

For Classes:

- Use **PascalCase** (e.g., `Student`, `CarModel`).
- Class names should represent **nouns** or **noun phrases**.

Keywords

- Reserved words with specific meaning in C++.
- Cannot be used as identifiers.



asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

Keywords vs Identifiers

Keywords	Identifiers
Predefined/Reserved words	User-defined names
Defines the type of entity	Classifies the name of the entity
Contain only alphabetical characters	Can consist of letters, digits, and underscores
Always lowercase	Can be uppercase, lowercase, or mixed case
Cannot use special symbols	Can use underscores, but no other special characters
Examples: int, char, while, if, for, return	**Examples**: studentName, totalAmount, myFunction, variable_1

Constants

- Constants are variables with fixed values that **cannot** be changed during the program's execution.
- Once initialized, the constant value remains the same throughout the program.
- Constants can be of any data type in C++ such as `int`, `char`, `float`, `string`, etc.

Types of Constants in C++

Using `const` Keyword

- This is one of the older methods inherited from the C language.

Syntax:

```
const DATATYPE variable_name = value;
```

- Constants defined with `const` must be initialized at the time of declaration and their values cannot be changed later.

Using constexpr Keyword

- constexpr constants are initialized at **compile-time**.
- The value of the constant must be known during the compilation process.
- **Syntax**:

Syntax:

```
constexpr DATATYPE variable_name = value;
```

More efficient as the values are evaluated by the compiler

Using #define Preprocessor

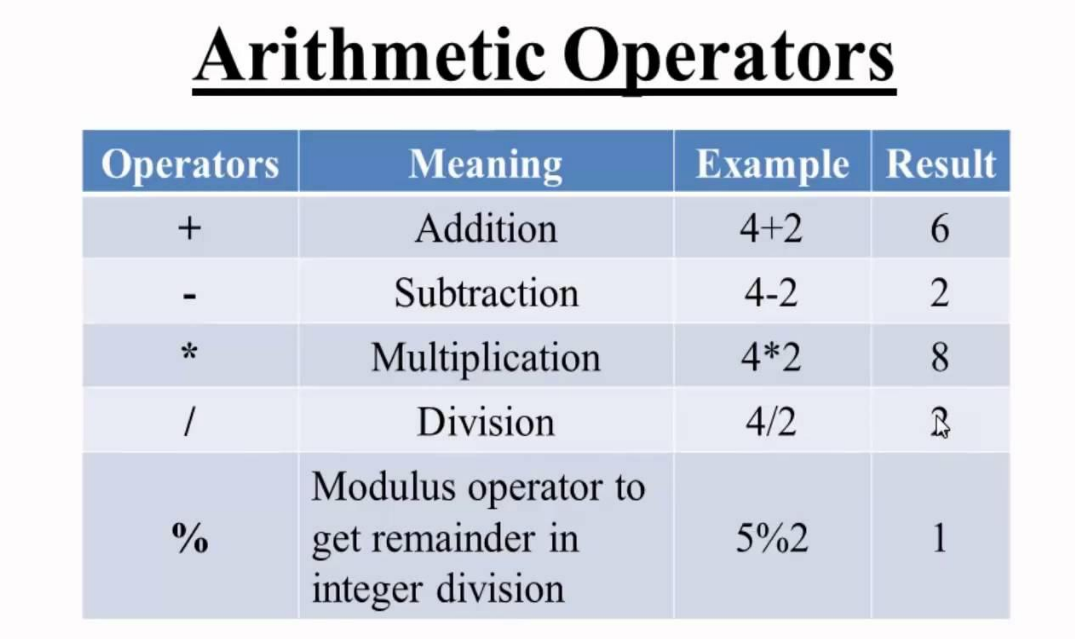
- Defines **macro constants** (alias for values) during the preprocessing stage.
- This method is **less preferred** due to lack of **type safety**.

Syntax:

```
#define MACRO_NAME replacement_value
```

Operators

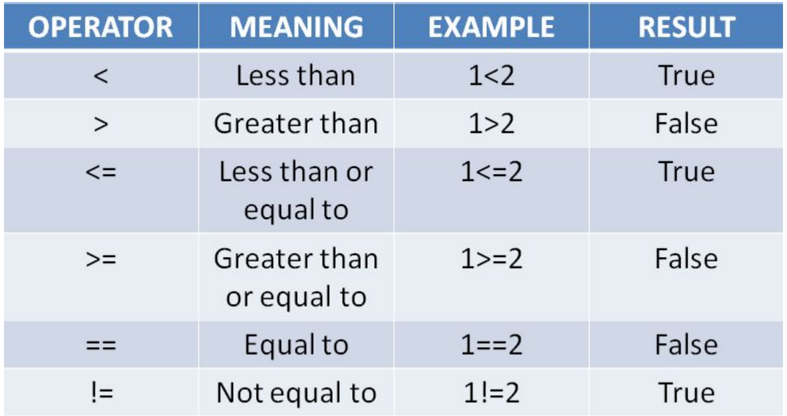
- Arithmetic Operators
  Symbols: +, -, *, /, %, ++, –



Arithmetic Operators

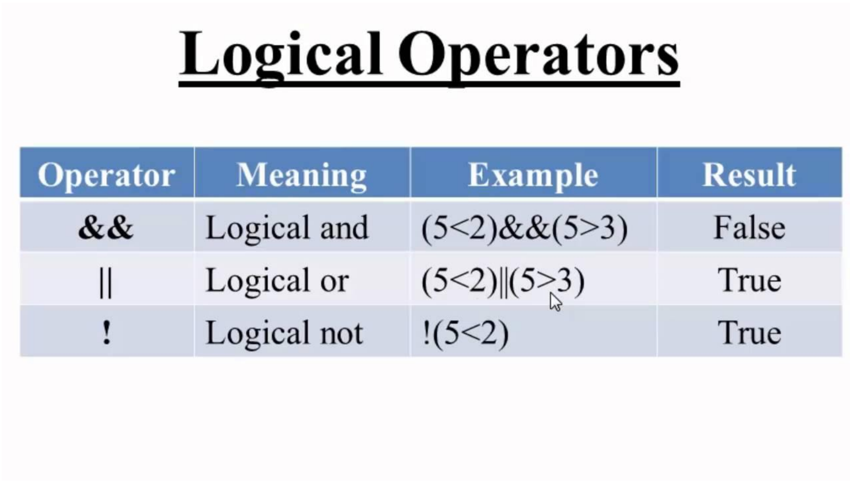
Operators	Meaning	Example	Result
+	Addition	4+2	6
-	Subtraction	4-2	2
*	Multiplication	4*2	8
/	Division	4/2	2
%	Modulus operator to get remainder in integer division	5%2	1

- Relational Operators
  Symbols: ==, >, <, >=, <=, ! =



OPERATOR	MEANING	EXAMPLE	RESULT
<	Less than	1<2	True
>	Greater than	1>2	False
<=	Less than or equal to	1<=2	True
>=	Greater than or equal to	1>=2	False
==	Equal to	1==2	False
!=	Not equal to	1!=2	True

- Logical Operators
  Symbols: &&, | |, !

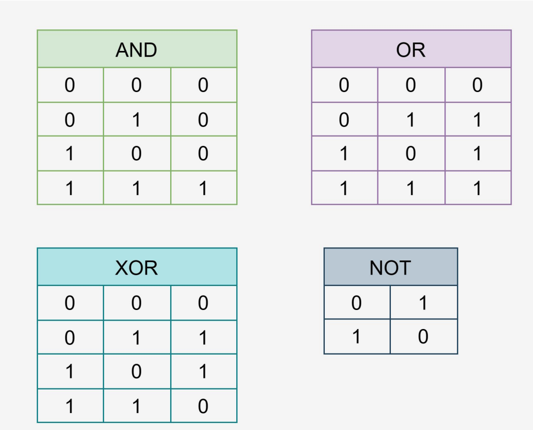


Logical Operators

Operator	Meaning	Example	Result
&&	Logical and	(5<2)&&(5>3)	False
||	Logical or	(5<2)||(5>3)	True
!	Logical not	!(5<2)	True

- Bitwise Operators

Symbols: &, |, ^, <<, >>, ~

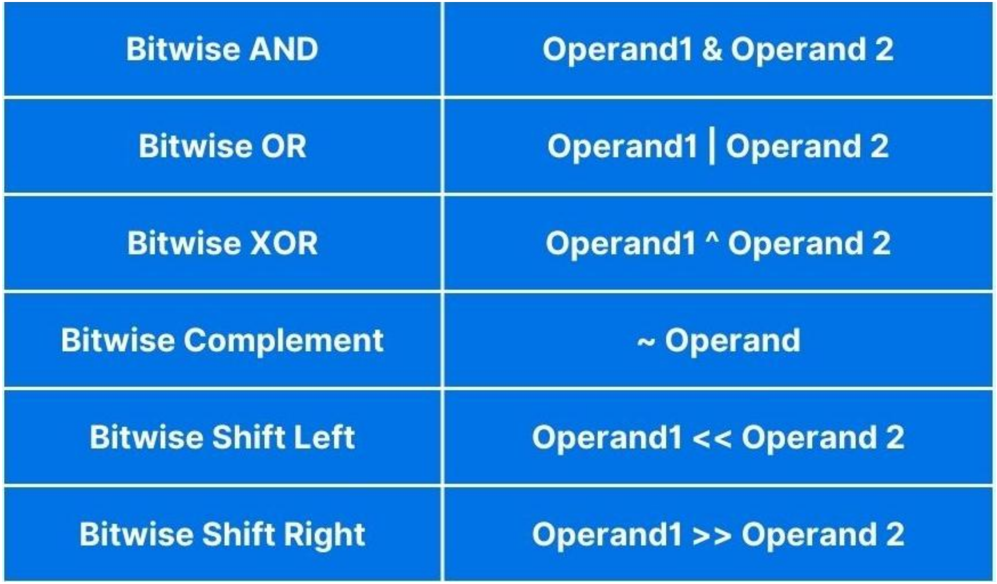


AND		
0	0	0
0	1	0
1	0	0
1	1	1

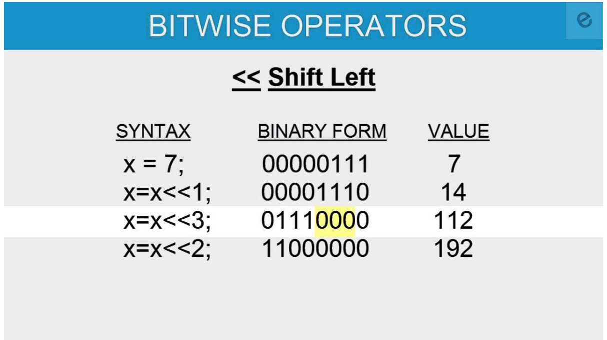
OR		
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
0	0	0
0	1	1
1	0	1
1	1	0

NOT	
0	1
1	0



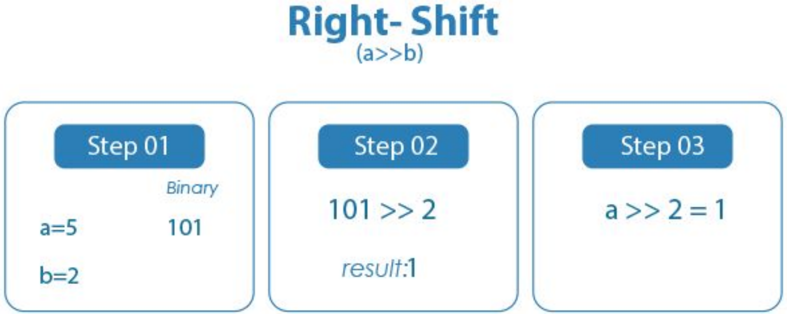
Bitwise AND	Operand1 & Operand 2
Bitwise OR	Operand1 | Operand 2
Bitwise XOR	Operand1 ^ Operand 2
Bitwise Complement	~ Operand
Bitwise Shift Left	Operand1 << Operand 2
Bitwise Shift Right	Operand1 >> Operand 2



BITWISE OPERATORS

<< Shift Left

SYNTAX	BINARY FORM	VALUE
x = 7;	00000111	7
x=x<<1;	00001110	14
x=x<<3;	01110000	112
x=x<<2;	11000000	192



Right- Shift

(a>>b)

Step 01

Binary

a=5 101

b=2

Step 02

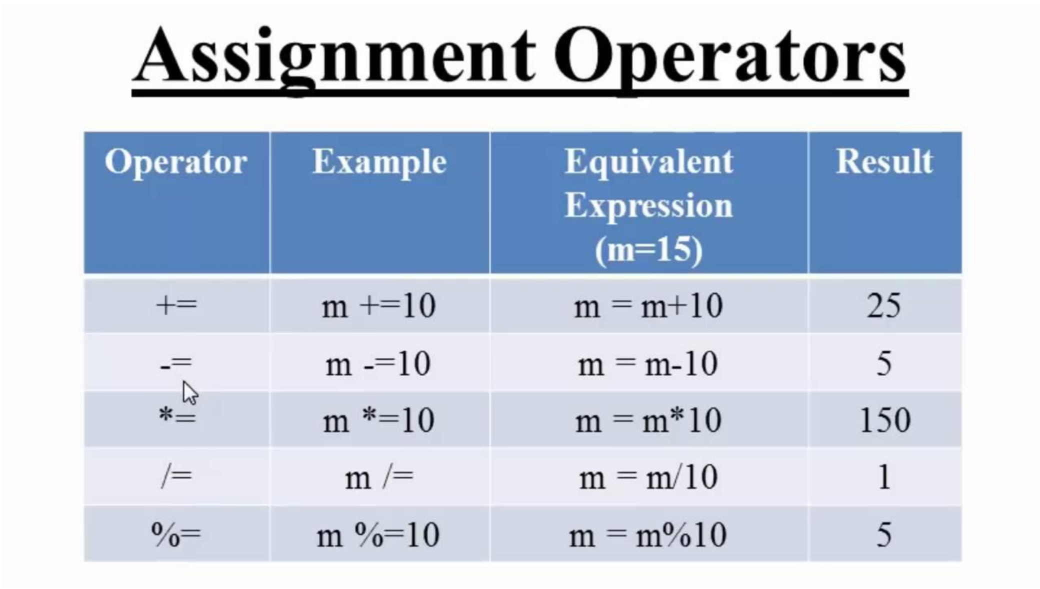
101 >> 2

result:1

Step 03

a >> 2 = 1

- Assignment Operators
  Symbols: =, +=, -=, *=, /=

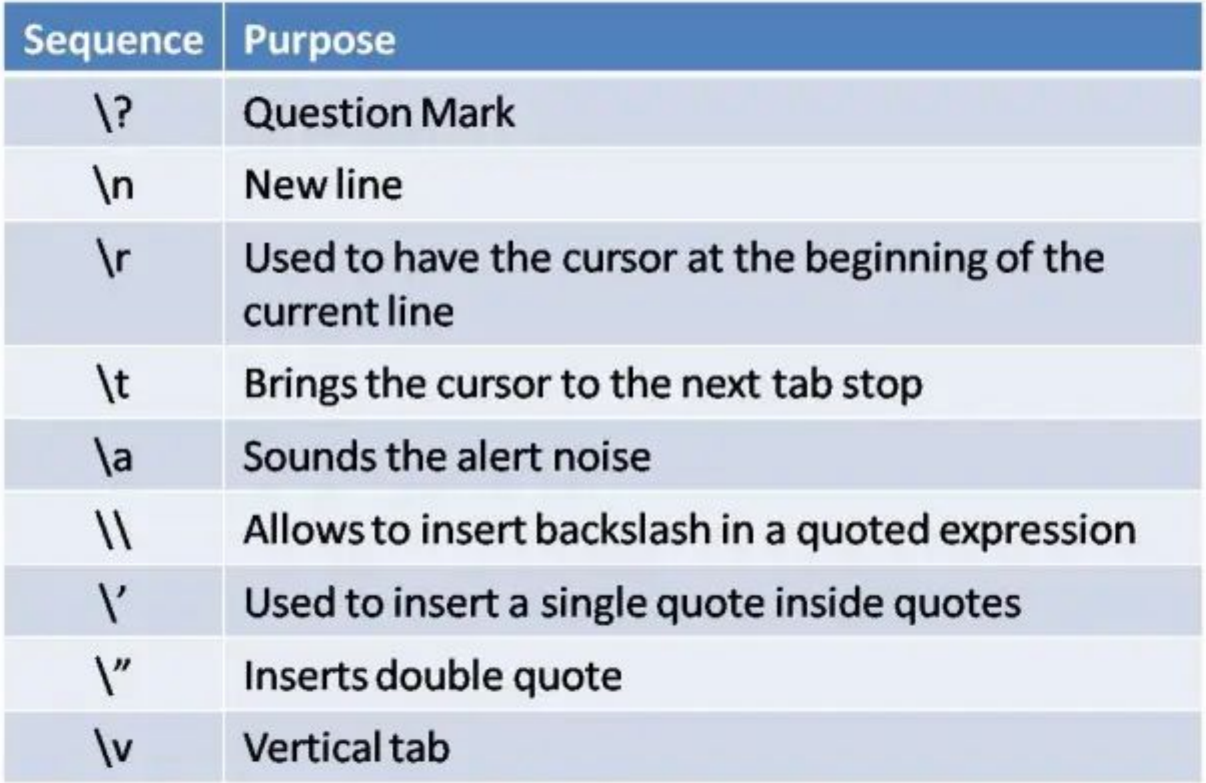


Assignment Operators

Operator	Example	Equivalent Expression (m=15)	Result
+=	m +=10	m = m+10	25
-=	m -=10	m = m-10	5
*=	m *=10	m = m*10	150
/=	m /=	m = m/10	1
%=	m %=10	m = m%10	5

- Ternary Operator
  Symbol: ? :
  Example: Expression1 ? Expression2 : Expression3

Special characters/ Escape Sequence



Sequence	Purpose
\?	Question Mark
\n	New line
\r	Used to have the cursor at the beginning of the current line
\t	Brings the cursor to the next tab stop
\a	Sounds the alert noise
\\	Allows to insert backslash in a quoted expression
\’	Used to insert a single quote inside quotes
\”	Inserts double quote
\v	Vertical tab

C++ Data types

- **Data types** specify what kind of data a variable can store.
- Data types help the **compiler** allocate memory according to the variable's type.
- C++ supports a wide variety of data types, each designed for different uses.

1. **Basic/Primitive**
   Built-in types used to store simple values.
   Examples: `int`, `char`, `float`, `double`, `bool`, `void`.
2. **Derived**
   Data types derived from basic data types.
   Examples: Arrays, pointers, references, functions.
3. **User defined**
   Custom data types created by programmers to meet specific needs.
   Examples: `class`, `struct`, `union`, `typedef`, `using`.

Basic Data Types in C++

- **Character Data Type (`char`)**
  - Stores a single character.
  - **Size**: 1 byte.
  - **Syntax**:

    ```
    char name;
    ```

- **Integer Data Type (`int`)**
- Stores integer numbers.
- **Size**: 4 bytes (64-bit systems).
- **Range**: -2,147,483,648 to 2,147,483,647.
- **Syntax**:

  ```
  int name;
  ```

- **Boolean Data Type (`bool`)**
- Stores logical values: `true` (1) or `false` (0).
- **Size**: 1 byte.
- **Syntax**:

  ```
  bool name;
  ```

➔ **Floating Point Data Type (`float`)**

- Stores decimal numbers with single precision.
- **Size**: 4 bytes.
- **Range**: 1.2E-38 to 3.4e+38.
- **Syntax**:

```
float name;
```

➔ **Double Data Type (`double`)**

- Stores decimal numbers with double precision.
- **Size**: 8 bytes.
- **Range**: 1.7e-308 to 1.7e+308.

```
double name;
```

➔ **Void Data Type (`void`)**

- Represents **absence of value**.
- Used for pointers and functions that don't return a value.
- **Syntax**:

```
void functionName();
```

➔ The size of data types **varies across different systems** (32-bit vs. 64-bit systems).

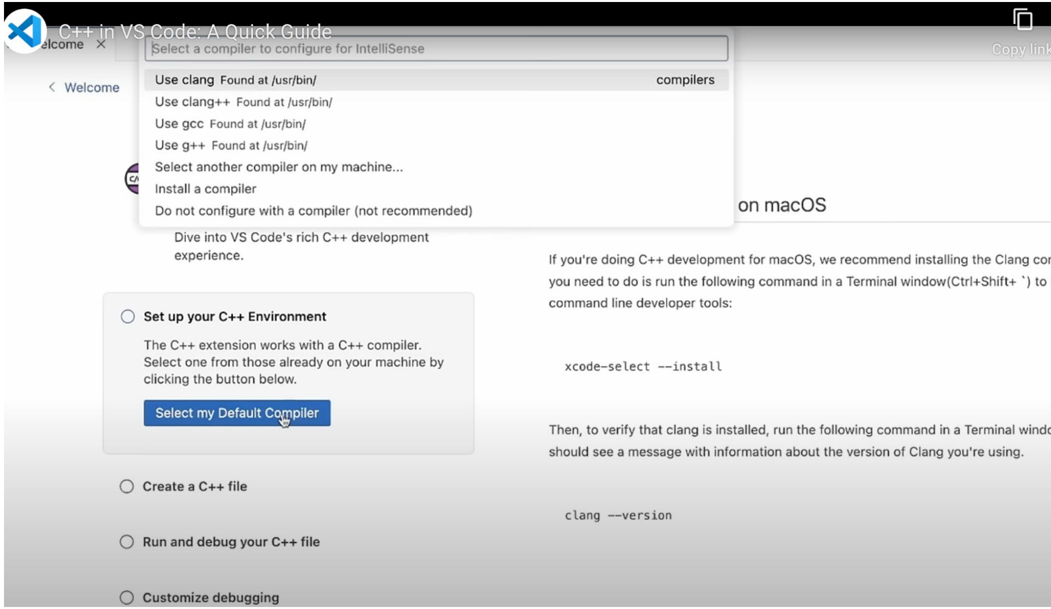
```
sizeof(data_type)
```

Installation IDE VSCode

Installation Steps

https://code.visualstudio.com/download

- **Windows (Personal Use):** Install the **User Installer**
- **Mac:** Download and install the **.zip package**
- **Linux (Ubuntu/Debian):** Use the **.deb package**



C++ in VS Code: A Quick Guide

Select a compiler to configure for IntelliSense

- Use clang Found at /usr/bin/
- Use clang++ Found at /usr/bin/
- Use gcc Found at /usr/bin/
- Use g++ Found at /usr/bin/
- Select another compiler on my machine...
- Install a compiler
- Do not configure with a compiler (not recommended)

Dive into VS Code's rich C++ development experience.

○ Set up your C++ Environment

The C++ extension works with a C++ compiler. Select one from those already on your machine by clicking the button below.

Select my Default Compiler

○ Create a C++ file

○ Run and debug your C++ file

○ Customize debugging

on macOS

If you're doing C++ development for macOS, we recommend installing the Clang co
you need to do is run the following command in a Terminal window(Ctrl+Shift+ `) to
command line developer tools:

```
xcode-select --install
```

Then, to verify that clang is installed, run the following command in a Terminal windo
should see a message with information about the version of Clang you're using.

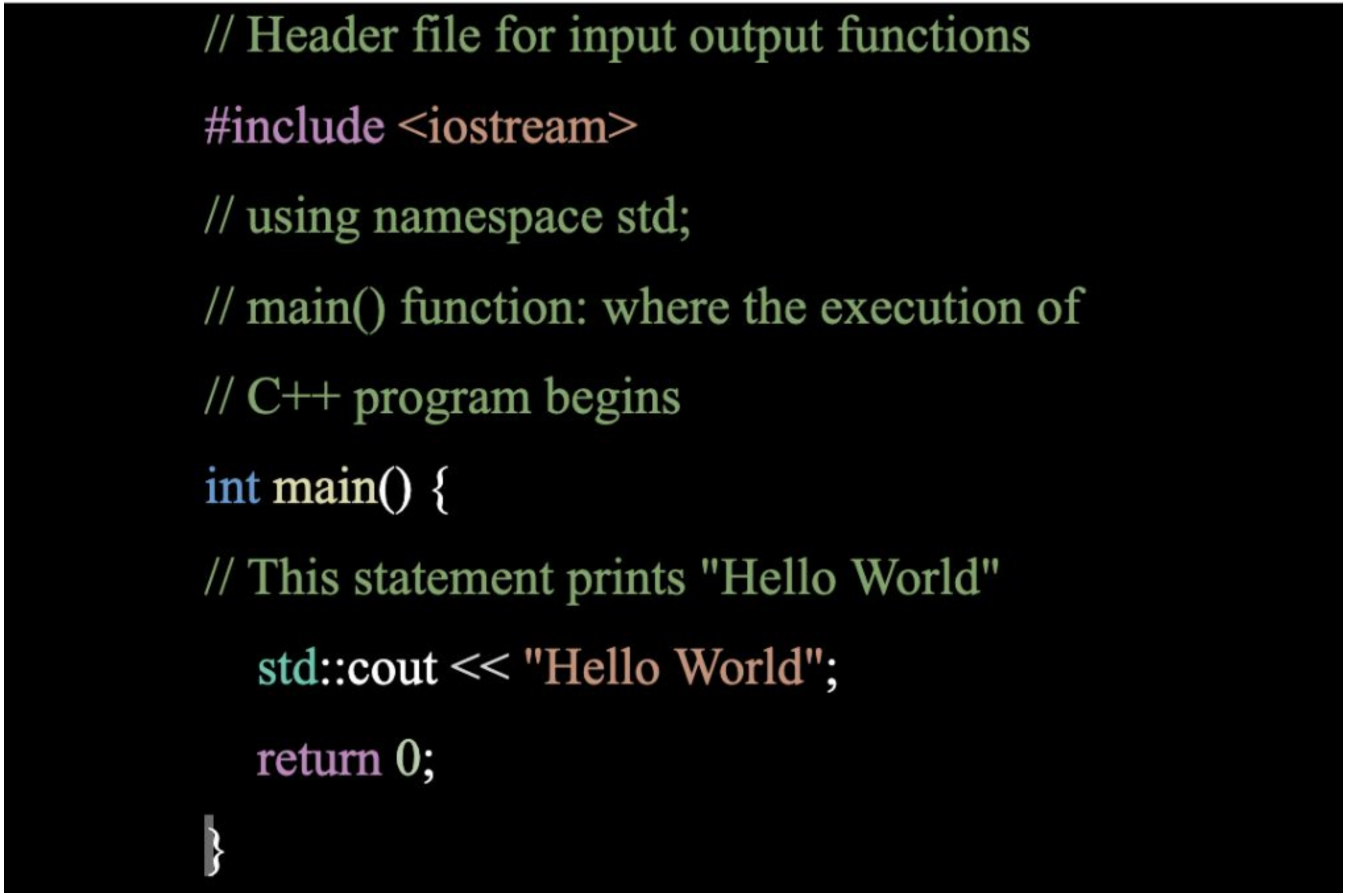
```
clang --version
```

Setting Up C++ in VS Code

https://code.visualstudio.com/docs/cpp/introvideos-cpp

1. Open the **Left Panel** and go to **Extensions**.
2. Search for **C++** and install the **C/C++ extension**.
3. Select a compiler from the available options or install one if required.
4. Click **Set as Default Compiler** to apply your selection.

How to Write and Run Your First Program in C++



```cpp
// Header file for input output functions
#include <iostream>
// using namespace std;
// main() function: where the execution of
// C++ program begins
int main() {
// This statement prints "Hello World"
    std::cout << "Hello World";
    return 0;
}
```

Assignment

1. List all versions of C++ and highlight the major changes introduced in each version.
2. Select 10 reserved keywords in C++ and explain their usage with examples.
3. How are bits allocated for each data type in 32-bit and 64-bit systems?