Unit 1

- **Class:** The blueprint or template (e.g., the architectural drawing of a house).
- **Object:** The real-world entity created from the blueprint (e.g., the actual house built from the drawing).
- **Encapsulation:** Wrapping data and functions together into a single unit (class) to protect it.
- **Abstraction:** Hiding complex implementation details and showing only the necessary features.
- **Inheritance:** Creating a new class from an existing class to reuse code.
- **Polymorphism:** One name, many forms (e.g., a function behaving differently based on input).

CLASS	OBJECT
Class is a blue print from which objects are created.	Object is an instance of a class.
Class is a group of similar objects.	Object is a real world entity such as chair, pen, table, laptop etc.
Class is a logical entity.	Object is a physical entity.
Class is declared once.	Object is created many times as per requirement.
Class doesn't allocate memory when it is created.	Object allocates memory when it is created.
Class is declared using class keyword. Ex: class Employee{}	Object is created through new keyword. Ex: Employee ob = new Employee();
There is only one way to define a class, i.e., by using class keyword.	There are different ways to create object in java: New keyword, newInstance() method, clone() method, and deserialization.

Inline Function:

An inline function is a function in C++ whose code is expanded at the point of call at compile time. It reduces function-call overhead.

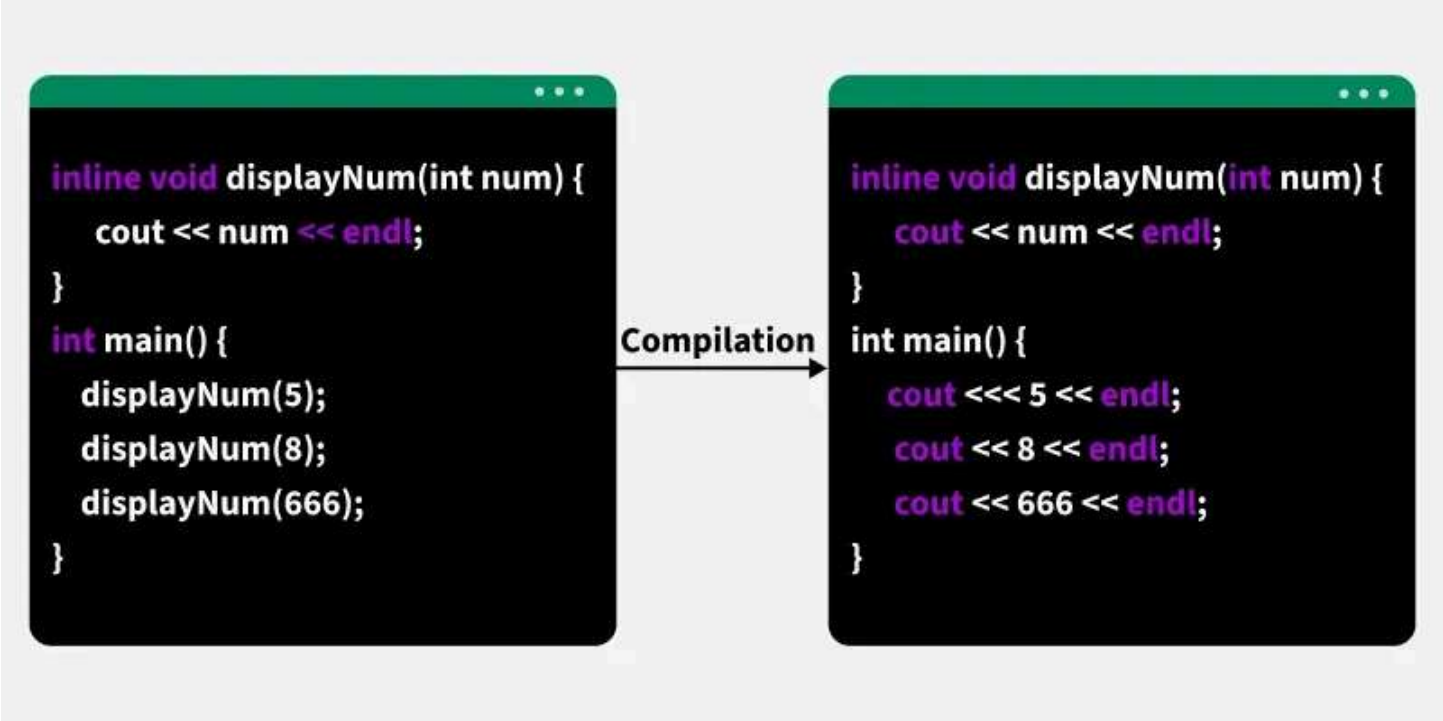
For small, frequently called functions, the time taken for these "jumping" and stack operations (the function call overhead) can be greater than the time taken to execute the function's actual code.

reduce function-call overhead and potentially increase execution speed, especially for **small, frequently used functions**.

- The inline keyword suggests replacing a function call with its code to reduce overhead.
- Inlining is a request, not guaranteed by the compiler.
- The compiler may ignore inlining if the function contains loops, recursion, static variables, switch/goto, or a non-void function without a return statement..

Need for Inline Functions:

- Function calls involve overhead from storing the return address, passing arguments, and returning control, which can be significant for small, frequently used functions.
- Inline functions help eliminate this overhead by replacing the function call with the actual code, improving efficiency.
- Inline functions are useful only when the function call overhead is higher than the function's execution time.

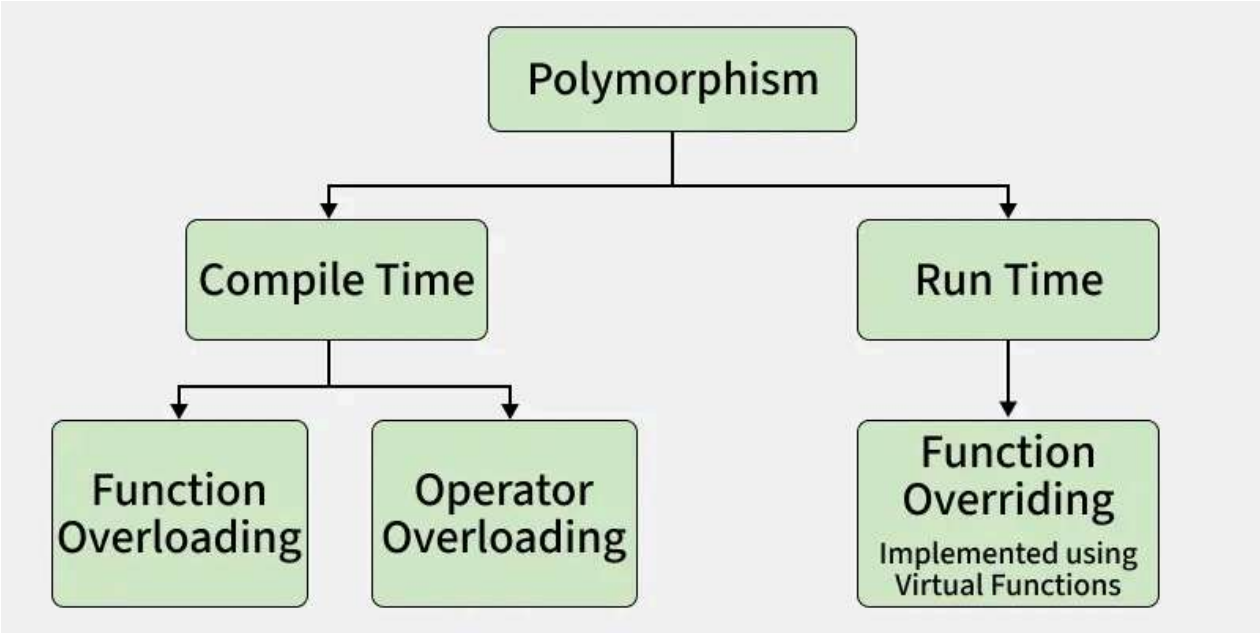


Reference Variable:

a reference variable acts as an alias or an alternative name for an existing variable. It provides a way to refer to the same memory location as the original variable, meaning any changes made through the reference directly affect the original variable.They do not occupy separate memory like a new variable would.

1.
```
int a = 10;
int &ref = a; // ref is an alias for a
```
——————————————————————————
2.
```
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    int &r = x;    // r is a reference to x

    cout << x << endl; // 5
    cout << r << endl; // 5

    r = 20;    // change r → x also changes

    cout << x << endl; // 20
    cout << r << endl; // 20
}
```

Polymorphism



Function Overloading

Function overloading in C++ is a compile-time polymorphism technique where two or more functions share the same name but differ in their parameter list (type, number, or sequence). It enables functions to perform similar tasks with different types or amounts of data. The compiler decides which function to call using the concept of function signature.

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

// Function for Circle
float area(float r) {
    return 3.14 * r * r;
}

// Function for Rectangle (Overloaded)
int area(int l, int b) {
    return l * b;
}

int main() {
    cout << "Area of Circle: " << area(5.0f) << endl;
    cout << "Area of Rectangle: " << area(5, 10) << endl;
    return 0;
}
```

Operator Overloading

Operator overloading is a compile-time polymorphism technique in C++ that allows developers to give special meaning to existing operators (like `+`, `-`, `*`, `<<`, `++`) when used with user-defined data types (classes). It enables objects to be manipulated just like built-in data types (int, float, etc.), making the code more intuitive and readable. The `operator` keyword is used to define the function.

Key points:

- It does not change the original meaning of the operator for built-in types.
- It cannot change the precedence or associativity of the operator.
- Some operators (like `::`, `.`, `.*`, `sizeof`) cannot be overloaded.

Example: Below is an example of overloading the `+` operator to add two **Complex** numbers.

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

```
class Score {
public:
    int val;

    // Constructor to set the score
    Score(int x = 0) {
        val = x;
    }

    // Overloading the '+' operator
    Score operator + (Score other) {
        Score temp;
        // Add the value of the current object to the value of the 'other' object
        temp.val = val + other.val;
        return temp;
    }
};

int main() {
    Score student1(50);
    Score student2(40);
    Score total;

    // Here, the '+' triggers the operator function we wrote above
    total = student1 + student2;

    cout << "Total Score: " << total.val << endl;

    return 0;
}
```

Output:

Total score 90

Function overriding:

Function overriding occurs when a derived class defines a function that has the same name, return type, and parameter list as a function in the base class. It is used to achieve runtime polymorphism, allowing the derived class to replace or modify the behavior of the base class function. The **virtual** keyword is used in the base class to enable overriding.

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

class Animal {
public:
    virtual void sound() {   // still needs virtual for polymorphism
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() {   // overriding the base class function
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* a = new Dog();
    a->sound();  // Calls Dog's sound() because of virtual

    return 0;
}
```

Method Overloading	Method Overriding
1. Method overloading is used to increase the readability of the program.	1. Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2. It is is a static (or) compile time binding	2. It is dynamic (or) runtime binding.
3. Method overloading is performed within class.	3. Method overriding occurs in two classes that have IS-A (inheritance) relationship.
4. In case of method overloading, parameter must be different.	4. In case of method overriding, parameter must be same.
5. Method overloading is the example of compile time polymorphism.	5. Method overriding is the example of run time polymorphism.
6. In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	6. Return type must be same or covariant in method overriding.

Unit II: Classes, Objects, and Constructors

Access Specifiers:

public:

- From **anywhere** in the program
- Outside the class
- Using object of the class

```
class Demo {
public:
    int x;   // public
};

int main() {
    Demo d;
    d.x = 10;   // allowed
}
```

Private:

- **Only inside the same class**
- NOT accessible outside the class
- NOT accessible by derived classes

```
class Demo {
private:
    int x;   // private
};

int main() {
    Demo d;
    d.x = 10;   // ❌ ERROR: x is private
}
```

Protected:

- Inside the same class

- Inside derived (child) classes
- NOT accessible by objects outside

```
class A {
protected:
    int x;
};

class B : public A {
public:
    void show() {
        x = 10;    // ✓ allowed (derived class can access protected)
    }
};

int main() {
    A obj;
    obj.x = 5;  // ❌ ERROR: protected cannot be accessed outside
}
```

Static Members:

A static member is a class-level member (data or function) that belongs to the class itself, not to any specific object of the class. This means there is only one copy of a static data member shared among all objects, and a static member function can be called without creating an object. They are useful for storing data common to all instances, like a counter for the number of objects created, or for implementing functions that don't need to access any specific object's data.

```
#include <iostream>

class Counter {
public:
    static int count; // Declaration of a static data member

    Counter() {
        count++; // Increment count every time an object is created
    }
};

// Definition and initialization of the static data member outside the class
int Counter::count = 0;

int main() {
    Counter obj1;
    Counter obj2;
```

```
    Counter obj3;

    std::cout << "Number of objects created: " << Counter::count << std::endl;
    // Accessing static member using class name

    return 0;
}
```

Static Members function:

A static member function is a function declared with the static keyword that belongs to the class itself, not to any specific object instance. This means it can be called using the class name and the scope resolution operator (::) without creating an object. Static member functions can only access other static members (both variables and functions) of the class and do not have access to the implicit this pointer.

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

class Student {
public:
    static int count;  // static data member
    Student() {
        count++;
    }

    static void showCount() {   // static function
        cout << "Total students: " << count << endl;
    }
};

int Student::count = 0;

int main() {
    Student s1, s2;
    Student::showCount();  // calling without object
    return 0;
}
```

Constructor:

A constructor is a special method in object-oriented programming that is automatically called when an object is created to initialize it. It sets up the new object by allocating memory and setting initial values for its variables, ensuring the object is ready for use from the moment it is created

Destructor:

A destructor is a special member function in object-oriented programming that is automatically called when an object is destroyed. Its primary purpose is to clean up resources the object was using, such as deallocating memory, closing files, or terminating network connections, before the memory is released.

Syntax:

```
class Demo {
   public:
      Demo() { ... }      // Default Constructor
      Demo(int x) { ... } // Parameterized Constructor
      ~Demo() { ... }     // Destructor (starts with tilde ~)
};
```

Unit III: Inheritance

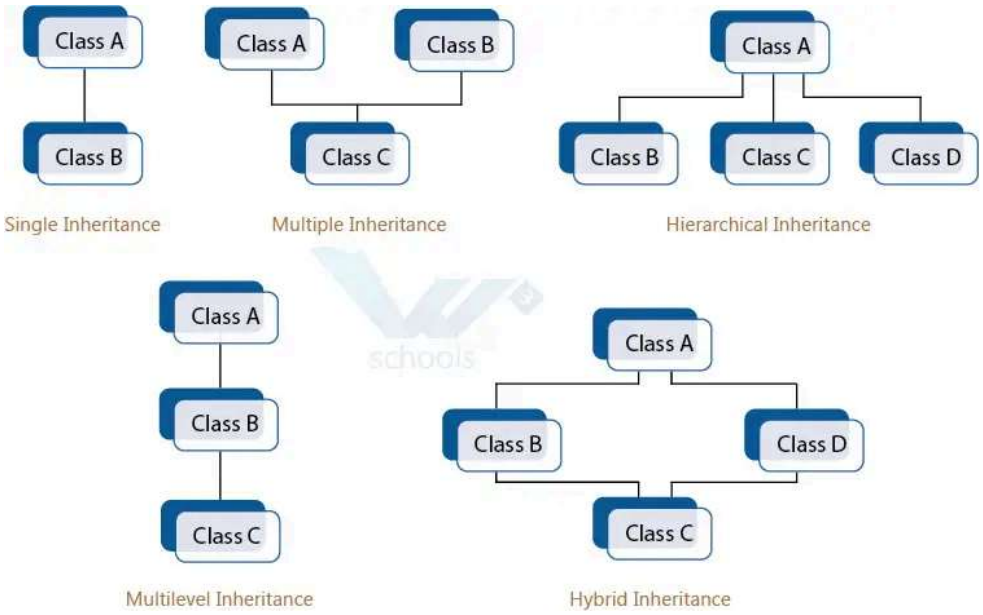
Inheritance:

Inheritance in C++ is a core concept of Object-Oriented Programming (OOP) that allows a new class, known as the derived class (or child class), to inherit properties and behaviors (data members and member functions) from an existing class, known as the base class (or parent class). This mechanism promotes code reusability, extensibility, and establishes an "is-a" relationship between classes.

Syntax:

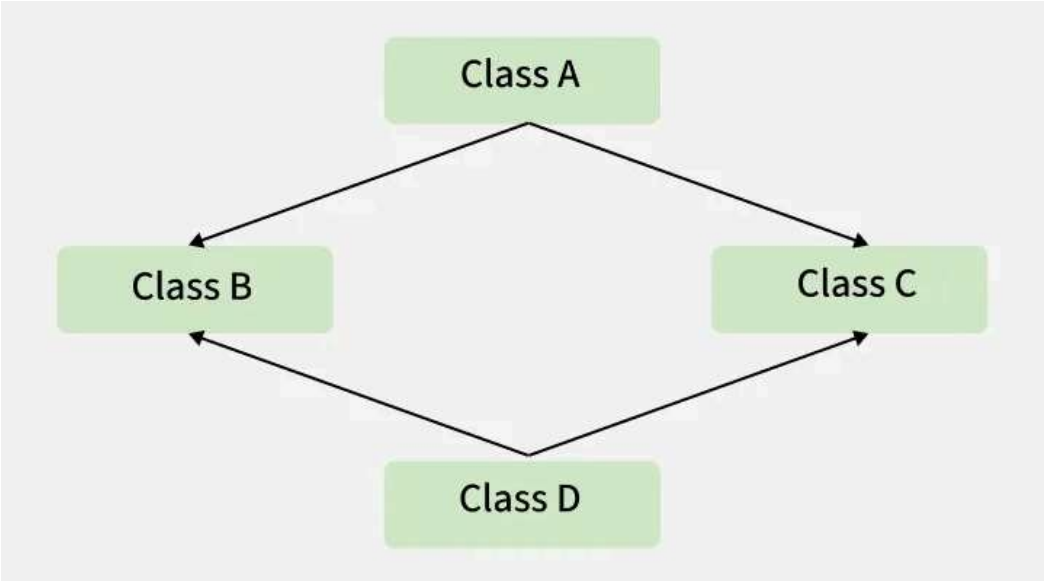
```
class BaseClass {
   // Base class members (public, protected, private)
};

class DerivedClass : accessSpecifier BaseClass {
   // Derived class members
};
```

Types:



Virtual Base Class:

A virtual base class in C++ is a mechanism used to resolve ambiguity and prevent data member duplication in scenarios involving multiple inheritance




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

class A {
public:
```

```
        int a;
        A() // constructor
        {
                a = 10;
        }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
        D object;
        cout << "a = " << object.a << endl;

        return 0;
}
//output
a=10
```

Virtual Functions:

a virtual function is a member function declared in a base class using the virtual keyword, which can be redefined (overridden) in a derived class. Its primary purpose is to enable runtime polymorphism, allowing the correct version of a function to be invoked based on the actual type of the object, even when accessed through a pointer or reference to the base class

Pure virtual function:

A pure virtual function is a function in a base class that has no body, only a declaration.
It forces the derived class to must provide its own function body.
A pure virtual function is declared by assigning = 0 to its declaration within the base class.

Program:

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

class Animal {
public:
    virtual void sound() = 0;   // Pure virtual function
};

class Dog : public Animal {
public:
    void sound() {
        cout << "Dog barks" << endl;
    }
};

class Cat : public Animal {
public:
    void sound() {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Dog d;
    Cat c;

    d.sound();  // Dog barks
    c.sound();  // Cat meows
}
```

Abstract class:

an **abstract class** is a class that **cannot be instantiated** (you cannot create an object of it directly). Its primary purpose is to serve as a base class (a blueprint) for other classes to inherit from.

To make a class abstract, it must contain at least one **pure virtual function**.

- A class with at least one pure virtual function becomes an abstract class and Objects of abstract classes cannot be created directly.
- Abstract classes are used to define interfaces and ensure common structure among derived classes.
- Useful in polymorphism where different classes share the same interface but have different behaviors.
- A pure virtual function forces derived classes to override it.

- virtual void draw() = 0; declares a pure virtual function, forcing derived classes to provide their own implementation.

Ex:

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

class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle\n";
    }
};

int main() {
    s->draw();

    delete s;
}
```

Unit IV: Templates and Exception Handling

Templates:

A way to write "Generic Code." Instead of writing one function for int, one for float, and one for char, you write **one** template that works for all.

template is a powerful tool for creating generic classes or functions. This allows us to write code that works for any data type without rewriting it for each type.

Class Templates

Class templates work similarly for classes, specifying parameters in angle brackets. An example is a simple stack,queue,linkedList

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

// Defining class template
```

```
template <typename T>
class Geek
{
  public:
    T x;
    T y;

    // Constructor
    Geek(T val1, T val2) : x(val1), y(val2)
    {
    }

    // Method to get values
    void getValues()
    {
        cout << x << " " << y;
    }
};

int main()
{
    Geek<int> intGeek(10, 20);
    Geek<double> doubleGeek(3.14, 6.28);
    intGeek.getValues();
    cout << endl;
    doubleGeek.getValues();

    return 0;
}
```

Output:
10 20
3.14 6.28

Function Templates

Function templates define a blueprint for functions using the template keyword followed by parameters like typename T. For example, a max function can compare any comparable types:

```
template <typename T>
T max(const T &a, const T &b) {
    return a < b ? b : a;
}
```

Exception Handling:

Exception handling in C++ is a mechanism to detect and manage runtime errors (errors that occur during program execution) in a structured way. Examples of runtime errors are..

- Division by zero
- Accessing invalid memory
- File I/O failures

1. try-catch Block

C++ provides an inbuilt feature for handling exceptions using try and catch block. It is an exception handling mechanism where the code that may cause an exception is placed inside the try block and the code that handles the exception is placed inside the catch block.

```
try {
    // Code that might throw an exception
}
catch (ExceptionType e) {
    // exception handling code
}
```

Example:

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

int main() {
    try {
        throw 42;  // Try different throws
    }
    catch (int e) {
        cout << "Integer: " << e << endl;
    }
    catch (const char* e) {
        cout << "String: " << e << endl;
    }
    catch (...) {
        cout << "Unknown exception" << endl;
    }
    return 0;
}
```

When an exception occurs in try block, the execution stops, and the control goes to the matching catch block for handling.

2. Throwing Exceptions

Throwing exception means returning some kind of value that represent the exception from the try block. The matching catch block is found using the type of the thrown value. The throw keyword is used to throw the exception.

```
try {
    throw val
}
catch (ExceptionType e) {
    // exception handling code
}
```

There are three types of values that can be thrown as an exception:

- Built-in Types
- Standard Exceptions
- Custom Exceptions

example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int x = 7;
    try {
        if (x % 2 != 0) {

            // Throwing int
            throw -1;
        }
    }

    // Catching int
    catch (int e) {
        cout << "Exception Caught: " << e;
    }
    return 0;
}
```

Unit V: File Handling

File Handling:

File handling in refers to the process of interacting with files stored on a computer's disk, allowing a program to read, write, and modify data permanently. This contrasts with data stored in RAM, which is lost when a program terminates.

Normally, when you close a program, data is lost (RAM is volatile). File handling allows you to save data permanently to the hard disk.

- **ofstream:** Output File Stream (Writes data to file).
- **ifstream:** Input File Stream (Reads data from file).
- **fstream**: Both Read and Write.

Reading from File:

Use >> operator or getline() on ifstream or fstream objects.

Example:

```
ifstream file("data.txt");
string line;
while (getline(file, line)) {
    cout << line << endl;
}
file.close();
```

Writing to file:

Use << operator on ofstream or fstream.

Example:

```
ofstream file("output.txt");
file << "Hello, file!" << endl;
file.close();
```

Modifying Text File Content

Use fstream with
ios::in
ios::out
to open existing files for both reading and writing. Read line-by-line, modify specific content (e.g., replace words), then rewrite using file pointers like seekp() or by overwriting the file

Example:

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

int main() {
    fstream file("data.txt", ios::in | ios::out);
    string line, modified;

    getline(file, line);
    modified = line.replace(0, 5, "Updated");  // Replace first 5 chars

    file.seekp(0);  // Go to start
    file << modified << endl;

    file.close();
    return 0;
}
```

Command Line Arguments:

Command-line arguments are arguments that are passed to a program when it is executed from the command line or terminal. They are provided in the command-line shell of operating systems with the program execution command.

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

int main(int argc, char* argv[]) {
    cout << "Arguments count: " << argc << endl;
    for(int i = 0; i < argc; i++) {
        cout << "argv[" << i << "]: " << argv[i] << endl;
    }
    return 0;
}
```