Okay, let's continue with the elaborated explanations and examples, starting with Object-Oriented Programming (OOP) in C++.

III. Object-Oriented Programming (OOP) in C++

- **Classes and Objects:**
  - **Explanation:** OOP is a programming paradigm centered around the concept of "objects," which are instances of "classes." A **class** is a blueprint or template that defines the characteristics (data members) and behaviors (member functions) of objects of that class. An **object** is a specific instance of a class, representing a real-world entity or concept.
  - **Class Definition (Data Members, Member Functions):**

```
C++
#include <iostream>
#include <string>

class Dog { // Class definition
public: // Access specifier (public: accessible from outside the class)
  std::string name; // Data member (attribute)
  int age;        // Data member (attribute)

  void bark() { // Member function (behavior)
    std::cout << "Woof!" << std::endl;
  }
  void displayInfo() { // Member function (behavior)
    std::cout << "Name: " << name << ", Age: " << age << std::endl;
  }
};
```

  - **Object Instantiation:** Creating objects (instances) of a class. ```c++ int main() { Dog myDog; // Creates an object named myDog of the[1] class Dog myDog.name = "Buddy";[2] myDog.age = 3; myDog.bark(); // Calls the bark member function on the myDog object myDog.displayInfo(); // Calls the displayInfo member function return 0; }

- **Encapsulation (Data Hiding):**
  - **Explanation:** Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on that data within a single unit (a class). It also involves **data hiding**, which means restricting direct access to some of the object's components (often data members) from outside the object. This is achieved using access specifiers.
  - **Access Specifiers:**
    - **public:** Members declared as public are

accessible from anywhere outside the class.

   - **private:** Members declared as private are only accessible from within the class itself (by its member functions). This provides data hiding and prevents direct external manipulation of the object's internal state.
   - **protected:** Members declared as protected are accessible from within the class itself and from derived classes (in inheritance).

- **Example:**

```cpp
#include <iostream>

class BankAccount {
private:
   double balance; // Private data member
public:
   BankAccount(double initialBalance) : balance(initialBalance) {}

  void deposit(double amount) {
     if (amount > 0) {
         balance += amount;
      }
   }

   void withdraw(double amount) {
      if (amount > 0 && amount <= balance) {
         balance -= amount;
      }
   }

   double getBalance() const { // const indicates this method does not modify the object's
state
```

```
        return balance;
    }
};

int main() {
    BankAccount account(1000.0);
    // account.balance = 500.0; // Error: balance is private
    account.deposit(200.0);
    account.withdraw(100.0);
    std::cout << "Current balance: " << account.getBalance() << std::endl;
    return 0;
}
```

- **Constructors:**
  - **Explanation:** Special member functions that are automatically called when an object of the class is created. They are used to initialize the object's data members. Constructors have the same name as the class and do not have a return type (not even `void`).
  - Default Constructor: A constructor that takes no arguments. If you don't define any constructors, the compiler provides a default constructor (which does nothing).

Code snippet
```
class MyClass {
public:
    MyClass() { // Default constructor
        std::cout << "Default constructor called." << std::endl;
    }
};

int main() {
  MyClass obj; // Calls the default constructor
  return 0;
}
```

- Parameterized Constructor: A constructor that takes one or more arguments. Used to initialize data members with specific values at the time of object creation.

C++
```
#include <iostream>
#include <string>

class Car {
public:
  std::string brand;
  std::string model;

  Car(std::string b, std::string m) : brand(b), model(m) { // Parameterized constructor
     std::cout << "Parameterized constructor called for " << brand << " " << model << "." <<
std::endl;
  }

  void display() const {
    std::cout << "Brand: " << brand << ", Model: " << model << std::endl;
  }
};

int main() {
  Car myCar("Toyota", "Camry"); // Calls the parameterized constructor
  myCar.display();
  return 0;
}
```

- Copy Constructor: A special constructor that is called when an object is initialized with another object of the same class (e.g., during object creation or when passing objects by value). The default copy constructor performs a shallow copy.

C++
```cpp
#include <iostream>

class MyInt {
public:
  int value;
  MyInt(int val = 0) : value(val) {
    std::cout << "Constructor called for " << value << std::endl;
  }
  MyInt(const MyInt& other) : value(other.value) { // Copy constructor
     std::cout << "Copy constructor called for " << value << std::endl;
  }
};

void process(MyInt obj) { // Pass by value, invokes the copy constructor
   std::cout << "Processing: " << obj.value << std::endl;
}

int main() {
  MyInt obj1(10);
  MyInt obj2 = obj1; // Calls the copy constructor
  process(obj1);     // Calls the copy constructor
   return 0;
}
```

◦ Constructor Initialization List: An efficient way to initialize member variables in the constructor. Members are initialized before the constructor body is executed.

C++
```cpp
#include <iostream>

class Rectangle {
public:
  int width;
  int height;
```

```
    Rectangle(int w, int h) : width(w), height(h) { // Constructor initialization list
        std::cout << "Rectangle constructor called." << std::endl;
    }

    int area() const { return width * height; }
};

int main() {
    Rectangle rect(5, 10);
    std::cout << "Area: " << rect.area() << std::endl;
    return 0;
}
```

- Delegating Constructors (C++11): Allows one constructor of a class to call another constructor of the same class to perform some or all of its initialization tasks. This helps to reduce code duplication.

C++
```
#include <iostream>
#include <string>

class Employee {
public:
    int id;
    std::string name;

    Employee() : Employee(0, "Unknown") { // Delegating to the other constructor
        std::cout << "Default Employee constructor called." << std::endl;
    }

    Employee(int empId, std::string empName) : id(empId), name(empName) {
        std::cout << "Parameterized Employee constructor called." << std::endl;
    }

    void display() const {
        std::cout << "ID: " << id << ", Name: " << name << std::endl;
    }
};
```

```
int main() {
    Employee emp1;
    emp1.display();
    Employee emp2(123, "Alice");
    emp2.display();
    return 0;
}
```

- Order of Initialization: Member variables are initialized in the order they are declared in the class definition, not in the order they appear in the constructor initialization list. It's good practice to list them in the initialization list in the same order as their declaration.
- Explicit Constructors (Example):

C++
```
#include <iostream>

class MyNumber {
private:
    int value;
public:
    explicit MyNumber(int val) : value(val) {
        std::cout << "Explicit constructor called with value: " << val << std::endl;
    }

    int getValue() const { return value; }
};

void processNumber(const MyNumber& num) {
    std::cout << "Processing number: " << num.getValue() << std::endl;
}

int main() {
    // processNumber(5); // Error: Implicit conversion from int to MyNumber is not allowed
because the constructor is explicit
    processNumber(MyNumber(5)); // OK: Explicit construction
     return 0;
}
```

- Copy Elision and Return Value Optimization (RVO): Compiler optimizations that can eliminate unnecessary copying of objects in certain situations (e.g., when returning a newly created object from a function).

- Destructors:
  - Explanation: Special member functions that are automatically called when an object of the class is destroyed (e.g., when it goes out of scope or is explicitly deleted). Destructors have the same name as the class with a tilde (~) prefix, take no arguments, and have no return type. They are typically used to release resources (like dynamically allocated memory) that were acquired by the object during its lifetime.
  - Example:

```C++
#include <iostream>

class ResourceHolder {
private:
  int* data;
public:
  ResourceHolder(int size) {
    data = new int[size];
    std::cout << "Resource allocated." << std::endl;
    }

    ~ResourceHolder() {
        deletedata;
        std::cout << "Resource deallocated." << std::endl;
    }
};

int main() {
    {
        ResourceHolder holder(10); // Constructor called when holder is created
    } // Destructor called when holder goes out of scope
     return 0;
}
```

- Virtual Destructors (Importance in Inheritance): When dealing with inheritance and polymorphism, if you delete a derived class object through a base class pointer, you need a virtual destructor in the base class. Otherwise, only the base class's destructor will be called, and the derived class's specific cleanup might not happen, leading to resource leaks.

C++
```
#include <iostream>

class Base {
public:
    virtual ~Base() {
        std::cout << "Base destructor called." << std::endl;
    }
};

class Derived : public Base {
public:
    ~Derived() override {
        std::cout << "Derived destructor called." << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Both Derived and Base destructors will be called because the base destructor
is virtual
     return 0;
}
```

- Rule of Three/Five/Zero:
  - Rule of Three (C++03): If a class needs a custom destructor, it likely also needs a custom copy constructor and a custom copy assignment operator to manage resources correctly (to avoid issues like double deletion or shallow copies when deep copies are needed).
  - Rule of Five (C++11): With the introduction of move semantics (for efficiency), if a class needs any of the destructor, copy constructor, copy assignment operator, move constructor, or move assignment operator, it likely needs all five to handle resource management properly.
  - Rule of Zero (C++11 onwards): The best practice is often to avoid manually managing

    resources by using RAII (Resource[3] Acquisition Is Initialization) principles, such as using smart pointers (like `std::unique_ptr` and `std::shared_ptr`) and standard library[4] containers. If you follow the Rule of Zero, you typically don't need to write custom destructors, copy/move constructors, or assignment operators.
  - `noexcept` Specifier: Indicates whether a function is expected to throw exceptions. Destructors should ideally not throw exceptions, and you can mark them as `noexcept`. This can help the compiler with optimizations and prevent unexpected program termination during stack unwinding.
- `this` Pointer:
  - Explanation: A special pointer that is implicitly available within the member functions of a non-static class. It points to the object for which the member function is being called.
  - Uses:
    - To access the object's own data members and member functions.[5]

        - To return a reference to the current object (often used in operator overloading for chaining).
        - To resolve naming conflicts between a member variable and a local variable or a parameter.
    - Example:

```
C++
#include <iostream>
#include <string>

class Person {
public:
   std::string name;
   int age;

   Person(std::string name, int age) : name(name), age(age) {}

   void printInfo() const {
      std::cout << "Name: " << this->name << ", Age: " << this->age << std::endl; // Using 'this' is
optional here
   }

   Person& celebrateBirthday() {
      this->age++;
      return *this; // Return a reference to the current object
   }
};

int main() {
   Person person1("Alice", 30);
   person1.printInfo();
   person1.celebrateBirthday().printInfo(); // Chaining using the returned reference
    return 0;
}
```

- Inheritance:

- Explanation: A mechanism in OOP that allows a new class (the derived class or subclass) to inherit properties (data members) and behaviors (member functions) from an existing class (the base class or superclass). Inheritance establishes an "is-a" relationship (e.g., a Dog is-a Animal).
- Base and Derived Classes ("is-a" relationship):

```
C++
#include <iostream>
#include <string>

class Animal { // Base class
public:
    std::string name;
    Animal(std::string n) : name(n) {
        std::cout << "Animal constructor called." << std::endl;
    }
    virtual void makeSound() const {
        std::cout << "Generic animal sound." << std::endl;
    }
    virtual ~Animal() {
        std::cout << "Animal destructor called." << std::endl;
    }
};

class Dog : public Animal { // Derived class inheriting from Animal
public:
    Dog(std::string n) : Animal(n) {
        std::cout << "Dog constructor called." << std::endl;
    }
    void makeSound() const override { // Overriding the base class method
        std::cout << "Woof!" << std::endl;
    }
    ~Dog() override {
        std::cout << "Dog destructor called." << std::endl;
    }
};

int main() {
  Dog myDog("Buddy");
  myDog.makeSound(); // Calls the Dog's makeSound method
   return 0;
}
```

- Access Specifiers in Inheritance (`public`, `protected`, `private` - Detailed Explanation):
  - `public` inheritance: Public members of the base class become public members of the derived class. Protected members of the base class become protected members of the derived class. Private members of the base class are inherited but are not directly accessible in the derived class (they can only be accessed through public or protected members of the base class).
  - `protected` inheritance: Public and protected members of the base class become protected members of the derived class. Private members remain inaccessible.
  - `private` inheritance: Public and protected members of the base class become private members of the derived class. Private members remain inaccessible. Private

inheritance is less common and often used for implementation details ("has-a" relationship).

- Order of Constructor and Destructor Calls: When a derived class object is created, the base class's constructor is called first, followed by the derived class's constructor. When a derived class object is destroyed, the derived class's destructor is called first, followed by the base class's destructor.
- Types of Inheritance:
  - Single Inheritance: A derived class inherits from only one base class.[6] (Example above with Dog inheriting from[7] Animal).
  - Multiple Inheritance: A derived class inherits from more than one base class.

```
#include <iostream> class Swimmer {
public:
    virtual void swim() const {
        std::cout << "Swimming." << std::endl;
    }
    virtual ~Swimmer() {}
};

class Walker {
public:
    virtual void walk() const {
        std::cout << "Walking." << std::endl;
    }
    virtual ~Walker() {}
};
```

```
class Frog : public Swimmer, public Walker { // Multiple inheritance
public:
    void makeSound() const {
        std::cout << "Croak!" << std::endl;
    }
};

int main() {
    Frog frog;
    frog.swim();
    frog.walk();
    frog.makeSound();
    return 0;
}
```

 * **Hierarchical Inheritance:** Multiple derived classes inherit from a single base class (e.g., `Cat` and `Dog` both inheriting from `Animal`).
 * **Multilevel Inheritance:** A derived class inherits from another derived class (e.g., `GrandChild` inheriting from `Child`, which inherits from `Parent`).
 * **Hybrid Inheritance:** A combination of different types of inheritance.

- Virtual Inheritance (Diamond Problem, Example): Occurs in multiple inheritance when two base classes inherit from a common grandparent class. This can lead to ambiguity if the derived class tries to access members of the grandparent class. Virtual inheritance uses the `virtual` keyword in the inheritance list to ensure that only one instance of the grandparent class is shared among the derived classes.

C++
```cpp
#include <iostream>

class Grandparent {
public:
    Grandparent() { std::cout << "Grandparent constructor." << std::endl; }
    virtual void someMethod() { std::cout << "Grandparent method." << std::endl; }
    virtual ~Grandparent() { std::cout << "Grandparent destructor." << std::endl; }
};

class Parent1 : public virtual Grandparent {
public:
    Parent1() { std::cout << "Parent1 constructor." << std::endl; }
    void someMethod() override { std::cout << "Parent1 method." << std::endl; }
    virtual ~Parent1() { std::cout << "Parent1 destructor." << std::endl; }
};

class Parent2 : public virtual Grandparent {
public:
    Parent2() { std::cout << "Parent2 constructor." << std::endl; }
    void someMethod() override { std::cout << "Parent2 method." << std::endl; }
    virtual ~Parent2() { std::cout << "Parent2 destructor." << std::endl; }
};

class Child : public Parent1, public Parent2 {
public:
    Child() { std::cout << "Child constructor." << std::endl; }
    void someMethod() override { std::cout << "Child method." << std::endl; }
    virtual ~Child() { std::cout << "Child destructor." << std::endl; }
};

int main() {
    Child child;
    child.someMethod(); // Calls Child's version
     return 0;
}
```

- Virtual Functions and the Virtual Table (vtable) and Virtual Pointer (vptr) (Mechanism of Dynamic Polymorphism): When a function in the base class is declared as `virtual`, it means that the derived classes can provide their own implementation of that function (using `override` in modern C++). The compiler creates a virtual table (vtable) for each class that has virtual functions. The vtable contains pointers to the virtual functions defined in that class. Each object of a class with virtual functions has a virtual pointer (vptr), which points to the vtable of its class. When a virtual function is called through a base class pointer or reference, the program uses the vptr to look up the correct function to call in the vtable of the actual object's type. This allows for dynamic polymorphism (deciding at runtime which function to execute).

- Covariant Return Types (C++11, Example):

C++
```cpp
#include <iostream>

class Animal {
public:
  virtual Animal* clone() const {
    std::cout << "Cloning an animal." << std::endl;
    return new Animal(*this);
  }
};

class Dog : public Animal {
public:
  Dog* clone() const override { // Covariant return type: Dog* instead of Animal*
     std::cout << "Cloning a dog." << std::endl;
    return new Dog(*this);
  }
  void bark() const { std::cout << "Woof!" << std::endl; }
};
```

```
int main() {
    Animal* animalPtr = new Dog();
    Dog* dogPtr = animalPtr->clone(); // Returns a Dog*
    dogPtr->bark();
    delete animalPtr;
    delete dogPtr;
    return 0;
}
```

- final Keyword (for virtual functions and classes):
  - final for virtual functions: Prevents derived classes from further overriding a virtual function.

    C++
    ```
    class Base {
    public:
        virtual void someMethod() { std::cout << "Base method." << std::endl; }
    };

    class Derived : public Base {
    public:
        void someMethod() override final { std::cout << "Derived method." << std::endl; }
    };

    class GrandDerived : public Derived {
    public:
        // void someMethod() override { ... } // Error: Cannot override final method
    };
    ```
  - final for classes: Prevents a class from being inherited from.

    C++
    ```
    class FinalClass final {
    public:
        void doSomething() { std::cout << "Doing something." << std::endl; }
    };
    ```

```
// class AnotherClass : public FinalClass {}; // Error: Cannot inherit from final class
```

- Abstract Classes and Concrete Classes:
  - Abstract Class: A class that contains at least one pure virtual function. Abstract classes cannot be instantiated (you cannot create objects of an abstract class). They serve as interfaces, defining a contract that derived classes must fulfill.
  - Pure Virtual Function: A virtual function declared with `= 0` in the base class (e.g., `virtual void draw() = 0;`).
  - Concrete Class: A class that is not abstract. It can be instantiated, and it provides implementations for all inherited pure virtual functions.

- Polymorphism:
  - Explanation: The ability of objects of different classes to respond to the same message (function call) in their own way. It allows you to write more flexible and extensible code.
  - Compile-time (Static) Polymorphism: Determined at compile time. Achieved through

function overloading and operator overloading (already explained in Fundamentals).

- Run-time (Dynamic) Polymorphism: Determined at runtime. Achieved through virtual functions and inheritance (explained above).
- Slicing: When a derived class object is assigned to a base class object (not a pointer or reference), the extra information (data members) of the derived class is "sliced off," and you are left with only the base class part of the object. This can lead to loss of information and unexpected behavior if you rely on the derived class's specific members. Always use pointers or references to the base class to avoid slicing when working with polymorphism.

C++
```
#include <iostream>
#include <string>

class Base {
public:
  std::string baseData = "Base";
  virtual void print() const { std::cout << "Base print: " << baseData << std::endl; }
};

class Derived : public Base {
```

```
public:
    std::string derivedData = "Derived";
    void print() const override { std::cout << "Derived print: " << baseData << ", " << derivedData
<< std::endl; }
};

int main() {
    Derived d;
    Base b = d; // Object slicing occurs here
    b.print();   // Calls Base::print, derivedData is lost
     return 0;
}
```

- Run-Time Type Identification (RTTI) (`typeid`, `dynamic_cast`): Allows you to determine the type of an object at runtime.
    - `typeid`: Returns a `std::type_info` object representing the type of an object. You can compare these objects to check if two objects are of the same type.
    - `dynamic_cast`: Used for safe downcasting in inheritance hierarchies. It checks the actual type of the object at runtime. If the cast is valid, it returns a pointer to the derived type; otherwise, it returns `nullptr` (for pointer types). For reference types, it throws a `std::bad_cast` exception if the cast fails. RTTI can have some performance overhead, so use it judiciously.

- Friend Functions and Friend Classes:
  - Friend Function: A non-member function that is granted access to the private and protected members of a class. Declared within the class using the `friend` keyword.

```
C++
#include <iostream>

class Box {
private:
    double width;
public:
    Box(double w) : width(w) {}
    friend void printWidth(const Box& box); // Friend function declaration
};

void printWidth(const Box& box) { // Friend function definition (not a member of Box)
    std::cout << "Width of box: " << box.width << std::endl; // Can access private member
}

int main() {
    Box box(10.0);
    printWidth(box);
    return 0;
}
```

  - Friend Class: A class that is granted access to the private and protected members of another class. Declared within the other class using the `friend` keyword.

```
C++
#include <iostream>

class ClassA {
private:
    int valueA;
public:
    ClassA(int val) : valueA(val) {}
    friend class ClassB; // ClassB is a friend of ClassA
};

class ClassB {
public:
    void displayValueA(const ClassA& objA) {
        std::cout << "Value of valueA in ClassA: " << objA.valueA << std::endl; // Can access private
member
    }
};

int main() {
    ClassA objA(5);
    ClassB objB;
    objB.displayValueA(objA);
    return 0;
}
```

- Shallow Copy vs. Deep Copy (Detailed Example):
    - Shallow Copy: When an object is copied, and the new object's members point to the same memory locations as the original object's members (especially for pointers to dynamically allocated memory). This can lead to issues like double deletion (when both objects try to deallocate the same memory) or unintended side effects (when modifying the data through one object affects the other). The default copy constructor and default assignment operator in C++ perform shallow copies.

- Deep Copy: When an object is copied, and if the object contains pointers to dynamically allocated memory, new memory is allocated for the copy, and the data is copied from the original object's memory to the new memory. This ensures that the original and the copy are independent.
- Example (Revisited):

```
C++
#include <iostream>
#include <cstring>

class StringWrapper {
private:
  char* data;
  int length;
public:
  // Constructor
  StringWrapper(const char* str = "") : length(std::strlen(str)) {
    data = new char[length + 1];
    std::strcpy(data, str);
    std::cout << "Constructor called for: " << data << std::endl;
  }

  // Destructor
  ~StringWrapper() {
    std::cout << "Destructor called for: " << (data ? data : "(null)") << std::endl;
    deletedata;
  }

  // Getter
   const char* getString() const {
    return data;
  }

  // Deep Copy Constructor
  StringWrapper(const StringWrapper& other) : length(other.length) {
    data = new char[length + 1];
    std::strcpy(data, other.data);
    std::cout << "Deep copy constructor called for: " << data << std::endl;
  }

  // Assignment Operator (for deep copy)
  StringWrapper& operator=(const StringWrapper& other) {
    std::cout << "Assignment operator called." << std::endl;
    if (this != &other) { // Prevent self-assignment
        deletedata; // Release existing memory
        length = other.length;
        data = new char[length + 1];
        std::strcpy(data, other.data);
    }
    return *this;
  }
};

void printString(StringWrapper s) { // Pass by value will invoke the copy constructor
   std::cout << "String inside function: " << s.getString() << std::endl;
}

int main() {
  {
    StringWrapper str1("Hello");
    printString(str1); // Deep copy constructor is called

    StringWrapper str2 = str1; // Deep copy constructor is called
      std::cout << "String in str1: " << str1.getString() << std::endl;
    std::cout << "String in str2: " << str2.getString() << std::endl;

    StringWrapper str3("World");
    str3 = str1; // Assignment operator is called
      std::cout << "String in str3: " << str3.getString() << std::endl;
  } // Destructors for str3, str2, and str1 will be called
   return 0;
}
```

I will continue with the next section, Memory Management in C++, in the subsequent response. This level of detail for each topic will take some time and multiple responses. Please let me know if you have any questions about the OOP concepts explained so far.

Sources
1.
https://medium.com/@pathakvikas034/understanding-object-oriented-programming-oop-in-python-f50674eda1ab
2. https://github.com/mannnisingh/text12
3. https://cbceskillindia.com/blog/1367/concepts-of-object-oriented-programming-in-english
4. https://github.com/nickthelegend/CPP-Object-Oriented-Programming
5. https://github.com/Abdullah-Niaz/CPP-Complete-Course
6. https://github.com/hong539/coding_collects
7. https://github.com/enthusiasticgeek/common_coding_ideas
8. https://github.com/plops/cl-cpp-generator2/blob/master/README_guidlines.md
9. https://cplusplus.com/forum/beginner/168535/
10. https://aleksandarhaber.com/introduction-and-easy-explanation-of-inheritance-in-c/
11. https://allprogramingcodes.blogspot.com/2023/02/what-are-different-types-of-visibility.html
12. https://slideplayer.com/slide/16851545/
13. https://quizlet.com/837651803/cs341-exam-butler-university-dr-ryan-flash-cards/