MQF633 C++ FOR FINANCIAL ENGINEERING

Lecture 7: Smart Pointer, Type Casting and Design Pattern

Part I: Smart Pointer

Memory leak using raw pointer without delete

Let see a very simple example, and result of running this code is:

```
// C++ program to demonstrate working of a Pointers

#include <iostream>
using namespace std;

class Rectangle {
private:
	int length;
	int breadth;
};

void fun()
{
	// By taking a pointer p and dynamically creating object of class rectangle
	Rectangle* p = new Rectangle();
}

int main()
{
	// Infinite Loop
	while (1) {
		fun();
	}
}
```

Result of running above code:

Memory limit exceeded

A modified code like below will solve the issue of memory leak.

```
// C++ program to demonstrate working of a Pointers

#include <iostream>
#include <vector>

using namespace std;

class Rectangle {
private:
        int length;
        int breadth;
};

void some_complex_function(Rectangle* ptr)
{
    // By taking a pointer p and do some complex operations
}

int main()
{
        Rectangle* p = new Rectangle();
        some_complex_function(p); // if some error happened in this function, then there will be pointer p not deleted, causing memory leak
        delete p;
 }
```

Refer to example code of memory_leak.cpp

Look deeper in new and delete

Refer to example code of memory.cpp

Parsing pointer into other function

Parsing raw pointer to function without transferring ownership

Parse as const *
No need to delete pointer in function

Parsing raw pointer to function and transfer ownership

Not parse as const *
Need to delete raw pointer inside the function

Refer to code example parsing_ptr.cpp to function.

Dangling pointer

A dangling pointer is a pointer that points to a memory location that has been deallocated or freed. This can happen when the memory previously pointed to by the pointer is released, but the pointer itself is not updated to reflect this change. As a result, the pointer still contains the address of the deallocated memory, which is now invalid. Accessing or dereferencing such a pointer can lead to undefined behaviour, as the memory it points to may have been reallocated to another use, or the memory may no longer be accessible by the program.

Dangling pointers are a common source of bugs in C and C++ programs, particularly in scenarios involving dynamic memory allocation and deallocation. They can cause crashes, memory corruption, or other unexpected behaviour in the program. Proper management of pointers, including setting pointers to nullptr after deallocation and avoiding accessing pointers after the memory they point to has been freed, is crucial for avoiding dangling pointer issues.

Point to variable out of scope

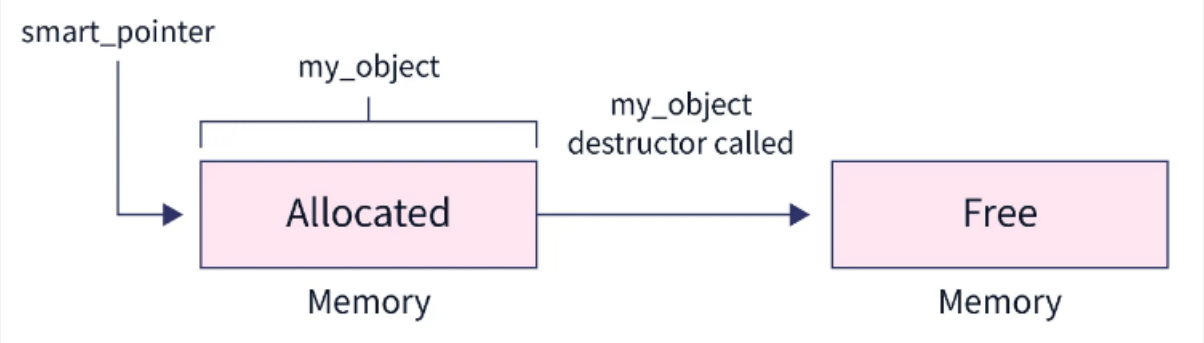
Return pointer from function

When the local variable is not static and the function returns a pointer to that local variable. The pointer pointing to the local variable becomes dangling pointer.

Refer to dangling_ptr.cpp

Smart Pointer

In modern C++ programming, the Standard Library includes smart pointers, which are used to help ensure that programs are free of memory and resource leaks and are exception-safe.



Syntax: smart_pointer_type<data_type>pointer_name(new data_type());

Smart pointer types

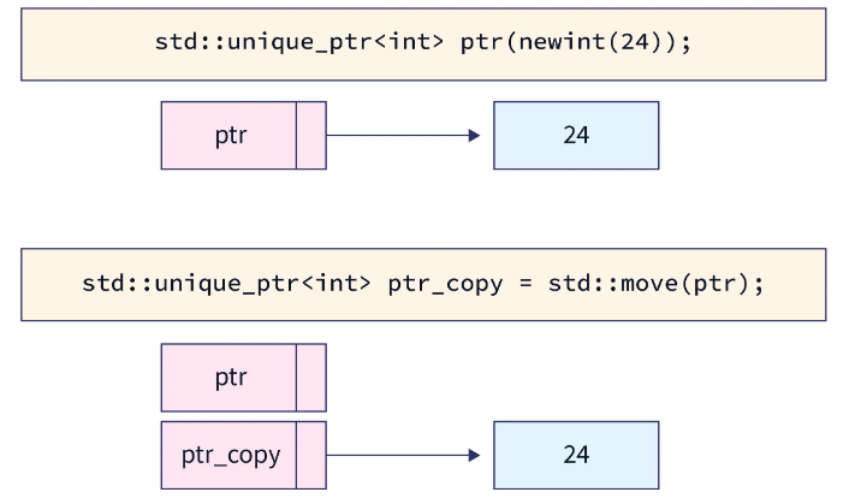
unique_ptr, shared_ptr, or weak_ptr and auto_ptr (not recommended)

unique_ptr

The unique_ptr is one of the most common smart pointers. If you are not sure which smart pointer to use, this is the pointer you should go with.The unique_ptr, as the name suggests, holds a unique pointer, i.e., the memory held by this pointer cannot be represented by any other pointer. This will be the only pointer that holds that memory. You will not be allowed to create another copy of the pointer. While the memory held by the pointer cannot be copied or shared by any other pointer, you can always move it to a new pointer. This can be achieved by the std::move() function of the C++ Standard Template Library.

Syntax: unique_ptr<A> p1(new A());

1. unique_ptr<A>: It specifies the type of the std::unique_ptr. In this case- an object of type A.
2. new A: An object of type A is dynamically allocated on the heap using the new operator.
3. p1: This is the name of the std::unique_ptr variable.



Refer to unique_ptr.cpp example

In this example given above, we create a new pointer variable ptr for an integer value of 13 within the function called my_func(). Then we perform some operations and then move the unique pointer to a new variable called ptr2. And then we check a condition called $x == 45$ and when it evaluates to be true we just return from the function. We don't bother to free the memory we created, because that will be automatically taken care of by the smart pointer.

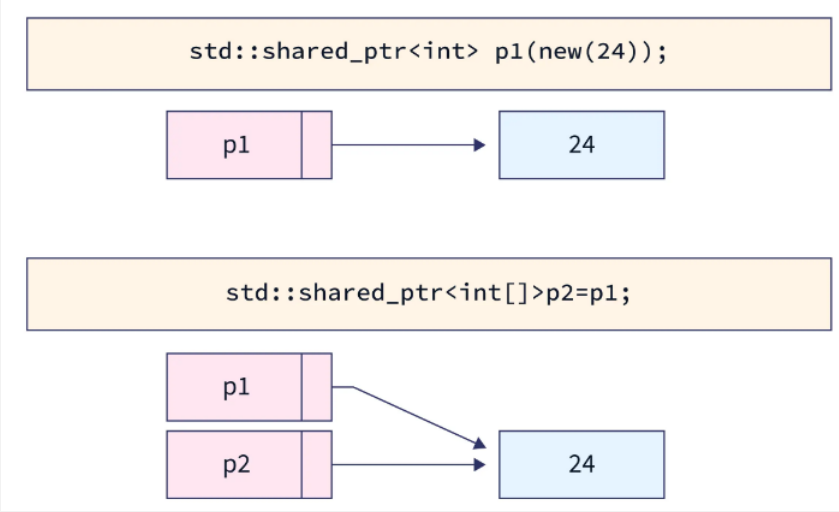
Due to unique_ptr cannot share the ownership of same underlying data, we need to use std::move<>() to transfer to ownership from ptr1 into ptr2.

shared_ptr

The unique_ptr will not always be enough for all your use cases. You might have some time required to make a copy of the pointer to pass it to another function. So to help you in those situations, the shared_ptr was developed. The shared_ptr allows you to make a copy of the pointer. It will hold the memory until all the pointer holding that memory gets out of scope. This is done by maintaining a reference counter.

The Reference counter will hold the count of pointers pointing to that memory location. The destructor will check the reference counter and free the memory only if the reference counter value is 1, i.e., only the current pointer is pointing to that memory. You can also revoke the ownership you hold over the memory that the pointer holds with the reset method. In that case, the reference count will be decreased by 1, representing that there is one less owner for that memory location. At any point in time if you need to know the number of pointers pointing to a location you can use the use_count() function to get that.

Syntax: shared_ptr<A> p1(new A());



Refer to shared_ptr.cpp example

In this example given above, we create 2 shared pointers named p1 and p2 for a class called Articles. And we work with them. The memory gets freed only when both the variables p1 and p2 goes out of scope.

Circular reference with shared_ptr.

The classic example of circular references is where you have two classes A and B, where A has a member of shared_ptr<B> and class B has member as shared_ptr<A>. When we create object1 of class A and object2 of class B, and let them holding the share_ptr of each other, then both object 1 and object 2 will not be released even they are out of scope. This will leads to some potential memory leak issue.

How we solve this issue? Simple answer is to use weak_ptr, but a good practice is always try into avoid creating the circular reference case from happening. Holding pointer of some object means the program running time ownership.

weak_ptr

A weak pointer is a type of smart pointer in C++ that provides a non-owning reference to an object that is managed by a shared pointer. Unlike a shared pointer, a weak pointer does not increment the reference count of the managed object, so it does not affect the lifetime of the object. The primary purpose of a weak pointer is to provide a way to access the managed object of a shared pointer without extending its lifetime. This is useful in situations where you need to temporarily access the object, but you don't want to keep it alive longer than necessary. For example, a weak pointer can be used to check if the object still exists before attempting to access it, or to break a cyclic reference between two shared pointers. A weak pointer can be created from a shared pointer using the std::weak_ptr class template. It provides a lock method that returns a shared pointer to the managed object if it still exists, or an empty shared pointer if the object has been deleted. The expired method can be used to check if the managed object has been deleted.

Syntax: weak_ptr<A> p1(new A);

Refer to code example weak_ptr.cpp

A few key points:

- Use weak_ptr.lock() to check if the object is still exists, it will return a shared pointer if yes, and nullptr if not.
- Reset() function is to reset the ref_count() of share_ptr.

Auto_ptr

The auto_ptr is the first implementation of the smart pointers. This was similar to that of the unique_ptr. However, when the new standard was defined in C++ 11, The auto_ptr was dropped and replaced with the three-pointers mentioned above.

Guidelines to Follow When Working with Smart Pointers in C++

- Always try to use smart pointers in C++ because it is better to be sure that you won't be running out of memory due to a memory leak.
- Use Unique_ptr if you are not sure to use which of the smart pointer to use.
- Use Shared_ptr only when dealing with multiple pointers or threads where you will need to share the location.
- If you want just to examine an object and not gonna work with it on any serious level, you can then use the Weak_ptr.
- Try to reduce the usage of raw pointers as much as possible, and when you do, make sure that you properly free the memory held by the pointer.

Conclusion

- Smart pointers in C++ programming is an idea to compensate for the absence of garbage collectors.
- With Smart Pointers in C++, you can forget about the nightmare of running out of memory because of memory, cause you forgot to free the memory after the execution of a program.
- There are three common types of Smart Pointers in C++ they are Unique_ptr, Shared_ptr, and Weak_ptr.
- Use Unique_ptr when not sure which of the smart pointer to use.
- Use Shared_ptr only when you need to share the pointer to other functions or threads.
- Use Weak_ptr when you don't want the memory to be held by the pointer just for this one pointer.
- Never use Auto_ptr. It is always better to use the updated and widely used Smart Pointers.
- Try to reduce the usage of standard raw pointers as much as possible in your programs.

Quiz for Part I:

1. Which of the following is an advantage of using smart pointers over raw pointers?
    a. Smart pointers are faster
    b. Smart pointers simplify memory management and help prevent memory leaks
    c. Smart pointers are more lightweight
    d. Smart pointers allow direct access to memory addresses

2. Which of the following is not a type of smart pointer in C++?
    a) std::auto_ptr
    b) std::unique_ptr
    c) std::reference_ptr
    d) std::shared_ptr

3. What is the difference between std::unique_ptr, std::shared_ptr, and std::weak_ptr?
    a) They are all identical
    b) std::unique_ptr allows only one pointer to an object, std::shared_ptr allows multiple pointers with shared ownership, and std::weak_ptr provides a non-owning reference
    c) std::unique_ptr allows multiple pointers with shared ownership, std::shared_ptr allows only one pointer to an object, and std::weak_ptr provides a non-owning reference
    d) std::unique_ptr is deprecated, std::shared_ptr allows only one pointer to an object, and std::weak_ptr provides a weak reference to an object

4. When should you use std::unique_ptr?
    a. When you need multiple pointers to an object
    b. When you need a non-owning reference to an object
    c. When you need exclusive ownership of an object
    d. When you need to create a circular reference

5. When should you use std::shared_ptr?
    a) When you need exclusive ownership of an object
    b) When you need a non-owning reference to an object
    c) When you need to create a circular reference
    d) When you need multiple pointers with shared ownership

6. How does std::weak_ptr help prevent circular references?
    a) By automatically deleting the pointer

b) By providing a non-owning reference to an object
c) By creating a strong reference to an object
d) By invalidating the pointer

Part II: Type casting for C++

Type casting in C++ refers to the process of converting one data type into another. This can be useful when you want to perform operations that involve different types or when you need to ensure compatibility between different data types. There are two main types of type casting in C++:

1. Implicit Type Conversion (Automatic Type Conversion): This occurs when the compiler automatically converts one type of data into another type without any explicit instruction from the programmer. Implicit type conversion usually happens when:
2. Assigning a value of one data type to a variable of another data type.
3. When arithmetic or relational operations are performed between different data types.

```
int num_int = 10;
double num_double = num_int; // Implicitly converts int to double
int new_int = num_int * num_double; //implicitly convert to int
```

4. Explicit Type Conversion (Type Casting): This involves the programmer explicitly instructing the compiler to convert one data type into another. Explicit type casting is done using casting operators such as **static_cast, dynamic_cast, reinterpret_cast, or const_cast**. This is useful when you want to override the default behavior of implicit type conversion or when you need to perform a conversion that might result in data loss.

Here's an example of explicit type conversion using static_cast:

```
double another_double = 3.14;
int another_int = static_cast<int>(another_double); // Explicitly converts double to int
```

Static casting on user defined types

In C++, static_cast is a type of casting operator used for explicit type conversions. It allows you to convert one data type into another in a type-safe manner. static_cast is generally used for conversions that are well-defined and safe at compile time, such as converting between related types, like integers and floating-point numbers, or performing upcasts in inheritance hierarchies.

Syntax: result_type result = static_cast<result_type>(expression);

Where:

- result_type is the type you want to convert to.
- expression is the value or expression you want to convert.

Refer to static_casting.cpp

In this example: We use static_cast to convert an integer num_int to a double num_double. We demonstrate the usage of static_cast for converting a pointer to the base class Base to a pointer to the derived class Derived. This is known as an upcast in inheritance. We ensure that the conversion is safe because we know that the actual object pointed to by base_ptr is of type Derived.

Dynamic casting on user defined types

In C++, dynamic_cast is a type of casting operator primarily used for performing safe downcasting in inheritance hierarchies. Downcasting refers to converting a base class pointer or reference to a derived class pointer or reference. Unlike static_cast, which performs only compile-time type checking, dynamic_cast performs both compile-time and runtime type checking, ensuring that the conversion is safe.

result_type result = dynamic_cast<result_type>(expression);

Where:

- result_type is the type you want to convert to.
- expression is the value or expression you want to convert.

If the conversion is not possible or safe, dynamic_cast returns a null pointer (for pointer types) or throws a std::bad_cast exception (for reference types) when used inappropriately.

Refer to dynamic_catsting.cpp for an example demonstrating the usage of dynamic_cast:

In this example:

- We have a base class Base and a derived class Derived.
- We create a base class pointer base_ptr pointing to an object of the derived class Derived.
- We use dynamic_cast to attempt downcasting the base pointer to a derived pointer derived_ptr. If the downcast is successful, we safely call the print() function of the derived class through the derived pointer. If the downcast fails (for example, if the object pointed to by base_ptr is not of type Derived), dynamic_cast returns a null pointer.

It's important to note that dynamic_cast only works with polymorphic classes, i.e., classes that have at least one virtual function.

reinterpret_cast

In C++, reinterpret_cast is a type of casting operator that allows you to convert one pointer type to another pointer type, even if they are unrelated. It performs a low-level reinterpretation of the bit pattern of the pointer, essentially treating the object as if it were of a different type. It should be used with caution because it can lead to undefined behaviour if misused.

Syntax: result_type result = reinterpret_cast<result_type>(expression);

Where:
result_type is the type you want to convert to.
expression is the value or expression you want to convert.
reinterpret_cast is mainly used for converting between pointer types or between pointer and integral types, especially when dealing with low-level programming, such as working with hardware registers or memory-mapped I/O.

```
#include <iostream>

int main() {
    int number = 42;

    // Casting an integer pointer to a pointer to char
    char* char_ptr = reinterpret_cast<char*>(&number);

    // Outputting the individual bytes of the integer
    for (size_t i = 0; i < sizeof(int); ++i) {
        std::cout << "Byte " << i + 1 << ": " << static_cast<int>(*(char_ptr + i)) << std::endl;
    }

    return 0;
}
```

In this example:
We have an integer variable number with a value of 42.
We use reinterpret_cast to convert a pointer to number (of type int*) to a pointer to char (of type char*).
We then iterate over each byte of the integer using the char pointer and output the value of each byte. This demonstrates how the individual bytes of the integer are stored in memory, with reinterpret_cast allowing us to treat the integer as a sequence of characters.

const_cast

In C++, const_cast is a type of casting operator used to remove the constness or volatility of a variable. It allows you to cast away the constness of a variable and modify its value, potentially enabling dangerous operations if used incorrectly.

Syntax const_cast<new_type>(expression)

```
#include <iostream>

class MyClass {
private:
    int value;
public:
    MyClass(int v) : value(v) {}

    // Getter method
    int getValue() const {
        return value;
    }

    // Setter method
    void setValue(int v) {
        value = v;
    }
};

int main() {
    const MyClass obj(10); // Creating a const object

    // Attempting to modify the const object directly will result in a compilation error
    // obj.setValue(20); // Error: member function 'setValue' is declared const

    // Using const_cast to remove constness and modify the object
    MyClass& nonConstObj = const_cast<MyClass&>(obj);
    nonConstObj.setValue(20); // Valid

    std::cout << "Value after modification: " << obj.getValue() << std::endl; // Output: 20

    return 0;
}
```

Quiz for part II

1. What is type casting in C++?
   a) Changing the type of a variable at runtime
   b) Converting a variable from one data type to another
   c) Converting a variable from one memory location to another
   d) Changing the name of a variable

2. What is the purpose of dynamic_cast?
   a) It performs arithmetic operations
   b) It converts a base class pointer to a derived class pointer
   c) It converts a derived class pointer to a base class pointer
   d) It converts between pointers and references

3. What is the purpose of reinterpret_cast?
   a) It performs arithmetic operations
   b) It converts between pointers and references of unrelated types
   c) It converts between pointers and references of related types
   d) It converts between integer and floating-point types

4. Which of the following is an unsafe type cast?
   a) static_cast
   b) dynamic_cast
   c) const_cast
   d) reinterpret_cast

5. Which cast is used to convert between pointers or references of unrelated types?
   a) static_cast
   b) dynamic_cast
   c) const_cast
   d) reinterpret_cast

Part III: C++ Design Pattern

What is design pattern

A design pattern in C++ is a generic repeatable solution to a frequently occurring problem in software design that is used in software engineering. It isn't a complete design that can be written in code right away. It is a description or model for problem-solving that may be applied in a variety of contexts.

Type of design pattern

- Creational Design Pattern: factory, builder, prototype, singleton …
- Structural Design Pattern: adapter, decorator, proxy …
- Behavioural Design Pattern: command, iterator …

Creational Patterns

Generally, a system has information about its object's creation, composition, and representation. Creational patterns have handled everything from object creation to its representation.

- The system does not need to worry about how its objects are created, represented, etc. Such creational patterns are known as object creational patterns. That delegates object instantiation to another object.
- In addition to this, there are creational patterns that can also manage the instantiation process of classes by using the concept of inheritance. These patterns are called creational patterns, out of these two patterns, more importance is given to object creational patterns. They define a small set of fundamental behaviours that can perform more complex tasks rather than defining a fixed set of behaviours.
- Creational patterns contain information about concrete classes that are mostly used by the system.
- They do not provide information about how different instances of concrete classes are created and how they are combined together.
- The only thing the system knows about its objects is the interfaces of objects that are declared by the abstract classes.

Factory Method Pattern

Factory Method Pattern provides an interface for creating objects but **leaves the actual object instantiation to derived classes**. This allows for flexibility in object creation and promotes **loose coupling between the creator (client code) and the concrete products**. In simple terms, the Factory Method Pattern is like a recipe for making things, but it lets different chefs (subclasses) create their own unique versions of those things while sticking to the same basic cooking instructions (the Factory Method).

Core Components of the Factory Method Pattern in C++
Let's break down the key participants of the Factory Method Pattern:

1. **Creator** (Abstract Creator)
Description: An abstract class or interface responsible for declaring the factory method, which returns an object of a product type. The Creator Class can also contain other method that rely on the factory method to create objects.
Purpose: Provides a common interface for creating objects, but the specific object creation is defereed to concrete creators.

2. **Concrete Creator**:
Description: Concrete subclasses of the Creator. Each concrete creator implements the factory method declared in the abstract creator, returning a specific type of product.
Purpose: Responsible for creating instances of concrete products, thus encapsulating the actual object creation logic.

3. **Product** (Abstract Product):
Description: The abstract class or interface for the objects the factory method creates. It defines the common interface that all products must implement.
Purpose: Specifies the interface that concrete products must adhere to, ensuring that they have a consistent set of methods.

4. **Concrete Product**:
Description: Concrete implementations of the Product interface. Each concrete product represents a distinct type of object.
Purpose: Defines the actual objects that the factory method creates. Each concrete product has its own implementation of the methods specified in the Product interface.

5. **Client**:
Description: The client code interacts with the Creator through the abstarct Creator class and relies on the factory method to create instances of products.
Purpose: Utilizes the factory method pattern to create objects without needing to know specific class of objects being created, promoting flexibility and decoupling between the client and product classes.

Let's see an example.

```
// Abstract product class
class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {} // Virtual destructor for polymorphism
};
// Concrete product class - Circle
class Circle : public Shape {
public:
    void draw() override {
        std::cout << “Drawing a Circle” << std::endl;
    }
};

// Concrete product class - Square
class Square : public Shape {
public:
    void draw() override {
        std::cout  << “Drawing a Square” << std::endl;
    }
};
```

```
// Abstract creator class
class ShapeFactory {
public:
    virtual Shape* createShape() = 0;
    virtual ~ShapeFactory() {} // Virtual destructor for polymorphism
};

// Concrete creator class - CircleFactory
class CircleFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        return new Circle();
    }
};

// Concrete creator class - SquareFactory
class SquareFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        return new Square();
    }
};
```

```
int main() {
    ShapeFactory* circleFactory = new CircleFactory();
    ShapeFactory* squareFactory = new SquareFactory();

    Shape* circle = circleFactory->createShape();
    Shape* square = squareFactory->createShape();

    Circle->draw(); // Output: Drawing a Circle
    Square->draw(); // Output: Drawing a Square

    delete circleFactory;
    delete squareFactory;
    delete circle;
    delete square;

    return 0;
}
```

Builder design pattern

The logic behind the Builder Pattern is to separate the construction of an object from its representations. This separation basically enables us to create different representations of an object using the same construction process.

Key components of the Builder Pattern in C++

- **Director**: The Director is the main component of the builder pattern it basically responsible for the construction process of the program. It works with a Builder to build an object. The Director knows the actual steps required to build an object, but it does not know the details of how each step is implemented.
- **Builder**: The Builder is the main interface or an abstract class that defines the construction steps required to create an object.
- **Concrete Builder**: Basically, these are the classes that implement the Builder interface. Each Concrete Builder is responsible for constructing a particular variant of the object.
- **Product**: The Product is the complex object that we want to create. The Product class may have methods to access or manipulate these components. It often has multiple parts or components that are built by the Builder.

Let’s see an example like below.

```
#include <iostream>
#include <string>

// Product class
class Computer {
public:
	void setCPU(const std::string& cpu) {
		cpu_ = cpu;
	}

	void setMemory(const std::string& memory) {
		memory_ = memory;
	}

	void setStorage(const std::string& storage) {
		storage_ = storage;
	}

	void display() {
		std::cout << "CPU: " << cpu_ << std::endl;
		std::cout << "Memory: " << memory_ << std::endl;
		std::cout << "Storage: " << storage_ << std::endl;
	}

private:
	std::string cpu_;
	std::string memory_;
	std::string storage_;
};
```

```
// continue ..

// Builder interface
class ComputerBuilder {
public:
	virtual void buildCPU(const std::string& cpu) = 0;
	virtual void buildMemory(const std::string& memory) = 0;
	virtual void buildStorage(const std::string& storage) = 0;
	virtual Computer getResult() = 0;
};

// Concrete Builder
class DesktopComputerBuilder : public ComputerBuilder {
public:
	DesktopComputerBuilder() {
		computer_ = Computer();
	}

	void buildCPU(const std::string& cpu) override {
		computer_.setCPU(cpu);
	}

	void buildMemory(const std::string& memory) override {
		computer_.setMemory(memory);
	}

	void buildStorage(const std::string& storage) override {
		computer_.setStorage(storage);
	}

	Computer getResult() override {
		return computer_;
	}

private:
	Computer computer_;
};
```

```
// Director
class ComputerAssembler {
public:
	Computer assembleComputer(ComputerBuilder& builder) {
		builder.buildCPU("Intel i7");
		builder.buildMemory("16GB");
		builder.buildStorage("512GB SSD");
		return builder.getResult();
	}
};

int main() {
	DesktopComputerBuilder desktopBuilder;
	ComputerAssembler assembler;
	Computer desktop = assembler.assembleComputer(desktopBuilder);

	std::cout << "Desktop Computer Configuration:" << std::endl;
	desktop.display();

	return 0;
}
```

Singleton pattern

A singleton pattern is a creational design pattern that ensures that **only one instance of a class can exist in the entire program**. This means that if you try to create another instance of the class, it will return the same instance that was created earlier. The Singleton pattern is useful when we need to have only one instance of a class, for example, a single database connection shared by multiple objects as creating a separate database connection for every object may be costly.

```
#include <iostream>

class Singleton {
public:
	// Static method to access the singleton instance
	static Singleton& getInstance() {
		// If the instance doesn't exist, create it
		if (!instance) {
			instance = new Singleton();
		}
		return *instance;
	}
	// Public method to perform some operation
	void someOperation() {
		std::cout
			<< "Singleton is performing some operation." << std::endl;
	}
	// Delete the copy constructor and assignment operator
	Singleton(const Singleton&) = delete;
	Singleton& operator=(const Singleton&) = delete;
private:
	// Private constructor to prevent external instantiation
	Singleton() {
		std::cout << "Singleton instance created." << std::endl;
	}
	// Private destructor to prevent external deletion
	~Singleton() {
		std::cout << "Singleton instance destroyed." << std::endl;
	}

	// Private static instance variable
	static Singleton* instance;
};

// Initialize the static instance variable to nullptr
Singleton* Singleton::instance = nullptr;
```

In the client code

```
int main()
{
    // Access the Singleton instance
    Singleton& singleton = Singleton::getInstance();

    // Use the Singleton instance
    singleton.someOperation();

    // Attempting to create another instance will not work
    // Singleton anotherInstance; // This line would not
    // compile

    return 0;
}
```

Decorator Pattern

The Decorator Pattern is a structural design pattern in software engineering that enables the **dynamic addition of new behaviours or responsibilities to individual objects without altering their underlying class structure**. It achieves this by creating a set of decorator classes that are used to wrap concrete components, which represent the core functionality.

- Decorators conform to the same interface as the components they decorate, allowing them to be used interchangeably.
- This pattern promotes flexibility and extensibility in software systems by allowing developers to compose objects with different combinations of functionalities at runtime.
- It adheres to the open/closed principle, as new decorators can be added without modifying existing code, making it a powerful tool for building modular and customizable software components.
- The Decorator Pattern is commonly used in scenarios where a variety of optional features or behaviors need to be added to objects in a flexible and reusable manner, such as in text formatting, graphical user interfaces, or customization of products like coffee or ice cream.

1. **Component Interface**

This is an abstract class or interface that defines the common interface for both the concrete components and decorators. It specifies the operations that can be performed on the objects.

2. **Concrete Component**

These are the basic objects or classes that implement the Component interface. They are the objects to which we want to add new behavior or responsibilities.

3. **Decorator**

This is an abstract class that also implements the Component interface and has a reference to a Component object. Decorators are responsible for adding new behaviors to the wrapped Component object.

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

// Component interface - defines the basic ice cream operations.
class IceCream {
public:
	virtual string getDescription() const = 0;
	virtual double cost() const = 0;
};

// Concrete Component - the basic ice cream class.
class VanillaIceCream : public IceCream {
public:
	string getDescription() const override {
		return "Vanilla Ice Cream";
	}
	double cost() const override { return 160.0; }
};

// Decorator - abstract class that extends IceCream.
class IceCreamDecorator : public IceCream {
protected:
	IceCream* iceCream;
public:
	IceCreamDecorator(IceCream* ic) : iceCream(ic) {}

	string getDescription() const override {
		return iceCream->getDescription();
	}

	double cost() const override {
		return iceCream->cost();
	}
};
```

```
// Concrete Decorator - adds chocolate topping.
class ChocolateDecorator : public IceCreamDecorator {
public:
    ChocolateDecorator(IceCream* ic) : IceCreamDecorator(ic) { }

    string getDescription() const override     {
        return iceCream->getDescription() + " with Chocolate";
    }

    double cost() const override {
        return iceCream->cost() + 100.0;
    }
};

// Concrete Decorator - adds caramel topping.
class CaramelDecorator : public IceCreamDecorator {
public:
    CaramelDecorator(IceCream* ic) : IceCreamDecorator(ic) {}

    string getDescription() const override{
        return iceCream->getDescription() + " with Caramel";
    }

    double cost() const override {
        return iceCream->cost() + 150.0;
    }
};
```

```
int main()
{
   // Create a vanilla ice cream
   IceCream* vanillaIceCream = new VanillaIceCream();
   cout << "Order: " << vanillaIceCream->getDescription()
        << ", Cost: Rs." << vanillaIceCream->cost()
        << endl;

   // Wrap it with ChocolateDecorator
   IceCream* chocolateIceCream
      = new ChocolateDecorator(vanillaIceCream);
   cout << "Order: " << chocolateIceCream->getDescription()
        << ", Cost: Rs." << chocolateIceCream->cost()
        << endl;

   // Wrap it with CaramelDecorator
   IceCream* caramelIceCream
      = new CaramelDecorator(chocolateIceCream);
   cout << "Order: " << caramelIceCream->getDescription()
        << ", Cost: Rs." << caramelIceCream->cost()
        << endl;

   delete vanillaIceCream;
   delete chocolateIceCream;
   delete caramelIceCream;

   return 0;
}
```

Adapter Pattern

Adapter Pattern is a structural design pattern used to **make two incompatible interfaces work together**. It acts as a bridge between two incompatible interfaces, allowing them to collaborate without modifying their source code. This pattern is particularly useful when integrating legacy code or third-party libraries into your application.
Let's see below example.
Suppose you have a legacy printer class that only understands commands in uppercase, and a modern computer class that sends commands in lowercase. You need to make the modern computer work with the legacy printer without modifying the existing printer class.

LegacyPrinter is the legacy component (Adaptee). It has a method printInUppercase that can print text in uppercase.
ModernComputer is the modern client class. It has a method sendCommand to send commands, but it sends them in lowercase.
PrinterAdapter is the adapter class. It encapsulates the LegacyPrinter and adapts it to work with the ModernComputer.

```
// Legacy Printer (Adaptee)
class LegacyPrinter {
public:
        void printInUppercase(const std::string& text) {
                std::cout << "Printing: " << text << std::endl;
        }
};
```

```
// Modern Computer (Client)
class ModernComputer {
public:
        void sendCommand(const std::string& command) {
                std::cout << "Sending command: " << command << std::endl;
        }
};
```

```
// Adapter class to make the LegacyPrinter compatible with ModernComputer
class PrinterAdapter {
private:
	LegacyPrinter legacyPrinter;

public:
	void sendCommand(const std::string& command) {
		// Convert the command to uppercase and pass it to the LegacyPrinter
		std::string uppercaseCommand = command;
		for (char& c : uppercaseCommand) {
			c = std::toupper(c);
		}
		legacyPrinter.printInUppercase(uppercaseCommand);
	}
};
```

```
//Essentially, the adapter class converts the lowercase command from the ModernComputer into uppercase and delegates it to the LegacyPrinter.

int main() {
	ModernComputer computer;
	PrinterAdapter adapter;

	computer.sendCommand("Print this in lowercase");
	adapter.sendCommand("Print this in lowercase (adapted)");

	return 0;
}
```

Quiz for design pattern

What is the purpose of a design pattern?
a) To provide a generic solution to a common problem in software design.
b) To enforce coding standards in C++ programs.
c) To optimize the performance of C++ programs.
d) To replace built-in language features in C++.

Which design pattern is used to ensure that a class has only one instance and provides a global point of access to that instance?
a) Singleton
b) Factory Method
c) Observer
d) Strategy

What is the primary difference between the Adapter and the Decorator pattern?
a) Adapter modifies the behaviour of an object, while Decorator provides a different interface.
b) Decorator modifies the behaviour of an object, while Adapter provides a different interface.
c) Adapter allows incompatible interfaces to work together, while Decorator adds new functionality to an object.
d) Decorator allows incompatible interfaces to work together, while Adapter adds new functionality to an object.

Which design pattern is used to encapsulate the instantiation of a complex object and separate its construction from its representation?
a) Builder
b) Prototype
c) Factory Method
d) Abstract Factory

In the Factory Method pattern, what is the responsibility of the concrete creator classes?
a) To define an interface for creating objects.
b) To encapsulate the object creation and allow subclasses to alter the type of objects that will be created.
c) To provide an interface for creating families of related or dependent objects without specifying their concrete classes.
d) To encapsulate the details of creating a particular object, allowing its subclasses to provide a concrete implementation.