VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION AND TECHNOLOGY

ASSIGNMENT REPORT

Research for 3 topics: Generic programming in C++ (Template), Singleton pattern, and Composite pattern.

23CLC03

INSTRUCTOR

- **NGUYEN LE HOANG DUNG**
- **PHAM BA THAI**

—o0o—

STUDENT

- **NGUYEN HAI DANG**
- **MSSV: 23127165**

HO CHI MINH CITY, DECEMBER 2024

TABLE OF CONTENTS

1 Generic Programming

1.1 What is Generic Programming?

Generic programming is a programming paradigm where algorithms and data structures are written in terms of data types to be specified later. These generic components are then instantiated for specific types when needed, allowing the same code to operate on different data types without duplication. This approach, which originated in the programming language ML (1973), emphasizes reusability, type safety, and abstraction. It reduces redundancy by enabling the creation of functions or data structures that work with multiple types, differing only in the types they operate on.

1.2 Key Concept in Generic Programming

- **Abstraction of Types:** Abstraction of types is about creating algorithms and data structures that can work with multiple types without being specifically rewritten for each type. This concept allows developers to write more flexible and versatile code that can operate on different data types while maintaining a consistent underlying logic.

```cpp
// Generic function to swap two elements of any type
template <typename T>
void genericSwap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}

// Usage examples
int main() {
    int x = 5, y = 10;
    genericSwap(x, y);  // Works with integers

    std::string s1 = "hello", s2 = "world";
    genericSwap(s1, s2);  // Works with strings

    double d1 = 3.14, d2 = 2.71;
    genericSwap(d1, d2);  // Works with floating-point numbers
}

```

- **Parameterization:** Parameterization involves using type parameters that are resolved to specific types during compilation. This allows creating generic components where the actual type is determined when the component is used.

```cpp
// Generic Pair class that can hold two values of different types
template <typename K, typename V>
class Pair {
private:
    K first;
    V second;

public:
    Pair(K k, V v) : first(k), second(v) {}

    K getFirst() const { return first; }
    V getSecond() const { return second; }
};

int main() {
    // Create pairs of different type combinations
    Pair<std::string, int> studentInfo("John Doe", 25);
    Pair<int, std::string> keyValue(42, "Answer");
}

```

- **Type Safety:** Type safety ensures that type-related errors are caught at compile-time rather than runtime. This prevents many potential errors and provides stronger guarantees about the correctness of the code.

```cpp
// Generic function with type constraints
template <typename T>
T calculateSum(const std::vector<T>& collection) {
    // Ensures T is a numeric type that supports addition
    T sum = T();  // Zero-initialized
    for (const auto& item : collection) {
        sum += item;
    }
    return sum;
}

int main() {
    std::vector<int> intNumbers = {1, 2, 3, 4, 5};
    std::cout << calculateSum(intNumbers);  // Works fine

    std::vector<std::string> strings = {"hello", "world"};
    // calculateSum(strings);  // Compile-time error
}

```

- **Reusability:** Reusability is about creating a single implementation that can work across multiple data types, significantly reducing code duplication and improving maintainability.

```cpp
// Generic comparison function that works with any comparable type
template <typename T>
T findMaximum(const std::vector<T>& collection) {
    if (collection.empty()) {
        throw std::runtime_error("Empty collection");
    }

    T maxElement = collection[0];
    for (const auto& element : collection) {
        if (element > maxElement) {
            maxElement = element;
        }
    }
    return maxElement;
}

int main() {
    // Reusable across different types
    std::vector<int> numbers = {1, 5, 3, 9, 2};
    std::cout << findMaximum(numbers);  // Works with integers

    std::vector<std::string> fruits = {"apple", "banana", "cherry"};
    std::cout << findMaximum(fruits);  // Works with strings

    std::vector<double> temperatures = {98.6, 99.1, 97.5};
    std::cout << findMaximum(temperatures);  // Works with floating-point numbers
}

```

2 C++ Template

- The C++ language requires programmers to declare variables, functions, and other entities using specific types. However, many algorithms and data structures share similar logic regardless of the type they operate on. To address this and reduce redundancy, C++ provides templates, a powerful feature that supports generic programming. Templates allow programmers to write reusable, type-independent code for functions and classes, making the code both efficient and type-safe.

- A template works by treating the data type as a parameter, enabling the same code to operate on different data types without duplication. For example, instead of writing separate sort() functions for integers, floating-point numbers, or strings, a single template-based sort() function can handle all these types, with the data type specified when the function is called.

2.1 Function template

- In C++, function templates provide a powerful mechanism for generic programming, enabling developers to create versatile, type-independent functions. Rather than writing multiple near-identical function implementations for different data types, programmers can define a single function template that works across various types.

```
template <typename T>
return_type function_name(parameter_list) {
    // Function implementation using generic type T
}
```

- Key characteristics:

  + Generates type-specific functions automatically
  + Provides compile-time type checking
  + Eliminates code duplication
  + Supports multiple type parameters
  + Ensures zero runtime performance overhead

- A function template acts as a blueprint for generating type-specific functions at compile-time. It allows the creation of generic functions that can operate on different data types while maintaining type safety and preserving performance. For example:

```cpp
template <typename T>
T findMaximum(T a, T b) {
    return (a > b) ? a : b;
}

// Automatically works with multiple types
int main() {
    std::cout << findMaximum(10, 5);        // Integer
    std::cout << findMaximum(3.14, 2.71);   // Double
    std::cout << findMaximum('a', 'z');     // Character
}
```

- If you want to use more than one abstract data type in template function, here is the following syntax:

```cpp
template <typename T1, typename T2>

```

By leveraging function templates, C++ developers can write more flexible, maintainable, and efficient code that adapts seamlessly to different data types.

2.2 Class template

- A class template in C++ is a powerful mechanism for creating generic classes that can work with multiple data types. It serves as a blueprint for generating type-specific classes at compile-time, enabling developers to write more flexible and reusable code.

```cpp
template <typename T>
class ClassName {
    // Class definition using generic type T
};

```

- Key characteristics:
    + **Type Independence**: A single class template can be instantiated for multiple data types.
    + **Compile-Time Type Safety**: The compiler enforces type constraints for each instance.
    + **Reusability**: Write once, use various types without rewriting the logic.
    + **Customization**: Allows specialization to handle specific types differently when needed.

```cpp
template <typename T>
class Container {
private:
    T element;

public:
    // Constructor
    Container(T value) : element(value) {}

    // Getter and setter methods
    void setValue(T value) { element = value; }
    T getValue() const { return element; }
};

int main() {
    // Creating instances with different types
    Container<int> intContainer(42);
    Container<std::string> stringContainer("Hello");
    Container<double> doubleContainer(3.14);
}

```

- Class templates can support multiple type parameters by specifying additional placeholders in the template declaration. For example:

```cpp
template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;

public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    T1 getFirst() const { return first; }
    T2 getSecond() const { return second; }
};

int main() {
    Pair<int, std::string> person(25, "John Doe");
    Pair<double, char> measurement(98.6, 'F');
}

```

Class templates are an essential tool in C++ for implementing **generic and reusable solutions**, and they form the foundation for libraries like the STL. They allow developers to write more flexible, efficient, and type-safe code while reducing redundancy.

3 Brief Introduction to Design Pattern

- In software engineering, a design pattern is a reusable solution to a common problem in software design. It is not a ready-to-use blueprint but rather a template or guide on how to approach a problem effectively and maintainably. Design patterns embody best practices refined by experienced developers over time, providing standardized solutions that enhance code quality and efficiency.

- Design patterns are broadly categorized into three main types based on their purpose:

- ❖ Creational Pattern: These patterns focus on object creation mechanisms, enhancing flexibility and promoting code reuse. Common creational patterns include:
  - ➢ Singleton
  - ➢ Factory Method
  - ➢ Abstract Factory
  - ➢ Builder
  - ➢ Prototype
- ❖ Structural Patterns: Structural patterns define how to compose classes and objects into larger, scalable structures while maintaining efficiency. Examples include:
  - ➢ Adapter
  - ➢ Bridge
  - ➢ Composite
  - ➢ Decorator
  - ➢ Facade
  - ➢ Flyweight
  - ➢ Proxy
- ❖ Behavioral Patterns: These patterns focus on effective communication and responsibility sharing among objects. They ensure a clear flow of information and proper role assignment. Notable examples include:
  - ➢ Strategy
  - ➢ Observer
  - ➢ Command
  - ➢ State

This overview highlights the essence and classification of design patterns, demonstrating their

role in solving recurring design challenges while promoting efficient, scalable, and maintainable software solutions.

4 Singleton Pattern

- The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global access point to that instance. It is widely used when exactly one object is needed to coordinate actions across a system.

- More specifically, the singleton pattern allows classes to:

+ Ensure they only have one instance
+ Provide easy access to that instance
+ Control their instantiation (for example, hiding the constructors of a class)
+ The term comes from the mathematical concept of a singleton.

The term "singleton" comes from the mathematical concept of a singleton.

4.1 Common uses

- Singletons are favored over global variables because they help maintain a cleaner namespace by avoiding pollution of the global or containing namespace. Moreover, they enable lazy allocation and initialization, conserving resources until the instance is actually needed, unlike global variables that typically allocate resources immediately.

- The singleton pattern also serves as a foundation for several other design patterns, including abstract factory, factory method, builder, and prototype patterns. Facade objects are another common example, as typically only one facade instance is needed to streamline interactions with subsystems.

- A practical real-world application of singletons is in logging systems. Since all components requiring logging need a consistent access point and write to a centralized log source, a singleton provides an efficient and unified solution.

4.2 Implementations

- Implementations of the **Singleton Pattern** ensure that only one instance of a class exists and provide a global access point to that instance. This is typically achieved using the following steps:

1. **Private Constructors**: All constructors of the class are declared as private, preventing external objects from creating new instances of the class.

2. **Static Access Method**: A public static method is provided to return a reference to the single instance. This method is the global access point for retrieving the instance.

3. **Static Instance Variable**: The instance of the class is stored as a private static variable. This instance is initialized when the variable is first accessed, ensuring lazy initialization and resource optimization.

```cpp
class Singleton {
private:
    // Private constructor to prevent direct instantiation
    Singleton() {
        cout << "Singleton instance created!" << endl;
    }

    // Static variable to hold the single instance
    static Singleton* instance;

public:
    // Static method to provide access to the instance
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }

    void showMessage() {
        cout << "Hello from Singleton!" << endl;
    }
};

// Initialize static member
Singleton* Singleton::instance = nullptr;

int main() {
    Singleton* s1 = Singleton::getInstance();
    s1->showMessage();

    Singleton* s2 = Singleton::getInstance();
    cout << "Are s1 and s2 the same? " << (s1 == s2 ? "Yes" : "No") << endl;

    return 0;
}

```

This implementation ensures that the Singleton pattern is followed, preventing duplicate instances and providing controlled access through a global static method.

5 Composite Pattern

- The Composite Pattern is a structural design pattern used to compose objects into tree-like structures to represent part-whole hierarchies. It allows individual objects and composites (collections of objects) to be treated uniformly. This pattern is particularly useful when dealing with hierarchies of objects, where you can treat both leaf nodes (individual objects) and composite nodes (groups of objects) in the same way.

- Key Concepts
    + Leaf: Represents the end objects in the hierarchy that do not have any children. These are the individual components.
    + Composite: Represents objects that can have children (either leaf nodes or other composites).
    + Component: The common interface or abstract class for both leaf and composite objects. It allows clients to treat individual objects and composites uniformly.

The Composite Pattern enables clients to work with objects in a tree structure as if they were individual objects, providing a flexible way to build complex structures.

5.1 Common uses

- File System Hierarchies: A file system often contains files and directories. Directories may contain files or other directories. The Composite Pattern can be used to represent files and directories uniformly.

- UI Components: In graphical user interfaces, components like buttons, panels, and other widgets can be composed into containers. The Composite Pattern allows these UI elements to be treated uniformly, enabling the construction of complex interfaces.

- Organization Hierarchies: For example, in a corporate structure, both managers and employees can be part of an organizational hierarchy, and the Composite Pattern can represent both types uniformly.

=> The Composite pattern is useful when clients do not distinguish between individual objects and their compositions. If developers notice they are handling multiple objects in the same way, often writing nearly identical code for each, then using Composite is a good choice. It simplifies the situation by treating both primitives and composites as homogeneous.

5.2 Implementations

```cpp
// Component class
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() {}
};

// Leaf class (Individual objects)
class Leaf : public Component {
public:
    void operation() override {
        cout << "Leaf operation" << endl;
    }
};

// Composite class (Groups of objects)
class Composite : public Component {
private:
    vector<Component*> children;
public:
    void add(Component* component) {
        children.push_back(component);
    }

    void remove(Component* component) {
        children.erase(remove(children.begin(), children.end(), component), children.end());
    }

    void operation() override {
        cout << "Composite operation" << endl;
        for (auto& child : children) {
            child->operation();
        }
    }
};

int main() {
    // Creating leaf objects
    Component* leaf1 = new Leaf();
    Component* leaf2 = new Leaf();

    // Creating a composite object
    Composite* composite = new Composite();
    composite->add(leaf1);
    composite->add(leaf2);

    // Performing operations
    composite->operation();

    // Cleanup
    delete leaf1;
    delete leaf2;
    delete composite;

    return 0;
}

```

- **Component Class**: This is an abstract base class defining the operation method that both leaf and composite classes must implement.
- **Leaf Class**: Represents the individual objects (leaf nodes) in the tree. It implements the operation method to perform an action.
- **Composite Class**: Represents a container (composite node) that can hold child components (either other composites or leaves). It implements the operation method by iterating through its children and calling their operation methods.

=> The **Composite Pattern** is particularly useful for representing hierarchical data structures, allowing clients to treat both individual objects and groups of objects uniformly. It helps manage complex structures by breaking them down into simpler components, while ensuring flexible and extensible design.

6 References

[1] https://www.quora.com/What-is-generic-programming-1

[2] https://www.geeksforgeeks.org/templates-cpp/

[3] https://en.wikipedia.org/wiki/Singleton_pattern

[4] https://refactoring.guru/design-patterns/composite

[5] Absolute C++