C++ Templates & Generic Programming

Introduction to Templates

Templates are a powerful feature in C++ that enables generic programming. They allow you to write code that works with any data type without having to rewrite the same logic for different types.

Function Templates

- **Basic Syntax**:

```
template <typename T>
T functionName(T parameter1, T parameter2) {
    // function body
    return result;
}
```

- **Example**: A max function template

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

- **Usage**:

```
int i = Max(10, 20);           // T is int
double d = Max(3.14, 2.72);    // T is double
```

Class Templates

- **Basic Syntax**:

```
template <typename T>
class ClassName {
    private:
        T member;
    public:
        // methods
};
```

- **Example**: A simple Stack template

```
template <typename T>
class Stack {
    private:
        T data[100];
        int top;
    public:
        Stack() : top(-1) {}
        void push(T element);
        T pop();
        bool isEmpty();
};
```

- **Implementation**:

```
template <typename T>
void Stack<T>::push(T element) {
    data[++top] = element;
}

template <typename T>
T Stack<T>::pop() {
    return data[top--];
}

template <typename T>
bool Stack<T>::isEmpty() {
    return top == -1;
}
```

Multiple Template Parameters

```
template <typename T, typename U>
class Pair {
    private:
        T first;
        U second;
    public:
        Pair(T a, U b) : first(a), second(b) {}
        T getFirst() { return first; }
        U getSecond() { return second; }
};
```

Template Specialization

- Allows creating a special version of a template for a specific type

```
// General template
template <typename T>
class MyContainer {
    // general implementation
};

// Specialization for int
template <>
class MyContainer<int> {
    // specialized implementation for int
};
```

Non-Type Template Parameters

```
template <typename T, int SIZE>
class Array {
    private:
        T elements[SIZE];
    public:
        T& operator[](int i) { return elements[i]; }
        int size() { return SIZE; }
};

// Usage
Array<int, 10> intArray;  // Array of 10 integers
```

Standard Template Library (STL)

- Collection of template classes and functions
- Main components:
  - **Containers**: vector, list, map, set, etc.
  - **Iterators**: used to access container elements
  - **Algorithms**: sort, find, transform, etc.

Common STL Containers Examples

```cpp
#include <vector>
#include <map>
#include <string>

std::vector<int> numbers = {1, 2, 3, 4, 5};
std::map<std::string, int> ages = {{"Alice", 25}, {"Bob", 30}};
```

Best Practices

1. Keep templates simple and readable
2. Provide clear documentation for template parameters
3. Use meaningful template parameter names (not just T)
4. Consider providing specializations for common types
5. Be aware of code bloat (multiple instantiations)
6. Put declarations and definitions in header files

Common Errors

- Forgetting to use `typename` or `template` keywords in nested dependent names
- Forgetting template arguments when using a template
- Compilation errors inside templates that only appear when instantiated