1 Class Templates

Just like functions, classes can also be parameterized with types. This is especially useful for creating data structures or containers that can operate on a variety of data types. Class templates provide a way to write flexible and reusable code while maintaining type safety.

1.1 Class Template Declaration and Definition

Declaring and defining class templates follows the same structure as function templates. The `template` keyword introduces the template parameter list, which can then be used throughout the class:

```cpp
template<class T>
class MyClass
{
    // ...
};
```

The members of a class template can use the type parameters declared in the template definition:

```cpp
template<class T, class U, class V>
class MyClass
{
private:
    T var1;
    U* var2;
    V& var3;
    // ...
};
```

We can then instantiate `MyClass` with specific types:

```cpp
MyClass<int, int, int> x {};
```

For demonstration purposes, the following `List` class (defined in `list.h`) will be used throughout the examples:

```
#include <iostream>

template<class T>
class List {
    struct Node {
        T data;                     // node's data
        Node* next;                 // ptr to next node
        Node(T val): data(val), next(nullptr) {}
    };
    Node* head;                     // ptr to first node

public:
    List(): head(nullptr) {}
    ~List() {
        for (Node* cursor = head; cursor != nullptr; ) {
            Node* next { cursor->next };  // save next node
            delete cursor;                // delete current
            cursor = next;                // move to next
        }
    }

    void insert(T val) {
        Node* node { new Node(val) };     // make new node
        node->next = head;                // link it
        head = node;                      // update head ptr
    }

    friend std::ostream& operator<<
    (std::ostream& os, const List<T>& l) {
        for (typename List<T>::Node* cursor = l.head;
             cursor != nullptr; cursor = cursor->next) {
            os << cursor->data << " -> "; // print node data
        }
        os << "nullptr" << std::endl;
        return os;                        // end of list
    }
};
```

Heres an example of how to use the `List` class:

```
#include "list.h"

int main()
{
    List<int> list {};

    list.insert(40);
    list.insert(30);
    list.insert(20);
    list.insert(10);

    std::cout << list << std::endl;
}
```

```
Terminal
$ ./a.out
10 -> 20 -> 30 -> 40 -> nullptr
```

1.1.1 Member Function Definitions

To define a member function outside of the class definition, we must specify that it is a template and fully qualify the class template with its parameter. Heres how the external definition of `insert()` would look:

```
template<class T>
void List<T>::insert(T val)
{
    Node* node { new Node(val) };      // make new node
    node->next = head;                 // link it
    head = node;                       // update head ptr
}
```

When a member function of a class template takes additional template parameters, those parameters must be specified when defining the function outside the class definition:

```
#include <iostream>

template<class T>
struct Structure {
    T value;
    template<class U> void method() const;
};

template<class T>
template<class U>
void Structure<T>::method() const {
    std::cout << U(value) << std::endl;
}

int main()
{
    Structure<double> s { 5.5 };
    s.method<int>(); // prints 5
}
```

1.1.2 Static Member Definitions

Class templates can have static members, just like regular classes. The definition of a static data member in a class template involves specifying the template parameter followed by the variable definition:

```
template<class T>
struct S {
    static int s_var;
};

template<class T> int S<T>::s_var { -1 };
```

In this example, `s_var` is initialized to -1, and its value is shared across all instances of `S<T>` for any given type `T`.