1. 模板函数 (Function Templates)

- **定义**：模板函数是一种通用的函数定义方式，允许根据传入的参数类型生成特定类型的函数实例。
- **要求**：模板函数对参数类型有要求，如拷贝构造、操作符支持等。

示例：

```
#include <iostream>

// 模板函数定义
template<typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // 使用 int 类型
    int intResult = add(3, 4);
    std::cout << "int: 3 + 4 = " << intResult << std::endl;

    // 使用 double 类型
    double doubleResult = add(3.5, 2.7);
    std::cout << "double: 3.5 + 2.7 = " << doubleResult << std::endl;

    // 使用 std::string 类型
    std::string str1 = "Hello, ";
    std::string str2 = "World!";
    std::string stringResult = add(str1, str2);
    std::cout << "string: " << stringResult << std::endl;

    return 0;
}
```

这个模板函数接受两个类型为 `T` 的参数，并返回它们的和。模板参数 `T` 必须支持 `operator+` 操作。

2. 模板类 (Class Templates)

- **定义**：模板类是一种类定义方式，通过模板参数实现通用类型的类。模板类的成员变量和成员函数可以使用模板参数来决定其类型。

- **成员函数定义**：
  - 可以直接在类内定义，也可以在类外定义，类外定义时需要附带模板参数的声明。

示例：

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

// 模板类定义
template<typename T>
class MyClass {
private:
    T value; // 成员变量，类型由模板参数T决定

public:
    // 构造函数
    MyClass(T val) : value(val) {}

    // 成员函数
    void printValue() const;
};

// 成员函数类外定义
template<typename T>
void MyClass<T>::printValue() const {
    std::cout << value << std::endl;
}

int main() {
    // 创建 MyClass<int> 对象，T 是 int 类型
    MyClass<int> intObj(42);
    std::cout << "Integer value: ";
    intObj.printValue();

    // 创建 MyClass<double> 对象，T 是 double 类型
    MyClass<double> doubleObj(3.14);
    std::cout << "Double value: ";
    doubleObj.printValue();

    // 创建 MyClass<std::string> 对象，T 是 std::string 类型
    MyClass<std::string> stringObj("Hello, Template!");
    std::cout << "String value: ";
    stringObj.printValue();

    return 0;
}
```

3. 默认模板参数 (Default Template Arguments)

- **定义**：默认模板参数允许为模板参数指定一个默认值，如果实例化模板时没有提供参数，就会使用默认值。

示例：

```
template<typename T = int>
class MyClass {
private:
    T value;

public:
    MyClass(T val) : value(val) {}
};

Myclass<int> obj(10)
Myclass<> obj(10)
```

在这个例子中，如果没有指定 `T` 的类型，模板类 `MyClass` 默认使用 `int` 类型。

4. 模板类中的友元函数 (Friend Functions in Class Templates)

- **定义**：模板类中的友元函数允许外部函数访问模板类的私有成员。友元函数本身也可以是模板函数。
- **声明**：在类中声明友元函数时，必须确保模板参数的一致性。

示例：

```
template<typename T>
class MyClass {
private:
    T value;

public:
    MyClass(T val) : value(val) {}

    // 友元函数，允许输出类的私有成员
    friend std::ostream& operator<<(std::ostream& os, const MyClass<T>& obj) {
        os << obj.value;
        return os;
    }
};
```

示例代码讲解：

以下是你提供的代码，并附加了注释：

```
#ifndef STUDIO1_H
#define STUDIO1_H

#include <iostream>
using std::ostream;

// 前向声明类和操作符重载
template<typename T>
class MyClass;

template<typename T>
ostream& operator<<(ostream&, const MyClass<T>&);

// 模板类定义，带默认参数T为int
template<typename T = int>
class MyClass {
private:
    T value; // 成员变量类型由模板参数T决定

public:
    // 构造函数，初始化成员变量
    MyClass(T val);

    // 友元函数声明，用于重载输出操作符
    friend ostream& operator<< <>(ostream& os, const MyClass<T>& obj);
};

#include "studio1.cpp" // 包含实现文件

#endif
```

```cpp
// studio1.cpp: 模板类的成员函数实现

// 构造函数实现
template<typename T>
MyClass<T>::MyClass(T val) : value(val) {}

// 输出操作符重载的实现
template<typename T>
ostream& operator<<(ostream& os, const MyClass<T>& obj) {
    os << obj.value; // 输出私有成员value
    return os;
}


// main.cpp: 测试代码

#include "studio1.h"
#include <iostream>
#include <algorithm>

using namespace std;

const int SUCCESS = 0;

int main() {
    MyClass<> myClass1(1); // 使用默认模板参数int
    MyClass<> myClass2(2);

    cout << "Before swap: " << myClass1 << " " << myClass2 << endl;

    // 使用标准库中的swap函数
    swap(myClass1, myClass2);

    cout << "After swap: " << myClass1 << " " << myClass2 << endl;

    return SUCCESS;
}
```

主要知识点总结

1. **模板函数** 允许我们定义通用的函数，并根据需要对不同的类型进行实例化。
2. **模板类** 允许定义基于模板参数的类，成员变量和函数根据模板参数的类型进行实例化。
3. **默认模板参数** 提供了模板参数的默认值，减少了模板实例化时的复杂度。
4. **友元函数**：模板类中可以声明友元函数，使其能够访问私有成员，友元函数本身也可以是模板函数。

考题示例：

考题示例 1：
"What does this function template do?"

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

答案：此函数模板将两个同类型的参数 `a` 和 `b` 相加，并返回结果。要求 `T` 类型支持 `operator+` 。

考题示例 2：
"What type requirements does this function impose on T?"

答案：模板参数 `T` 必须支持加法操作（`operator+`）才能调用 `add` 函数。

考题示例 3：
"When would you use a class template?"

答案：当你需要定义一个能够处理多种类型的类时（如容器类 `std::vector` ），可以使用模板类。

考题示例 4：
"What does the following class template do?"

```
template <typename T>
class Box {
private:
    T value;
public:
    Box(T val) : value(val) {}
    T getValue() const { return value; }
};
```

答案： `Box` 是一个泛型类，用于存储任意类型 `T` 的值。构造函数初始化 `value` ，并提供 `getValue` 方法返回该值。

考题示例 5：
"What is the benefit of using default template arguments?"

答案：默认模板参数简化了模板的实例化过程，允许在不指定参数的情况下使用默认值。

考题示例 6：
"Why is the syntax for declaring friend functions in template classes more complicated?"

答案：由于模板类的参数化特性，友元函数也必须是模板函数，并且模板参数需要与类模板的参数一致，这增加了语法的复杂性。