C++ STL（标准模板库）简介

C++ STL（Standard Template Library）是C++的一部分，提供了一组通用的、可复用的组件，以简化编程过程。STL通过模板和泛型编程的方式，使得开发者可以使用预定义的数据结构与算法来快速构建高效的软件。

STL包括哪些内容

1. **容器**：用于存储对象的集合，用于存储数据的类，支持各种数据组织方式，包括：
   - **序列容器**：如 `vector`、`list` 和 `deque`。
   - **关联容器**：如 `set`、`map`、`multiset` 和 `multimap`。
   - **无序关联容器**（C++11引入）：如 `unordered_set` 和 `unordered_map`。

2. **算法**：对容器的操作函数，如排序、查找、合并等。这些算法是与容器无关的，能够用于多种容器类型。

3. **迭代器**：为容器提供统一的访问接口，允许使用类似于指针的方式遍历容器元素，简化对容器元素的访问，提供标准化的遍历方法。

专注点

STL的主要关注点在于：

- **代码复用**：通过模板实现通用算法和数据结构，减少重复代码。
- **性能优化**：提供高效的实现，针对大规模数据集的操作。
- **易用性**：通过统一接口简化操作，使得开发者能迅速使用。

实现STL的技术

STL的实现主要依赖于**模板技术**，允许算法和数据结构根据不同类型进行实例化。这意味着开发者可以编写一次代码，并在多种类型上使用，极大提高了代码的灵活性和复用性。

C++模板简介

在C++编程中，对于不同数据类型的操作，常常需要编写多个相似的函数。例如，在一个项目中，我们可能需要实现一个加法功能。对于不同的数据类型，如整型、浮点型等，就需要如下的代码：

没有模板的实现

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

// 整数加法
int add(int a, int b) {
    return a + b;
}
```

```
// 浮点数加法
double add(double a, double b) {
    return a + b;
}

// 单精度浮点数加法
float add(float a, float b) {
    return a + b;
}

int main() {
    cout << "Int Addition: " << add(5, 3) << endl;           // 整数相加
    cout << "Double Addition: " << add(5.5, 3.2) << endl;    // 浮点数相加
    cout << "Float Addition: " << add(2.5f, 1.5f) << endl;   // 单精度浮点数相加
    return 0;
}
```

在上述代码中，我们为每种数据类型编写了一个加法函数，这种方式不仅繁琐，还容易出错，并且对代码的维护造成了很大的挑战。

引入模板

C++模板的出现使得我们能够编写通用的代码。通过使用模板，我们可以编写一个函数来处理多种数据类型，从而避免重复代码并提高可读性。

1. C++模板的类型

C++模板主要有两种类型：函数模板和类模板。

1.1 函数模板

函数模板允许定义以任意类型作为参数的函数。开发者可以在编写函数时不指定具体数据类型，而在调用时由编译器进行类型推导。

示例：函数模板

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

template <typename T>
T add(T a, T b) {
    return a + b; // 返回两个相同类型参数的和
}
```

```
int main() {
    // 调用同一模板函数进行不同数据类型的操作
    cout << "Int Addition: " << add(5, 3) << endl;           // 整数相加
    cout << "Double Addition: " << add(5.5, 3.2) << endl;    // 浮点数相加
    cout << "Float Addition: " << add(2.5f, 1.5f) << endl;   // 单精度浮点数相加
    return 0;
}
```

在这个示例中，add函数模板接受任意类型 T的参数，并返回它们的和。无论是整数、双精度浮点数还是单精度浮点数，都能够正常工作。

1.2 类模板

类模板允许开发者定义使用任意数据类型的类。

示例：类模板

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

template <typename T>
class Box {
private:
    T value; // 存储任意类型的成员变量
public:
    Box(T v) : value(v) {} // 构造函数
    T getValue() { return value; } // 获取值的方法
};

int main() {
    // 创建不同数据类型的Box实例
    Box<int> intBox(123);
    Box<string> strBox("Hello");

    cout << "Int Box Value: " << intBox.getValue() << endl;  // 输出整数值
    cout << "String Box Value: " << strBox.getValue() << endl; // 输出字符串值

    return 0;
}
```

在这个示例中，Box类模板可以存储任意数据类型的值，通过模板参数 T定义了类的通用性。

2. C++模板的作用

C++模板的作用不仅限于减少代码冗余和提高可读性，它们在实际应用中有着更为广泛和深刻的影响。下面通过一些生动的现实生活例子，形象化地展示C++模板的作用。

2.1 代码重用的便利性

想象一下，你是一位厨师，每天都在为顾客准备不同的食物。无论是面条、炒饭，还是其他菜品，基本步骤都是相似的：开火、放入食材、调味、装盘。如果每次都要重新写一份食谱，那你就得在厨房里忙得不可开交，甚至可能因为重复的工作而出错。

在代码世界里，如果没有模板，我们就得为每种数据类型编写一份加法函数，像是为每种菜肴都写不同的食谱。使用模板就像是你找到了一个万能食谱，只需定义一次加法的“烹饪方法”，然后用不同的食材（数据类型）进行操作。

```
template <typename T>
T add(T a, T b) {
    return a + b; // 一个通用的加法实现
}
```

这个简单的模板函数让你可以轻松地对不同类型的数据进行加法运算，简化了代码的复杂性，仿佛你在厨房里只需用一个食谱就能做出多道美味佳肴。

2.2 提高代码维护性

假设你是一位建筑工程师，负责一座大桥的维护。这座桥的每个部分，如横梁、桥面和支架，都需要定期检查。如果你为每个部分都写一份独立的检查程序，万一出现问题，想找到根源就像在大海捞针一样困难。

在C++中，使用模板就像为这座桥设计了一个通用的维护手册。你只需编写一套检查程序，然后针对不同的部分传入不同的参数。这样，如果需要修改检查逻辑，只需在一个地方进行更新，避免了逐一修改的麻烦。

```
template <typename T>
class Stack {
    // Stack的实现
};
```

通过这种方式，代码的可维护性大大提高，随时可以快速查阅和调整。

2.3 支持不同数据结构的算法通用性

想象一下，你是一个图书馆的图书管理员，书籍的种类繁多，有小说、非小说、杂志等。如果你想制定一个借书规则，这个规则应适用于所有书籍类型，而不是为每种书籍分别制定一份规则。这样一来，借书的流程就会变得简单高效。

在软件开发中，C++模板使得算法能够适用于不同类型的数据结构。比如，你可以通过模板对各种容器（如向量、链表、集合）使用相同的排序算法，从而实现算法的灵活复用。无论容器中的数据是什么类型，排序算法都能如鱼得水，轻松运行。

```
template <typename T>
void sortAndPrint(vector<T>& vec) {
    sort(vec.begin(), vec.end()); // 对任意类型的vector进行排序
}
```

这种灵活性就像是在图书馆中，借书规则适用于所有类型的书籍，让借书变得简单而高效。

3. typename与 class的区别

在定义模板时，typename和 class两者通常可以互换

示例:

```
template <typename T> // 使用typename
class MyClass {
public:
    T value;
};

template <class T> // 使用class
class MyClass2 {
public:
    T value;
};
```

在以上示例中，MyClass和 MyClass2是等价的，二者都能正确编译，主要在于个人的编程风格选择。

4. 代码实例

下面是一些使用C++模板的代码示例，展示其在实际开发中的应用。

下面是一个使用C++模板创建动态数组的示例。该模板类允许用户创建能够存储任意类型的数据，并支持基本的动态数组操作，如添加、获取和删除元素。

4.1动态数组模板类示例

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

template <typename T>
class DynamicArray {
private:
    T* arr;          // 动态数组指针
    size_t capacity; // 当前容量
    size_t size;     // 当前大小

    // 增加数组容量
    void resize() {
        capacity *= 2; // 容量翻倍
        T* newArr = new T[capacity];
        for (size_t i = 0; i < size; ++i) {
            newArr[i] = arr[i]; // 复制旧数组的元素
        }
        delete[] arr; // 删除旧数组
        arr = newArr; // 更新指针
    }

public:
    // 构造函数
    DynamicArray() : capacity(2), size(0) {
        arr = new T[capacity]; // 初始化数组
    }

    // 析构函数
    ~DynamicArray() {
        delete[] arr; // 清理动态分配的内存
    }

    // 添加元素
    void add(T value) {
        if (size >= capacity) {
            resize(); // 如果达到容量则扩展
        }
        arr[size++] = value; // 插入新元素并增加大小
    }

    // 获取元素
    T get(size_t index) const {
        if (index >= size) {
            throw out_of_range("Index out of range"); // 检查越界
        }
        return arr[index]; // 返回元素
    }

    // 获取当前大小
```

```
    size_t getSize() const {
        return size;
    }

    // 删除最后一个元素
    void remove() {
        if (size > 0) {
            --size; // 只是减少大小，保持内存不变
        }
    }
};

int main() {
    DynamicArray<int> intArray; // 创建整数动态数组

    // 添加元素
    intArray.add(10);
    intArray.add(20);
    intArray.add(30);

    cout << "Current size: " << intArray.getSize() << endl; // 打印当前大小
    cout << "Elements in array: ";
    for (size_t i = 0; i < intArray.getSize(); ++i) {
        cout << intArray.get(i) << " "; // 打印所有元素
    }
    cout << endl;

    // 删除元素
    intArray.remove();
    cout << "Size after removal: " << intArray.getSize() << endl; // 打印修改后的大小

    return 0;
}
```

代码解析

1. **成员变量**:
    - `T* arr`: 动态数组的指针，存储元素。
    - `size_t capacity`: 当前容量，表示数组能够容纳的最大元素个数。
    - `size_t size`: 当前数组中实际存储的元素个数。
2. **构造函数 (`DynamicArray`)**:
    - 初始化容量为2，创建一个动态数组。
3. **析构函数 (`~DynamicArray`)**:
    - 在对象销毁时，释放动态数组的内存。

4. `add` **方法**:
    - 添加新元素到数组中。如果数组已满，调用 `resize` 方法扩大容量。
5. `get` **方法**:
    - 根据索引获取数组中的元素，包含边界检查。
6. `remove` **方法**:
    - 删除最后一个元素，实际不会释放数组内存，只是减小逻辑大小。
7. `resize` **方法**:
    - 当数组满时，创建一个新数组，并将旧数组中的元素复制到新数组中。
8. `main` **函数**:
    - 演示如何使用 `DynamicArray` 类创建一个整数数组，添加元素，输出当前大小和元素，最后删除一个元素并显示新的大小。

4.2 使用函数模板实现排序

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <typename T>
void sortAndPrint(vector<T>& vec) {
    sort(vec.begin(), vec.end()); // 排序
    for (const auto& elem : vec) {
        cout << elem << " "; // 打印排序后的元素
    }
    cout << endl;
}

int main() {
    vector<int> intVec = {5, 1, 4, 2, 8};           // 整数向量
    vector<double> doubleVec = {5.3, 1.2, 4.7, 2.1, 8.4}; // 浮点数向量

    cout << "Sorted Int Vector: ";
    sortAndPrint(intVec); // 对整数向量排序并打印

    cout << "Sorted Double Vector: ";
    sortAndPrint(doubleVec); // 对浮点数向量排序并打印
```

```
    return 0;
}
```

代码解析

1. **头文件包含**:
    - `#include <iostream>`: 提供输入输出功能。
    - `#include <vector>`: 使用动态数组（向量）。
    - `#include <algorithm>`: 提供排序算法。
2. **模板函数定义 (`sortAndPrint`)**:
    - `template <typename T>`: 定义一个函数模板，允许它接受任何类型 `T`。
    - `void sortAndPrint(vector<T>& vec)`: 函数接受一个引用类型的向量，支持在该函数中对原始数据进行排序。
    - `sort(vec.begin(), vec.end())`: 使用标准库提供的排序算法对向量进行排序。
    - `for (const auto& elem : vec)`: 遍历排序后的向量并打印每一个元素。
3. **主函数 (`main`)**:
    - 创建两个向量 `intVec` 和 `doubleVec` 分别存储整数和浮点数。
    - 调用 `sortAndPrint` 函数对两个向量进行排序，并打印结果。

4.2 使用类模板实现栈

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

template <typename T>
class Stack {
private:
    vector<T> elements; // 使用向量存储元素
public:
    void push(const T& element) {
        elements.push_back(element); // 压入元素
    }

    void pop() {
        if (!elements.empty()) {
            elements.pop_back(); // 弹出顶部元素
        }
    }
```

```
    T top() const {
        return elements.back(); // 获取顶部元素
    }

    bool isEmpty() const {
        return elements.empty(); // 检查栈是否为空
    }
};

int main() {
    Stack<int> intStack; // 创建整型栈
    intStack.push(1); // 压入元素
    intStack.push(2);
    cout << "Top Element: " << intStack.top() << endl; // 输出顶部元素
    intStack.pop(); // 弹出顶部元素
    cout << "Top Element after pop: " << intStack.top() << endl; // 再次输出顶部元素

    return 0;
}
```

代码解析

1. **头文件包含**:
   - `#include <iostream>`: 提供输入输出功能。
   - `#include <vector>`: 使用动态数组（向量）作为存储结构。
2. **类模板定义 (`Stack`)**:
   - `template <typename T>`: 定义一个模板类，允许使用任何类型 `T`。
   - `private: vector<T> elements;`: 使用动态向量存储栈中的元素。
3. **成员函数**:
   - `void push(const T& element)`: 向栈中添加元素，通过向量的 `push_back` 方法插入。
   - `void pop()`: 删除栈顶元素，首先检查栈是否为空，若不为空则弹出顶部元素。
   - `T top() const`: 返回栈顶元素，使用 `back()` 方法获取。
   - `bool isEmpty() const`: 检查栈是否为空，返回布尔值。
4. **主函数 (`main`)**:
   - 创建一个整型栈 `intStack`，通过调用 `push` 方法添加元素。
   - 使用 `top` 方法获取并打印栈顶元素，展示栈的功能。
   - 使用 `pop` 方法弹出栈顶元素，再次调用 `top` 方法验证栈的改变。