**1. STL 容器**

STL 提供了多种容器，用于存储和管理数据。以下是你学过的容器及其详细说明：

**1.1 string**

* **定义**：std::string 是 STL 中的字符串类，用于处理字符序列。
* **特点**：
  + 可变长度的字符数组。
  + 支持字符串操作，如拼接、查找、替换等。
* **常用操作**：
  + **访问**：str[index]（直接访问）、str.at(index)（带边界检查）
  + **修改**：str.append()（追加）、str.insert()（插入）、str.erase()（删除）
  + **查找**：str.find()（查找子串）、str.rfind()（从右查找）
  + **比较**：str.compare()（字符串比较）
  + **转换**：std::stoi()（转为整数）、std::stof()（转为浮点数）

**1.2 vector**

* **定义**：std::vector 是一个动态数组，支持随机访问。
* **特点**：
  + 内存自动管理，容量可动态扩展。
  + 元素在内存中连续存储，访问效率高。
* **常用操作**：
  + **访问**：vec[index]、vec.at(index)
  + **修改**：vec.push\_back()（尾部添加）、vec.pop\_back()（尾部删除）、vec.insert()（插入）、vec.erase()（删除）
  + **大小**：vec.size()（元素个数）、vec.capacity()（当前容量）
  + **调整**：vec.resize()（调整大小）、vec.reserve()（预留空间）

**1.3 deque**

* **定义**：std::deque（double-ended queue）是一个双端队列。
* **特点**：
  + 支持随机访问，但效率低于 vector。
  + 两端插入和删除效率高。
* **常用操作**：
  + **访问**：deq[index]、deq.at(index)
  + **修改**：deq.push\_front()（头部添加）、deq.push\_back()（尾部添加）、deq.pop\_front()（头部删除）、deq.pop\_back()（尾部删除）
  + **大小**：deq.size()

**1.4 stack**

* **定义**：std::stack 是一个后进先出（LIFO）的容器适配器。
* **特点**：
  + 只能访问栈顶元素。
  + 底层容器默认是 deque，可指定为 vector 或 list。
* **常用操作**：
  + **压栈**：stk.push()
  + **出栈**：stk.pop()
  + **访问栈顶**：stk.top()
  + **判断空**：stk.empty()
  + **大小**：stk.size()

**1.5 queue**

* **定义**：std::queue 是一个先进先出（FIFO）的容器适配器。
* **特点**：
  + 只能访问队首和队尾。
  + 底层容器默认是 deque，可指定为 list。
* **常用操作**：
  + **入队**：que.push()
  + **出队**：que.pop()
  + **访问队首**：que.front()
  + **访问队尾**：que.back()
  + **判断空**：que.empty()
  + **大小**：que.size()

**1.6 list**

* **定义**：std::list 是一个双向链表。
* **特点**：
  + 不支持随机访问。
  + 插入和删除效率高，时间复杂度 O(1)。
* **常用操作**：
  + **插入**：lst.push\_front()（头部添加）、lst.push\_back()（尾部添加）、lst.insert()（指定位置插入）
  + **删除**：lst.pop\_front()（头部删除）、lst.pop\_back()（尾部删除）、lst.erase()（删除指定元素）
  + **访问**：lst.front()（头部元素）、lst.back()（尾部元素）
  + **大小**：lst.size()

**1.7 set**

* **定义**：std::set 是一个有序集合，元素唯一且自动排序。
* **特点**：
  + 基于红黑树实现，插入、删除、查找时间复杂度为 O(log n)。
  + 元素不可重复。
* **常用操作**：
  + **插入**：st.insert()
  + **删除**：st.erase()
  + **查找**：st.find()（返回迭代器）、st.count()（返回元素个数，0 或 1）
  + **大小**：st.size()

**1.8 map**

* **定义**：std::map 是一个有序键值对容器，键唯一且自动排序。
* **特点**：
  + 基于红黑树实现，插入、删除、查找时间复杂度为 O(log n)。
  + 键不可重复。
* **常用操作**：
  + **插入**：mp.insert()、mp[key] = value（直接赋值）
  + **删除**：mp.erase()
  + **查找**：mp.find()（返回迭代器）、mp.count()（返回键个数，0 或 1）
  + **访问**：mp[key]
  + **大小**：mp.size()

**1.9 multiset**

* **定义**：std::multiset 是一个有序集合，允许重复元素。
* **特点**：
  + 基于红黑树实现，插入、删除、查找时间复杂度为 O(log n)。
  + 元素可重复。
* **常用操作**：
  + **插入**：mst.insert()
  + **删除**：mst.erase()
  + **查找**：mst.find()（返回迭代器）、mst.count()（返回元素个数）
  + **大小**：mst.size()

**1.10 multimap**

* **定义**：std::multimap 是一个有序键值对容器，允许重复键。
* **特点**：
  + 基于红黑树实现，插入、删除、查找时间复杂度为 O(log n)。
  + 键可重复。
* **常用操作**：
  + **插入**：mmp.insert()
  + **删除**：mmp.erase()
  + **查找**：mmp.find()（返回迭代器）、mmp.count()（返回键个数）
  + **大小**：mmp.size()

**2. 函数对象、仿函数和谓词**

**2.1 函数对象（Function Object）**

* **定义**：函数对象是重载了 operator() 的类的对象，可以像函数一样调用。
* **特点**：
  + 可保存状态（例如计数器）。
  + 常作为算法参数，提供灵活性。
* **示例**：

cpp

CollapseWrapCopy

struct Add {

int operator()(int a, int b) { return a + b; }

};

Add add;

int sum = add(3, 4); *// sum = 7*

**2.2 仿函数（Functor）**

* **定义**：仿函数通常指 STL 中用作算法参数的函数对象。
* **常用仿函数**：
  + std::less<T>：判断 < 关系。
  + std::greater<T>：判断 > 关系。
* **用途**：例如在 sort 中自定义排序规则：

cpp

CollapseWrapCopy

std::sort(vec.begin(), vec.end(), std::greater<int>()); *// 降序*

**2.3 谓词（Predicate）**

* **定义**：谓词是返回布尔值的函数对象，用于条件判断。
* **类型**：
  + **一元谓词**：接受一个参数，例如 bool isEven(int x) { return x % 2 == 0; }
  + **二元谓词**：接受两个参数，例如 bool lessThan(int a, int b) { return a < b; }
* **用途**：常用于查找或筛选，例如 find\_if。

**3. STL 算法**

STL 算法是通用的操作函数，可以与各种容器配合使用。以下是你学过的算法分类和详细说明：

**3.1 常用遍历算法**

* **for\_each**：
  + **功能**：对容器中每个元素应用指定函数。
  + **用法**：for\_each(begin, end, func)
  + **示例**：打印 vector 中每个元素：

cpp

CollapseWrapCopy

std::for\_each(vec.begin(), vec.end(), [](int x) { std::cout << x << " "; });

* **transform**：
  + **功能**：对每个元素应用函数，结果存入另一容器。
  + **用法**：transform(begin, end, dest\_begin, func)
  + **示例**：将 vector 中每个元素加 1：

cpp

CollapseWrapCopy

std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x + 1; });

**3.2 常用查找算法**

* **find**：
  + **功能**：查找第一个匹配的元素，返回迭代器。
  + **用法**：find(begin, end, value)
* **find\_if**：
  + **功能**：查找第一个满足谓词的元素。
  + **用法**：find\_if(begin, end, pred)
  + **示例**：查找第一个偶数：

cpp

CollapseWrapCopy

auto it = std::find\_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; });

* **adjacent\_find**：
  + **功能**：查找相邻的重复元素。
  + **用法**：adjacent\_find(begin, end)
* **binary\_search**：
  + **功能**：在已排序容器中二分查找，返回布尔值。
  + **用法**：binary\_search(begin, end, value)
* **count**：
  + **功能**：统计等于某值的元素个数。
  + **用法**：count(begin, end, value)
* **count\_if**：
  + **功能**：统计满足谓词的元素个数。
  + **用法**：count\_if(begin, end, pred)

**3.3 常用排序算法**

* **sort**：
  + **功能**：对容器元素排序，默认升序。
  + **用法**：sort(begin, end) 或 sort(begin, end, comp)
  + **示例**：降序排序：

cpp

CollapseWrapCopy

std::sort(vec.begin(), vec.end(), std::greater<int>());

* **shuffle**：
  + **功能**：随机打乱容器元素。
  + **用法**：shuffle(begin, end, random\_engine)
* **merge**：
  + **功能**：合并两个已排序容器。
  + **用法**：merge(begin1, end1, begin2, end2, dest\_begin)
* **reverse**：
  + **功能**：反转容器元素顺序。
  + **用法**：reverse(begin, end)

**3.4 常用拷贝和替换算法**

* **copy**：
  + **功能**：将元素从一个容器拷贝到另一个容器。
  + **用法**：copy(begin, end, dest\_begin)
* **replace**：
  + **功能**：将等于某值的所有元素替换为新值。
  + **用法**：replace(begin, end, old\_value, new\_value)
* **replace\_if**：
  + **功能**：将满足谓词的元素替换为新值。
  + **用法**：replace\_if(begin, end, pred, new\_value)
* **swap**：
  + **功能**：交换两个容器的内容。
  + **用法**：swap(container1, container2)

**3.5 常用算术生成算法**

* **accumulate**：
  + **功能**：对容器元素进行累积操作（如求和）。
  + **用法**：accumulate(begin, end, init\_value)
  + **示例**：求和：

cpp

CollapseWrapCopy

int sum = std::accumulate(vec.begin(), vec.end(), 0);

* **fill**：
  + **功能**：用指定值填充容器。
  + **用法**：fill(begin, end, value)

**3.6 常用集合算法**

* **set\_intersection**：
  + **功能**：求两个已排序容器的交集。
  + **用法**：set\_intersection(begin1, end1, begin2, end2, dest\_begin)
* **set\_union**：
  + **功能**：求两个已排序容器的并集。
  + **用法**：set\_union(begin1, end1, begin2, end2, dest\_begin)
* **set\_difference**：
  + **功能**：求两个已排序容器的差集。
  + **用法**：set\_difference(begin1, end1, begin2, end2, dest\_begin)

**4. 总结与建议**

* **容器选择**：
  + 需要快速随机访问：用 vector。
  + 频繁两端操作：用 deque。
  + 栈或队列需求：用 stack 或 queue。
  + 频繁插入删除：用 list。
  + 自动排序和唯一性：用 set 或 map。
  + 允许重复元素：用 multiset 或 multimap。
* **算法使用**：
  + 遍历和修改：for\_each 和 transform。
  + 查找：根据条件选择 find、find\_if 或 binary\_search。
  + 排序：sort 是核心，搭配仿函数可自定义规则。
  + 集合操作：set\_intersection 等适用于已排序数据。
* **函数对象和谓词**：
  + 通过自定义仿函数和谓词，可以让算法更灵活，例如在 sort 或 find\_if 中使用。

**1. 模板（Templates）**

模板是 C++ 中实现泛型编程的核心工具，通过参数化类型，使得代码可以处理多种数据类型，从而提高复用性。

**1.1 函数模板（Function Templates）**

函数模板允许定义一个通用的函数，适用于不同数据类型。

* **语法**：

cpp

CollapseWrapCopy

template <typename T>

T functionName(T param) {

*// 函数体*

}

* **特点**：
  + typename T 可以用 class T 替代，二者等价。
  + 调用函数时，编译器根据实参类型自动推导 T 的具体类型。
  + 支持显式指定类型，例如 functionName<int>(param)。
* **示例**：

cpp

CollapseWrapCopy

template <typename T>

T max(T a, T b) {

return (a > b) ? a : b;

}

int main() {

int i = max(3, 7); *// T 被推导为 int，结果为 7*

double d = max(3.5, 2.1); *// T 被推导为 double，结果为 3.5*

return 0;

}

* **注意事项**：
  + 可以定义多个模板参数，例如 template <typename T1, typename T2>。
  + 如果参数类型不匹配，需显式指定类型或进行类型转换。

**1.2 类模板（Class Templates）**

类模板允许定义一个通用的类，适用于不同数据类型的成员。

* **语法**：

cpp

CollapseWrapCopy

template <typename T>

class ClassName {

public:

T member;

ClassName(T val) : member(val) {}

T getMember() { return member; }

};

* **特点**：
  + 实例化类模板时，必须指定类型，例如 ClassName<int>。
  + 成员函数只有在被调用时才会实例化。
* **示例**：

cpp

CollapseWrapCopy

template <typename T>

class Box {

public:

T content;

Box(T val) : content(val) {}

};

int main() {

Box<int> intBox(123); *// 存储整数*

Box<double> doubleBox(45.67); *// 存储双精度浮点数*

return 0;

}

* **成员函数定义**： 如果成员函数在类外定义，需带上模板参数：

cpp

CollapseWrapCopy

template <typename T>

T Box<T>::getMember() {

return content;

}

**1.3 模板特化（Template Specialization）**

模板特化允许为特定类型提供定制化的实现。

* **类型**：
  + **全特化**：为特定类型完全重定义模板。
  + **偏特化**：为部分类型或条件提供特化。
* **全特化示例**：

cpp

CollapseWrapCopy

template <>

class Box<char> {

public:

char content;

Box(char val) : content(val) {}

void print() { std::cout << "Char: " << content << std::endl; }

};

* **用途**：
  + 优化特定类型的逻辑，或处理特殊需求。

**2. 继承（Inheritance）**

继承是面向对象编程的重要特性，允许派生类复用基类的代码，并实现多态。

**2.1 基本概念**

* **基类（Base Class）**：被继承的类。
* **派生类（Derived Class）**：继承基类的类。
* **继承方式**：
  + public：基类的 public 和 protected 成员保持原访问权限。
  + protected：基类的 public 和 protected 成员变为 protected。
  + private：基类的 public 和 protected 成员变为 private。
* **语法**：

cpp

CollapseWrapCopy

class Derived : public Base {

*// 派生类成员*

};

**2.2 构造函数和析构函数**

* **构造函数**：
  + 派生类构造函数需要调用基类构造函数，通常通过初始化列表实现。
  + 示例：

cpp

CollapseWrapCopy

class Base {

public:

int baseVal;

Base(int val) : baseVal(val) {}

};

class Derived : public Base {

public:

int derivedVal;

Derived(int a, int b) : Base(a), derivedVal(b) {}

};

* **析构函数**：
  + 基类的析构函数应声明为虚函数，以确保派生类对象正确析构。
  + 示例：

cpp

CollapseWrapCopy

class Base {

public:

virtual ~Base() { std::cout << "Base destroyed\n"; }

};

class Derived : public Base {

public:

~Derived() { std::cout << "Derived destroyed\n"; }

};

**2.3 多重继承**

* **定义**：一个派生类继承多个基类。
* **语法**：

cpp

CollapseWrapCopy

class Derived : public Base1, public Base2 {

*// 派生类成员*

};

* **注意事项**：
  + 多重继承可能导致**菱形继承问题**，即多个基类继承自同一祖先类，导致重复继承。
  + **虚继承**解决菱形问题：

cpp

CollapseWrapCopy

class Base {};

class Derived1 : virtual public Base {};

class Derived2 : virtual public Base {};

class FinalDerived : public Derived1, public Derived2 {};

**2.4 虚函数和多态**

* **虚函数**：基类中声明为 virtual 的函数，派生类可重写。
* **多态**：通过基类指针或引用调用虚函数时，根据实际对象类型执行相应版本。
* **示例**：

cpp

CollapseWrapCopy

class Base {

public:

virtual void show() { std::cout << "Base\n"; }

};

class Derived : public Base {

public:

void show() override { std::cout << "Derived\n"; }

};

int main() {

Base\* ptr = new Derived();

ptr->show(); *// 输出 "Derived"*

delete ptr;

return 0;

}

**3. 友元（Friends）**

友元机制允许特定函数或类访问另一个类的私有和保护成员。

**3.1 友元函数**

* **定义**：在类中声明为 friend 的函数，可以访问该类的私有和保护成员。
* **语法**：

cpp

CollapseWrapCopy

class MyClass {

private:

int secret;

public:

MyClass() : secret(42) {}

friend void friendFunc(MyClass& obj);

};

void friendFunc(MyClass& obj) {

std::cout << obj.secret << std::endl; *// 输出 42*

}

* **特点**：
  + 友元函数不是类的成员函数。
  + 可以是全局函数或另一个类的成员函数。

**3.2 友元类**

* **定义**：一个类声明另一个类为友元，允许友元类访问其私有和保护成员。
* **语法**：

cpp

CollapseWrapCopy

class FriendClass {

public:

void access(MyClass& obj) {

std::cout << obj.secret << std::endl;

}

};

class MyClass {

private:

int secret;

public:

MyClass() : secret(42) {}

friend class FriendClass;

};

**3.3 注意事项**

* 友元关系是单向的，不具有传递性或继承性。
* 友元破坏封装性，应谨慎使用。
* 友元声明必须在类定义内部。

**4. 总结**

* **模板**：
  + 函数模板和类模板实现泛型编程，减少代码重复。
  + 模板特化提供特定类型的优化。
* **继承**：
  + 实现代码复用和多态，支持 public、protected 和 private 继承。
  + 注意构造函数调用顺序和虚析构函数的使用。
* **友元**：
  + 提供访问私有成员的途径，但应避免滥用。

类模板中的继承和友元

1. 类模板中的继承

类模板可以像普通类一样参与继承关系。继承可以发生在类模板之间，也可以是类模板和普通类之间。

1.1 类模板继承普通类

定义：类模板可以继承一个普通类（非模板类）。

特点：

普通类作为基类，类模板作为派生类。

派生类模板可以访问基类的所有成员（根据访问控制）。

示例：

cpp

Collapse

Wrap

Copy

class Base {

public:

void baseFunc() { std::cout << "Base function\n"; }

};

template <typename T>

class Derived : public Base {

public:

void derivedFunc() {

baseFunc(); // 调用基类函数

std::cout << "Derived function with type: " << typeid(T).name() << std::endl;

}

};

int main() {

Derived<int> d;

d.derivedFunc(); // 输出 "Base function" 和 "Derived function with type: int"

return 0;

}

1.2 类模板继承类模板

定义：一个类模板可以继承另一个类模板。

特点：

继承时需要指定基类模板的参数。

派生类模板的参数可以与基类模板相同或不同。

示例：

cpp

Collapse

Wrap

Copy

template <typename T>

class Base {

public:

T baseMember;

Base(T val) : baseMember(val) {}

};

template <typename U>

class Derived : public Base<U> { // 继承 Base<U>

public:

U derivedMember;

Derived(U val1, U val2) : Base<U>(val1), derivedMember(val2) {}

void show() {

std::cout << "Base member: " << this->baseMember << ", Derived member: " << derivedMember << std::endl;

}

};

int main() {

Derived<int> d(10, 20);

d.show(); // 输出 "Base member: 10, Derived member: 20"

return 0;

}

1.3 普通类继承类模板

定义：普通类可以继承一个类模板的实例。

特点：

需要在继承时指定类模板的参数。

示例：

cpp

Collapse

Wrap

Copy

template <typename T>

class Base {

public:

T baseMember;

Base(T val) : baseMember(val) {}

};

class Derived : public Base<int> { // 继承 Base<int>

public:

Derived(int val) : Base<int>(val) {}

void show() {

std::cout << "Base member: " << baseMember << std::endl;

}

};

int main() {

Derived d(100);

d.show(); // 输出 "Base member: 100"

return 0;

}

2. 类模板中的友元

友元机制在类模板中同样适用，允许指定函数或类访问类模板的私有和保护成员。

2.1 友元函数与类模板

定义：在类模板中声明友元函数。

特点：

友元函数可以是普通函数、函数模板或另一个类模板的成员函数。

示例：

cpp

Collapse

Wrap

Copy

template <typename T>

class MyClass {

private:

T secret;

public:

MyClass(T val) : secret(val) {}

friend void friendFunc(MyClass<T>& obj); // 声明友元函数

};

template <typename T>

void friendFunc(MyClass<T>& obj) {

std::cout << "Secret: " << obj.secret << std::endl;

}

int main() {

MyClass<int> mc(42);

friendFunc(mc); // 输出 "Secret: 42"

return 0;

}

2.2 友元类与类模板

定义：在类模板中声明友元类。

特点：

友元类可以是普通类或另一个类模板。

示例：

cpp

Collapse

Wrap

Copy

template <typename T>

class FriendClass {

public:

void access(MyClass<T>& obj) {

std::cout << "Accessing secret: " << obj.secret << std::endl;

}

};

template <typename T>

class MyClass {

private:

T secret;

public:

MyClass(T val) : secret(val) {}

friend class FriendClass<T>; // 声明 FriendClass<T> 为友元类

};

int main() {

MyClass<int> mc(42);

FriendClass<int> fc;

fc.access(mc); // 输出 "Accessing secret: 42"

return 0;

}

2.3 友元函数模板

定义：在类模板中声明友元函数模板。

特点：

友元函数模板可以访问类模板的私有成员。

示例：

cpp

Collapse

Wrap

Copy

template <typename T>

class MyClass {

private:

T secret;

public:

MyClass(T val) : secret(val) {}

template <typename U>

friend void friendFunc(MyClass<U>& obj); // 声明友元函数模板

};

template <typename U>

void friendFunc(MyClass<U>& obj) {

std::cout << "Secret: " << obj.secret << std::endl;

}

int main() {

MyClass<int> mc(42);

friendFunc(mc); // 输出 "Secret: 42"

return 0;

}

3. 注意事项

继承中的模板参数：

在继承类模板时，基类模板的参数必须明确指定。

派生类模板的参数可以与基类模板的参数相同或不同。

友元声明：

友元声明必须在类模板内部。

友元函数或类可以访问类模板的私有和保护成员。

模板特化与继承：

类模板的特化版本也可以参与继承关系。

多重继承：

类模板可以参与多重继承，但需注意菱形继承问题。

4. 总结

继承：

类模板可以继承普通类、类模板或被普通类继承。

继承关系中，模板参数的传递和指定是关键。

友元：

类模板中可以声明友元函数、友元类或友元函数模板。

友元机制允许外部函数或类访问类模板的私有成员。