

# Templates – Part 1/2

---

Proto Team 技术培训系列

# Overview

# 编译期技术

技术	侧重点	优点	缺点
宏	简化代码、提高可读性	简洁、高效	容易出错、不易维护
constexpr	提高效率、减少错误	高效、安全	使用范围有限、不易使用
模板	提高可重用性、提高效率	可重用、高效	学习曲线较长、容易出错
concept	提高安全性、减少错误	安全、高效	是 C++20 的新特性、支持度不够广泛

# Templates

- 支持任意类型/值的通用代码
  - 使用 `template<placeholders>` 来定义（例外，有些情况下使用 `auto` 可以省略）
  - 类型/值在模板被引用时在编译期被具化
- 成为非常重要的C++特性
  - 甚至超过了继承（比如在 `stl` 库中的广泛使用）
- 一种递归的语言
  - 不只是简单的替换
  - 可以在编译期进行“计算”

# Template 的缺点

- 可读性相对较差
  - 模板代码通常相对复杂, 降低了代码的可读性。
  - 模板元编程尤其晦涩难懂, 因为涉及许多模板元函数和递归。
- 错误消息复杂
  - 当模板的使用中出现错误时, 编译器生成的错误消息可能会变得冗长且难以追踪。
- 实例化开销
  - 模板可能导致代码体积增大, 因为每个模板实例都需要额外的内存和代码。
- 语法和思维方式与普通代码不同
  - 模板的一些行为可能不如非模板代码直观, 导致使用模板的代码在语法和行为上与其他代码不一致。
- 通常需要在头文件中声明和实现
- 编译时间增加
  - 由于模板是在编译时实例化的, 当使用大型模板库或复杂的模板时, 可能会导致编译时间显著延长。

# Contents

- Templates (1/2)

- Function Templates
- Class Templates
- Variable Templates (NTTP)
- Variadic Templates
- Alias Templates

- Templates (2/2)

- Template Meta-Programmin
- Template Recursion
- SFINAE
- Type traits

# Function Templates

# Function template – Basic Usage

```
5 template <typename T>
6 T getMax(T a, T b) {
7     return (a > b) ? a : b;
8 }
```

模板定义

- typename == class

```
int num1 = 10, num2 = 20;
std::cout << getMax(num1, num2) << std::endl;

double double1 = 3.14, double2 = 2.71;
std::cout << getMax(double1, double2) << std::endl;

std::string str1 = "apple", str2 = "banana";
std::cout << getMax(str1, str2) << std::endl;

std::cout << getMax<std::string>("apple", "banana");
```

实例化

- 实例化出三个具体方法
- 有时可以推导出类型

如果省略<std::string>会如何？

如果换成<const char\*>可以么？

对类型T的隐含要求是什么？



# Example: basic usage

```
class Student {  
public:  
    std::string name;  
    int age;  
  
    // Constructor  
    Student(const std::string& n, int a) : name(n), age(a) {}  
  
    // Spaceship operator for three-way comparison based on age  
    auto operator<=>(const Student& other) const {  
        return age <=> other.age;  
    }  
};
```

Student需要支持 > 操作符和拷贝构造函数

如果delete拷贝构造函数会如何？

<https://godbolt.org/z/554xfahc4>

# 打印任意容器的元素

要求：给定任意容器，比如 vector、set、list、map，依次打印出每个元素

```
23 template <typename Container>
24 void printContainer(const Container& c) {
25     for (const auto& element : c) {
26         std::cout << element << " ";
27     }
28     std::cout << std::endl;
29 }
```

对Container的隐含要求是什么？

1. 支持range based loop 迭代
2. 元素支持 ostream << 操作符

```
// 测试 std::vector
std::vector<int> vec = {1, 2, 3, 4, 5};
printContainer(vec);
```

```
// 测试 std::map
std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}, {3, "three"}};
printContainer(myMap);
```

```
// 测试 std::set
std::set<double> mySet = {3.14, 2.718, 1.414};
printContainer(mySet);
```

```
17 template <typename T1, typename T2>
18 std::ostream& operator<<(std::ostream& os, const std::pair<T1, T2>& p) {
19     os << "(" << p.first << ", " << p.second << ")";
20     return os;
21 }
```

- Templates are usually *defined* in header files
  - Not only *declared*
  - No `inline` necessary

mycode.hpp:

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

```
#include "mycode.hpp"
...

int i1=42, i2=77;
auto a = mymax(i1, i2);      // OK
auto b = mymax(0.7, 33.4);  // OK
std::string s{"he"}, t{"ho"};
auto c = mymax(s, t);       // OK
```

mycode.hpp:

```
template<typename T>
T mymax(T a, T b);
```

```
#include "mycode.hpp"
...

int i1=42, i2=77;
auto a = mymax(i1, i2);      // Error
auto b = mymax(0.7, 33.4);  // Error
std::string s{"he"}, t{"ho"};
auto c = mymax(s, t);       // Error
```

# Specialization

```
4 template <typename T>
5 bool equal(T a, T b)
6 {
7     return a == b;
8 }
9
10 template<>
11 bool equal<double>(double a, double b)
12 {
13     return std::abs(a - b) < 0.001;
14 }
15
16
17 int main()
18 {
19     std::cout << std::boolalpha << equal(1.0002, 1.0001) << std::endl;
20     return 0;
21 }
```

<https://godbolt.org/z/6revoPvz9>

# Class Template

# 定义

类模板定义了一个类, 其中一些变量的类型、返回方法的类型和/或方法的参数被指定为模板参数。

# 通用的Stack

```
12 template <typename T>
13 class Stack
14 {
15 private:
16     std::vector<T> elems;
17 public:
18     Stack();
19     void push(const T&);
20     T pop();
21     T top() const;
22     bool empty() const {
23         return elems.empty();
24     }
25 };
```

直接实现

```
32 template <typename T>
33 void Stack<T>::push(const T& e)
34 {
35     elems.push_back(e);
36 }
37
38 template <typename T>
39 T Stack<T>::pop()
40 {
41     assert(!elems.empty());
42     T elem = elems.back();
43     elems.pop_back();
44     return elem;
45 }
```

```
47 template <typename T>
48 T Stack<T>::top() const
49 {
50     assert(!elems.empty());
51     return elems.back();
52 }
```

```
Stack<int> intStack;
intStack.push(7);
std::cout << intStack.top() << std::endl;

Stack<std::string> strStack;
strStack.push("hello");
std::cout << strStack.pop() << std::endl;

Stack<std::complex<double>> complexStack;
complexStack.push({1.0, 2.0});
std::cout << complexStack.top().real() << std::endl;
```

在外部实现, Function Template 的方式

但通常也需要在头文件内

# 模板类的实例化

```
1 template <typename T>
2 class Stack
3 {
4 private:
5     std::vector<T> elems;
6 public:
7     Stack() {}
8     void push(const T& e) {elems.push_back(e);}
9     T pop() {
10         assert(!elems.empty());
11         T elem = elems.back();
12         elems.pop_back();
13         return elem;
14     }
15     T top() const {
16         assert(!elems.empty());
17         return elems.back();
18     }
19     bool empty() const {
20         return elems.empty();
21     }
22 };
```

```
25 int main()
26 {
27     Stack<int> intStack;
28     if (intStack.empty())
29     {
30         std::cout << "Empty stack" << std::endl;
31     }
32
33     return 0;
34 }
```

这个模板会如何实例化？



# 模板类的实例化

<https://godbolt.org/z/7Yb3xP6qh>

```
1 template <typename T>
2 class Stack
3 {
4 private:
5     std::vector<T> elems;
6 public:
7     Stack() {}
8     void push(const T& e) {elems.push_back(e);}
9     T pop() {
10         assert(!elems.empty());
11         T elem = elems.back();
12         elems.pop_back();
13         return elem;
14     }
15     T top() const {
16         assert(!elems.empty());
17         return elems.back();
18     }
19     bool empty() const {
20         return elems.empty();
21     }
22 };
```



```
1 template<>
2 class Stack<int>
3 {
4 private:
5     std::vector<int> elems;
6 public:
7     Stack() {}
8     void push(const int& e) {elems.push_back(e);}
9     int pop() {
10         assert(!elems.empty());
11         int elem = elems.back();
12         elems.pop_back();
13         return elem;
14     }
15     int top() const {
16         assert(!elems.empty());
17         return elems.back();
18     }
19     bool empty() const {
20         return elems.empty();
21     }
22 };
```

# 模板类的实例化

模板类中的方法只有在使用的时候才初始化

```
1 template <typename T>
2 class Stack
3 {
4 private:
5     std::vector<T> elems;
6 public:
7     Stack() {}
8     void push(const T& e) {elems.push_back(e);}
9     T pop() {
10         assert(!elems.empty());
11         T elem = elems.back();
12         elems.pop_back();
13         return elem;
14     }
15     T top() const {
16         assert(!elems.empty());
17         return elems.back();
18     }
19     bool empty() const {
20         return elems.empty();
21     }
22 };
```



```
1 template<>
2 class Stack<int>
3 {
4 private:
5     std::vector<int> elems;
6 public:
7     Stack() {}
8     void push(const int & e);
9     int pop();
10    int top() const;
11    bool empty() const {
12        return elems.empty();
13    }
14 };
```

<https://godbolt.org/z/7Yb3xP6qh>

# 成员方法使用才实例化

- 可以减少实例化的类的大小
- 可能会产生出人意料的编译错误

```
1 template <typename T>
2 class Stack
3 {
4     ...
5     void print() const {
6         std::cout << "Elems: ";
7         for(const T& e: elems){
8             std::cout << e << " ";
9         }
10        std::cout << std::endl;
11    }
12    ...
13};
```

```
1 int main()
2 {
3     ...
4     Stack<int> intStack;
5     intStack.push(7);
6     intStack.print();
7
8     Stack<std::pair<int, int>> pairStack;
9     pairStack.push({1, 10});
10    pairStack.push({2, 20});
11    pairStack.print();
12
13 }
```

error: no match for 'operator<<' (operand types are 'std::ostream' (aka 'std::basic\_ostream<char>') and 'const std::pair<int, int>')

# 类模板参数推导 – CTAD

- Class Template Argument Deduction (since C++ 17)
- 从构造函数的参数自动推导模板的类型参数
- 尽量只对明显可推导出来的类型进行使用

```
3 template <typename T, typename U>
4 class Pair {
5 public:
6     T first;
7     U second;
8
9     Pair(T f, U s) : first(f), second(s) {}
10 };
11
12 int main() {
13     Pair myPair{42, "Hello, CTAD!"};
14
15     std::cout << "First: " << myPair.first << std::endl;
16     std::cout << "Second: " << myPair.second << std::endl;
17
18     return 0;
19 }
```

<https://godbolt.org/z/TqnaGW6qT>

扩展知识: CTAD的优先级问题, 比如vector v{10, 20}和vector v(10, 20) 的区别

# Specialization

```
1 template <>
2 class Stack<char*>
3 {
4 private:
5     std::vector<std::string> elems;
6
7 public:
8     Stack() : elems() {}
9
10    void push(const char* e) { elems.push_back(e); }
11    const char* pop()
12    {
13        assert(!elems.empty());
14        const char* elem = elems.back().c_str();
15        elems.pop_back();
16        return elem;
17    }
18    const char* top() const
19    {
20        assert(!elems.empty());
21        return elems.back().c_str();
22    }
23    bool empty() const
24    {
25        return elems.empty();
26    }
27 };
```

# Non-Type Template Parameters

NTTP

# 非类型模板参数 (NTTP)

- 模板的参数是一个值, 而非类型
- 给定不同的值会产生不同的类型
  - `std::array<int, 42>` 和 `std::array<int, 10>` 是两个不同的类型
- 支持的类型包括
  - An integral type
  - An enumeration type
  - A pointer or reference to a class object
  - A pointer or reference to a function
  - A pointer or reference to a class member function
  - `std::nullptr_t`
  - A floating point type (since C++20)

# Example: SizedStack

```
1 template <typename T, int SIZE>
2 class SizedStack {
3 private:
4     T elems[SIZE];
5     int topIndex;
6
7 public:
8     SizedStack() : topIndex(-1) {}
9     void push(const T& value) {
10         if (topIndex == SIZE - 1) {
11             throw std::overflow_error("Stack is full");
12         }
13         elems[++topIndex] = value;
14     }
15     T pop() {
16         assert(!empty());
17         return elems[topIndex--];
18     }
19     T top() const {
20         assert(!empty());
21         return elems[topIndex];
22     }
23
24     bool empty() const {
25         return topIndex == -1;
26     }
27 };
```

<https://godbolt.org/z/TKsn3sf1Y>



# Variadic Templates

# 可变模板 – Variadic Templates

- 模板类型的数目可变
- 可以应用于类模板和函数模板
- 语法使用参数包(Parameter Pack)
  - `typename... Args` - 参数类型
  - `Args... args` - 参数
  - `args...` - 访问参数
  - `sizeof...()` - 获得Pack的size
- 实现中需要用到递归

# Example – Print

```
1 void print()
2 {
3     std::cout << std::endl;
4 }
5
6
7 template<typename T, typename... Ts>
8 void print(T firstArg, Ts... args)
9 {
10     std::cout << firstArg << " ";
11     print(args...);
12 }
13
14
15 int main() {
16     print("1", 2, 3.0, '4');
17
18     return 0;
19 }
```

递归终止条件

```
22 template<typename T, typename... Ts>
23 void print(T firstArg, Ts... args)
24 {
25     std::cout << firstArg << " ";
26     if (sizeof...(args) > 0) {
27         print(args...);
28     }
29     else
30     {
31         std::cout << std::endl;
32     }
33 }
```

这样可以work么？

# Variadic class template

- Examples
  - `std::tuple`, `std::variant`
- 比Variadic function template 更难实现
  - 递归的struct比递归函数更难以理解
- tuple的模拟实现
  - <https://godbolt.org/z/f9c3rK9aa>
  - <https://godbolt.org/z/zf3aq75z5>

# Alias Templates

# Alias Templates

- 可以为template创建别名
- 可以在创建别名的时候进行部分特例化
- 别名模板本身不能再进一步特例化
- 作用
  - 增强代码可读性 – 使用别名可以使代码更易读
  - 简化复杂类型 – 可以通过别名简化复杂的类型声明, 提高代码的可维护性
  - 模板元编程 – 在模板元编程中, Alias Templates可以帮助创建更具表达力的代码

## Example – IntArray

```
3 template <size_t N>
4 using IntArray = std::array<int, N>;
5
6 int main() {
7     IntArray my_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
8
9     for (int i = 0; i < my_array.size(); i++) {
10         std::cout << my_array[i] << std::endl;
11     }
12 }
```

# 扩展内容

Lambda templates

Default Template Arguments

基于auto的 function templates

从类模板继承



# Exercises

1. Gameboard with class inheritance implementation

<https://godbolt.org/z/jKcGe6dWs>

2. 模板系列：

- a. 编写一个带有两个模板类型参数 (Key 和 Value) 的 KeyValuePair 类模板。该类应该有两个私有数据成员，用于存储键和值。提供一个接受键和值的构造函数，并添加适当的 getter 和 setter。通过在主函数 main() 中创建一些实例并尝试使用类模板参数推导来测试你的类。

<https://godbolt.org/z/Wfv5974Mo>

- b.

# THANKS



上海合见工业软件集团有限公司  
Shanghai UniVista Industrial Software Group Co., Ltd.