C++期末不挂科

CH8 - 模板

本章内容

- 模板与泛型的概念
- 模板函数
- 模板类

理解以下与函数有关的名词：

- 泛型编程
- 参数类型化
- 模板函数
- 模板类
- 模板类型参数
- 参数占位符
- 模板实例化
- 类型推断

思考并回答以下问题：

- 泛型编程/类型参数化的目的是什么
- C++如何确保模板逻辑可以正确应用在类型参数上

熟悉以下关键字：

- template
- typename

熟悉以下代码写法：

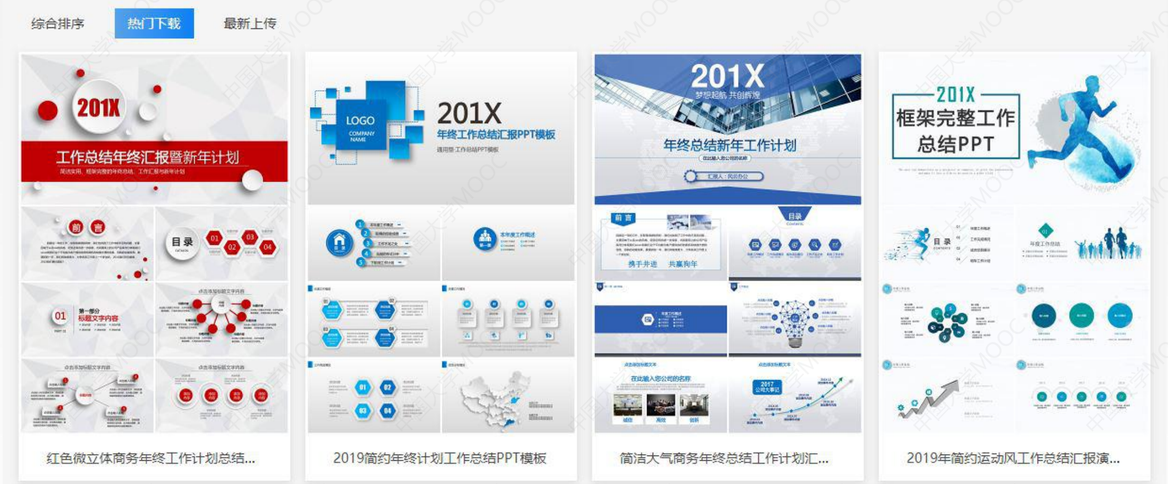
- 编写和实例化模板函数
- 通过类型推断自动实例化模板函数
- *编写带类型参数和非类型参数的模板类

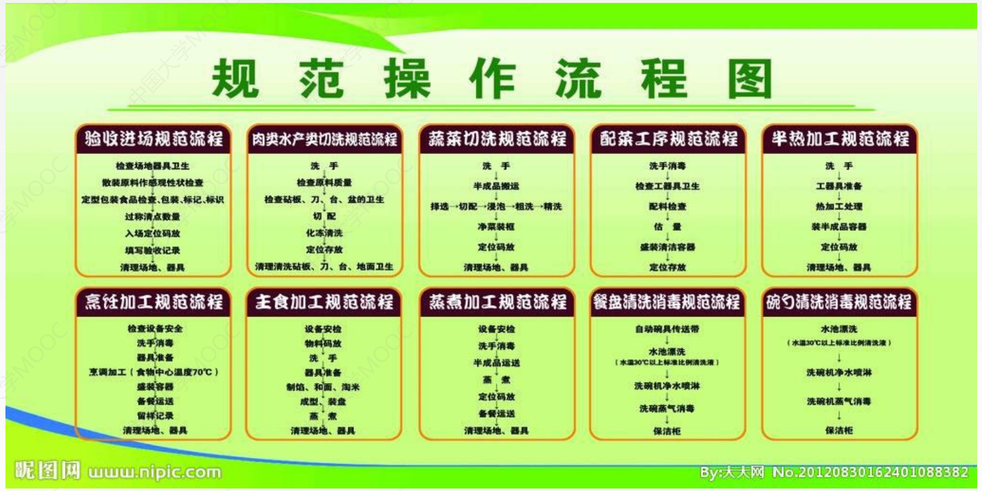
模板与泛型的概念

模板函数

模板类

模板 template





都可以用

- 小明的2019工作总结
- 小华的2020年度汇报
- 我的2022商务计划
- …

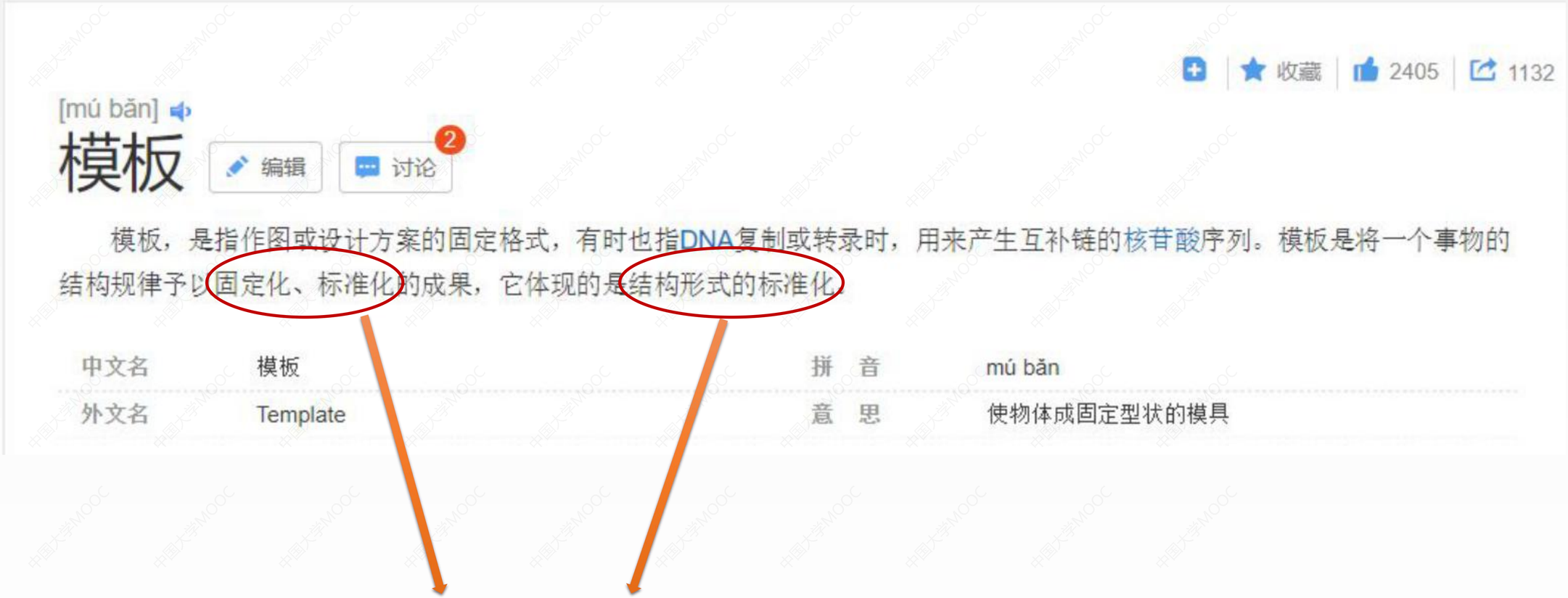
都这流程

- 西红柿炒鸡蛋
- 糖醋排骨
- 黑椒牛柳
- …

都一样洗

- 大盘子
- 小碗
- 刀叉筷子
- …

模板 template



所谓标准化：同一套流程应用于所有合理的对象

模板编程的目的

- 思考：如何利用函数重载实现不同类型数据的自定义相加(拼接)?

```
// 实现三个重载函数：
string myAdd(int a, int b) {
        return to_string(a) + to_string(b);
}
string myAdd(float a, float b) {
        return to_string(a) + to_string(b);
}
string myAdd(double a, double b) {
        return to_string(a) + to_string(b);
}
// 调用：
cout << myAdd(111, 999);
cout << myAdd(3.14f, 2.27f);
cout << myAdd(3.1415926535, 2.2777777777);
```

- 调用方确实减少了代码量，实现了代码复用
- 但实现方仍然冗余：一样的逻辑为三种类型各写一遍

模板编程的目的

- 回顾：从函数 -> 函数重载 -> 面向对象 -> 继承/多态 -> … 核心目的?
- 代码复用
- 但以上技术仍然是类型依赖的，需要为不同类型进行相应的实现，如函数重载
- 即“数据类型”信息是一个超参数，代码逻辑**依赖**这个超参数信息
- 模板编程：类型参数化，将类型信息也视作一个普通参数，使**代码逻辑与类型信息分离**

泛型的思想

模板编程的目的

- 对不同类型变量进行自定义的拼接相加

通过函数重载实现

```
// 实现三个重载函数：
string myAdd(int a, int b) {
        return to_string(a) + to_string(b);
}
string myAdd(float a, float b) {
        return to_string(a) + to_string(b);
}
string myAdd(double a, double b) {
        return to_string(a) + to_string(b);
}
// 调用：
cout << myAdd(111, 999);
cout << myAdd(3.14f, 2.27f);
cout << myAdd(3.1415926535, 2.2777777777);
```

增加代码复用

通过模板实现

```
// 实现一个模板函数：
template <typename T1, typename T2>
string myAdd(T1 a, T2 b) {
        return to_string(a) + to_string(b);
}
// 调用：
cout << myAdd<int, int>(111, 999);
cout << myAdd<float, float>(3.14f, 2.27f);
cout << myAdd<int, float>(8, 3.14f);
```

模板与泛型的概念

模板函数

模板类

模板函数

- 用模板函数对不同类型变量进行自定义的拼接相加
  - template 关键字：表明接下来定义一个模板
  - typename 关键字：表明该模板参数是一个**类型名**
  - T1、T2: 是两个类型形式参数，是**类型占位符**
- 于是在myAdd函数中就可以用T1和T2指代数据类型，编写代码逻辑

模板参数能不是一个类型吗？是可以的

```
template <typename T1, typename T2>
string myAdd(T1 a, T2 b) {
        return to_string(a) + to_string(b);
}
```

变量a和b你都不知道会是啥就to_string了

是不是很没安全感？

别担心，编译器在**实例化**时会检查的

模板函数的实例化

发生在**编译**时，由编译器完成：

```
template <typename T1, typename T2>
string myAdd(T1 a, T2 b) {
        return to_string(a) + to_string(b);
}
```

```
cout << myAdd<int, int>(111, 999);
cout << myAdd<float, float>(3.14f, 2.27f);
cout << myAdd<int, float>(8, 3.14f);
```

```
// T1为int, T2为int，模板被实例化为：
string myAdd(int a, int b) {
        return to_string(a) + to_string(b);
}
```

```
// T1为float, T2为float，模板被实例化为：
string myAdd(float a, float b) {
        return to_string(a) + to_string(b);
}
```

```
// T1为int, T2为float，模板被实例化为：
string myAdd(int a, float b) {
        return to_string(a) + to_string(b);
}
```

模板函数的实例化

```
template <typename T1, typename T2>
string myAdd(T1 a, T2 b) {
        return to_string(a) + to_string(b);
}
```

如果这样调用：

```
cout << myAdd<string, int>( “abc”, 999);
```

编译时进行实例化

T1为string, T2为int，函数实例化为：

```
string myAdd(string a, int b) {
        return to_string(a) + to_string(b);
}
```

to_string函数不能用在string对象上

因此这个问题会在实例化时被发现并报**编译错误**

模板类型推断

```
template <typename T1, typename T2>
string myAdd(T1 a, T2 b) {
        return to_string(a) + to_string(b);
}
```

如果这样调用：

```
cout << myAdd(8, 3.14f);
```

- 若不传模板类型参数，编译器会根据各个位置的实参类型进行**类型推断**
- 根据实参 8 和实参 3.14f 的类型，推断T1为int，T2为float，进而进行模板实例化

模板函数更多例子

- 模板参数也可以用在返回值上

```
template <typename T>
T* my_new(int size) {
        if (size < 0) {
                cout << "Invalid size:" << size << endl;
                return nullptr;
        }
        return new T[size];
}
```

模板与泛型的概念

模板函数

模板类

模板类

- 可以把模板参数用在类定义上
- 即模板参数的作用范围为该类内部
- 如自定义一个类型无关的数组类

```
// 使用：
MyArray<float> a;
a.setEle(2, 3.14f);
cout << a.getEle(2);
```

```
template <typename T>
class MyArray {
private:
        T data[100];
public:
        MyArray() {}
        T getEle(int index) {
                return data[index];
        }
        void setEle(int index, T ele) {
                data[index] = ele;
        }
};
```

非类型的模板参数

- 模板参数不一定是类型，可以是实际数据
- 如定义一个自定义大小、类型无关的数组类

```
// 使用：
MyArray<float, 100> a;
a.setEle(2, 3.14f);
cout << a.getEle(2);
```

```
template <typename T, int size>
class MyArray {
private:
        T data[size];
public:
        MyArray() {}
        T getEle(int index) {
                return data[index];
        }
        void setEle(int index, T ele) {
                data[index] = ele;
        }
};
```

模板类不可声明与定义分离

- 普通类可以模块化：在 .h 中声明，在 .cpp 中定义(实现)
- 但模板类不可以这样操作，必须声明同时就定义，为什么？
- 因为编译器无法事先知道类型占位符 T 是什么，所以必须同时写

☆ Key 1：模板的核心思想：类型参数化，不依赖数据类型来编写程序逻辑

☆ Key 2：模板可用于函数和类的编写：模板函数和模板类

☆ Key 3：template：声明模板，typename：定义模板的类型参数

1. 模板函数的使用实际上是在_____将模板实例化成一个_____：C

A. 编译期，宏定义　　B. 运行时，类　　C. 编译期，函数　　D. 编译期，类

2. 关于函数模板，描述错误的是：A

A. 函数模板必须由程序员实例化为可执行的函数模板

B. 函数模板的实例化由编译器实现

C. 一个类定义中，只要有一个函数模板，则这个类是类模板

D. 类模板的成员函数都是函数模板，类模板实例化后，成员函数也随之实例化

分析：选项C的原因是一个类只要有一个成员函数是模板函数，它就必须声明同时定义，因此可视为一个类模板

⭐ Key 1：模板的核心思想：类型参数化，不依赖数据类型来编写程序逻辑

⭐ Key 2：模板可用于函数和类的编写：模板函数和模板类

⭐ Key 3：template：声明模板，typename：定义模板的类型参数

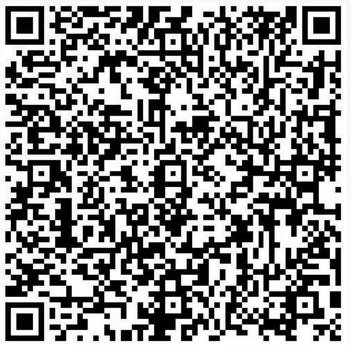
3. 下列的模板说明中，正确的是：C

A. template (typename T1, typename T2)

B. template < typename T1, T2 >

C. template < typename T1, typename T2 >

D. template < typedef T1, typedef T2 >



中国大学MOOC
搜索：C++不挂科

4. 假设有函数模板定义如下，则下列选项正确的是：B

A. int x, y; char z; Add(x, y, z) ;

B. double x, y, z; Add(x, y, z);

C. int x, y; float z; Add(x, y, z);

D. float x; double y, z; Add(x, y, z) ;

```
template <typename T>
void Add(T a, T b, T& c) { c = a + b; }
```