Template

函数模板与类模板

- 函数模板和类模板本质上可以理解成泛型函数和泛型类，但是它们实际上是模板，而不是真正的函数和类。添加 template<typename 类型 1, typename 类型 2,...> 并替换掉原有硬编码类型的声明即可
- 函数模板

```
template<typename T>
const T& maxOfThree(const T& a, const T& b, const T& c) {
    const T& maxValue = a > b ? a : b;
    return maxValue > c ? maxValue : c;
}
```

- 模板类

```
template<typename T>
class Buzz {
private:
    T m_data;
};
```

- **模板类、类模板，它们是完全不同的东西，就如同我们不能说月饼模具和月饼是同样的东西**
  - 类模板，可以认为是一个月饼模具，你可以往里面儿装五仁馅儿、莲蓉馅儿，猪肉馅儿就算了，反正做出来的月饼都长一个样，只是里面儿的内容不一样而已
  - 模板类，就是用模具做出来的五仁月饼、莲蓉月饼，是类模板实例化后的结果

函数模板的实参推断

- 实参推断
  - 在调用上述的 maxOfThree 模板函数时，我们并不需要显式的指定参数类型，可由编译器进行参数类型推断

```
int p = maxOfThree(15, 10, 225);          // 编译器推断
int q = maxOfThree<int>(15, 10, 225);   // 显式指定
```

- 实参类型转换
  - 在一些函数调用发生时，实参可能会进行适当的类型转换，以适应形参类型
    - 算数转换 — float 转 int，char 转 int，int 转 double 等
    - 子类向基类的转换 — 发生在函数形参为引用或指针时
    - 数组向指针转换 — 当形参不是引用类型时，数组名将会转换成指针
    - 非常量向常量转换 — 仅能将非常量实参转换常量实参
  - **在未显式指定模板类型时，形参仅能进行 「const 转换」以及 「数组名向指针转换」**

```
template<typename T>
inline const T& max(const T& a, const T& b) {
    return a > b ? a : b;
}

max(10, 25);          // right, T 将被推断为 int
max(10, 25.0);        // wrong，T 既可以被推断成 int ，也可以被推断成 double
max<int>(10, 25.0) // right，显式指定 T 的类型为 int，故 25.0 将会被隐式转换成 int
```

显示具体化（模板特化）

- C++ 没有办法限制类型参数的范围，所以我们可以将类型参数实例化成任意类型。但函数体或者是类体中并不能接收所有类型。

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

  - 例如 max 函数模板，结构体或者类在没有重载 > 的前提下是不能进行比较的
- Java 这一点就做的很好，例如 T extends Comparable<? super T> ，强制要求 T 可比较
- 为了解决诸如结构体、类实例无法调用 max 的情况，C++ 允许我们指出函数模板的具体类型

```
template<>
const Buz& max(const Buz& a, const Buz& b) {
    return a.length > b.length ? a : b;
}
```

- 偏特化 — 我们还可以这样玩儿
  - 类模板 — `template<typename T1, typename T2> class Buz {};` — 匹配优先级最低
  - 特化类模板 — `template<> class Buz<int, int> {};` — 匹配优先级最高
  - 偏特化类模板
    - `template<T1, int> class Buz {};` — 匹配优先级次高
    - vector 就是一种偏特化模板，针对于 vector<bool>

模板中的非类型参数

- array 是 c++ 中用来代替 C 数组的一种静态数组，它和 C 数组有着相同的特点: 大小不可更改
- 当我们需要使用静态数组时，尽可能的使用 array，除非需要和 C 相互兼容
- array 本身也是一个类模板，在 C++17 以前，我们需要在类型参数中显式的指定静态数组的大小
  - `array<int, 3> p = {1, 2, 3};`
  - 3 为静态数组的大小，类型为 size_t，即不是类型参数
- `template<typename T, size_t size> class MyArray {};`
  - 在实例化 MyArray 时，非类型参数必须是常量表达式

```
MyArray<String, 1024> a;  // 正确
int n = 10;
MyArray<String, n>;          // 错误，n 为非常量
```

引用折叠

- 左值与右值
  - 左值是一个能够对其进行取地址的变量或表达式，变量一般都是左值。右值是一种临时对象，不能对其取地址，例如 2 + 3，i++
  - 左值（&）引用引用左值，右值引用（&&）引用右值，右值引用会延长右值的生命周期，右值引用本身是一个左值
  - 对于一个实际类型来说，int& k 表示左值引用，int&& k 表示一个右值引用，函数形参同理
- **但是，类型参数的 T&& 推导会发生引用折叠**
  - 当然，T& 一定会被推导成左值
  - T&& 可能会被推导成左值，也可能被推导成右值
    - 当实参为左值引用时，T&& 的结果为左值引用。相当于 T&& &，将会被折叠成 T&
    - 当实参为右值时，T&& 的结果为实参类型本身，例如 int，double 等
    - 当实参为实际类型时，T&& 的结果为右值引用

模板于多文件编程

- 声明和定义分离 — 在大部分情况下，我们会将函数或者类的声明放在头文件中，其定义（实现）放到其它文件中，于链接阶段补上编译阶段缺省的函数或类定义
- **但是，对于函数模板和类模板来说，它们并不是真正同的函数或者类，而是一个模板，或者说一个模具。因此，我们不能将模板的声明和定义分开**
- 其原因在于模板的实例化是在编译阶段完成的，而不是链接阶段，这样就会导致在链接阶段可能找不到对应类型的函数定义