面向对象编程（OOP）和泛型编程都处理编写程序时未知的类型。 两者之间的区别在于，OOP 处理在运行时之前未知的类型，而在泛型编程中，类型在编译期间才已知

* template <typename T>：声明一个模板，T 是一个占位符类型，表示任意类型。

编译器会根据调用时的类型参数（如 int、double、char）生成具体的函数。

template <typename T1, typename T2>

返回类型 函数名(参数列表) {

// 函数体

}

**1. 模板的本质**

* 模板是一种**代码生成工具**，它允许你编写与类型无关的通用代码。
* 在编写模板时，使用**占位符类型**（如 typename T）来表示任意类型。
* 当使用模板时，编译器会根据提供的具体类型参数生成对应的类或函数

这些编译器生成的函数通常称为模板的实例化。

**. 非类型模板参数的定义**

* 非类型模板参数是指模板参数不是类型，而是一个具体的值。
* 非类型模板参数可以是以下类型：
  + **整型**（如 int、long 等）。
  + **指针**（指向对象或函数）。
  + **引用**（左值引用，指向对象或函数）。

**（2）指针或引用参数**

* 绑定到指针或引用非类型模板参数的实参必须具有**静态生命周期**。
  + 静态生命周期包括：
    - 全局变量。
    - 静态局部变量。
    - 静态成员变量。
  + 不能使用普通局部变量或动态分配的对象作为实参。
* 非类型模板参数可以是整型、指针或引用。
* 绑定到整型参数的实参必须是常量表达式。
* 绑定到指针或引用参数的实参必须具有静态生命周期。
* 非类型模板参数在模板定义中是常量值，可以用于需要常量表达式的场景。

模板程序应尽量减少对参数类型的要求数量。

**1. 模板中的名字**

模板中的名字可以分为两类：

1. **不依赖于模板参数的名字**：
   * 这些名字在模板定义时就已经确定。
   * 例如，模板中使用的标准库函数、全局变量等。
2. **依赖于模板参数的名字**：
   * 这些名字在模板实例化时才能确定。
   * 例如，模板参数类型的成员函数、运算符等。

**2. 模板提供者的责任**

* **确保非依赖名字可见**：
  + 模板提供者需要确保所有不依赖于模板参数的名字在模板使用时是可见的。
  + 例如，如果模板使用了 std::cout，则需要包含 <iostream> 头文件。
* **确保模板定义可见**：
  + 模板的定义（包括类模板的成员函数定义）必须在模板实例化时可见。
  + 通常将模板的定义放在头文件中。

**. 模板使用者的责任**

* **确保实例化类型相关声明可见**：
  + 模板使用者需要确保用于实例化模板的类型的所有相关声明（如函数、类型、运算符）是可见的。
  + 例如，如果模板使用了某个类的成员函数，则需要包含该类的头文件。
* 默认情况下，类模板的成员只有在被使用时才会被实例化。
* 这种机制可以减少编译时间和目标文件大小，并避免不必要的编译错误。
* 如果需要强制实例化某个成员，可以使用显式实例化。
* 类模板可以包含**非模板友元**或**模板友元**。
* 非模板友元可以访问类模板的所有实例化版本。
* 模板友元的访问权限可以由类模板控制：
  + 所有实例化版本：友元模板的所有实例化版本都可以访问类模板的所有实例化版本。
  + 特定实例化版本：只有友元模板的特定实例化版本可以访问类模板的特定实例化版本。
* 类模板的静态成员变量和静态成员函数与模板参数相关。
* 每个类模板的实例化版本都有自己独立的静态成员。
* 静态成员变量必须在类外进行定义和初始化。
* 静态成员函数可以直接通过类名调用，且只能访问静态成员变量。

**1. 静态成员变量的特性**

* 静态成员变量是类的所有对象共享的。
* 无论创建多少个 MyClass<int> 的对象，MyClass<int> 的静态成员变量只有一个副本。
* 静态成员变量的生命周期从程序开始到程序结束
* 成员模板是指类中的一个成员函数，它本身是一个模板。
* 成员模板可以是普通类或类模板的成员函数。
* 成员模板不能是虚函数。
* 成员模板的实例化由编译器根据调用时的类型参数完成。

当与作为模板类型参数 (T&&) 的右值引用的函数参数一起使用时，forward 会保留有关参数类型的所有详细信息。

**3. 重载解析的规则**

* 编译器会根据以下规则选择最佳匹配的函数：
  1. **转换的优先级**：
     + 编译器会评估每个候选函数所需的类型转换（如隐式类型转换、用户定义转换等）。
     + 转换越少，匹配越好。
  2. **选择最佳匹配**：
     + 如果有一个函数比其他所有函数都更匹配，则选择该函数。
  3. **处理多个匹配**：
     + 如果有多个函数提供同样好的匹配，则：
       - 如果其中只有一个**非模板函数**，则选择该非模板函数。
       - 如果没有非模板函数，但有多个函数模板，则选择**更特化**的函数模板。
       - 否则，调用是**二义性**的，编译器会报错。

在定义任何函数之前，先声明重载集中的每个函数。 这样你就不必担心编译器在看到你想要调用的函数之前是否会实例化一个调用。

**1. 可变参数模板的定义**

* 可变参数模板使用 typename... 或 class... 来声明模板参数包（Template Parameter Pack）。
* 在函数或类中，可以使用参数包来接受任意数量的参数。

**语法：**

cpp

复制

template <typename... Args>

void func(Args... args);

* Args 是一个模板参数包，表示任意数量的类型。
* args 是一个函数参数包，表示任意数量的参数。
* 在定义可变参数函数模板时，必须确保非可变参数版本的声明在作用域内。
* 非可变参数版本是递归的终止条件，防止无限递归。
* 如果没有非可变参数版本，可变参数函数会尝试继续递归，导致编译错误或运行时错误

模板是 C++ 的一个显着特征，也是该库的基础。 模板是编译器用来生成特定类类型或函数的蓝图。 这个过程称为实例化。 我们编写一次模板，编译器会为我们使用该模板的类型或值实例化该模板。

我们可以定义函数模板和类模板。 库算法是函数模板，库容器是类模板。

显式模板参数让我们可以修复一个或多个模板参数的类型或值。 正常转换适用于具有显式模板实参的参数。

模板特化是一种用户提供的模板实例，它将一个或多个模板参数绑定到指定的类型或值。 当存在我们无法在模板定义中使用（或不想使用）的类型时，特化非常有用。

最新版本的 C++ 标准的主要部分是可变参数模板。 可变参数模板可以采用不同数量和类型的参数。 可变参数模板允许我们编写函数，例如容器 emplace 成员和库 make\_shared 函数，将参数传递给对象的构造函数。