**C++学习笔记——模板**

模板是现代C++编程中最广泛使用，但也最难理解的特性。模板涉及的领域繁多，且多数非常抽象。想要理解并有效运用模板，需要大量的编程实践，这是成为优秀C++程序员的必由之路。

**1. 模板特化**

对于既有的类/函数模板，其中部分或全部模板类型，用具体类型来特化。这样，构造对象/调用函数时，如果实参类型符合特化的类型，则优先适用特化版本，而不是通用版本。

按照特化程度划分，如果所有模板类型都被特化，则称为全特化；如果只有部分模板类型被特化，则称为偏特化。

函数模板只能进行全特化；类模板则偏特化、全特化皆可。

**2. std命名空间模板特化**

按照C++编程约定，开发人员不得向std命名空间新增内容，也不得修改其中已有内容。

但是，开发人员可以特化std命名空间已有的模板类/函数。例如：

手机屏幕截图

描述已自动生成

注意：即使违背这一C++编程约定（例如使用上图中被注释的代码），代码仍然可能通过编译并正确运行。但违反这一约定的结果本身就是未定义的，并且它污染了std命名空间，会给协作开发带来巨大的潜在风险。

**3. Argument-dependent Lookup (ADL) / Koenig Lookup**

C++结合命名空间来实例化模板函数的机制。

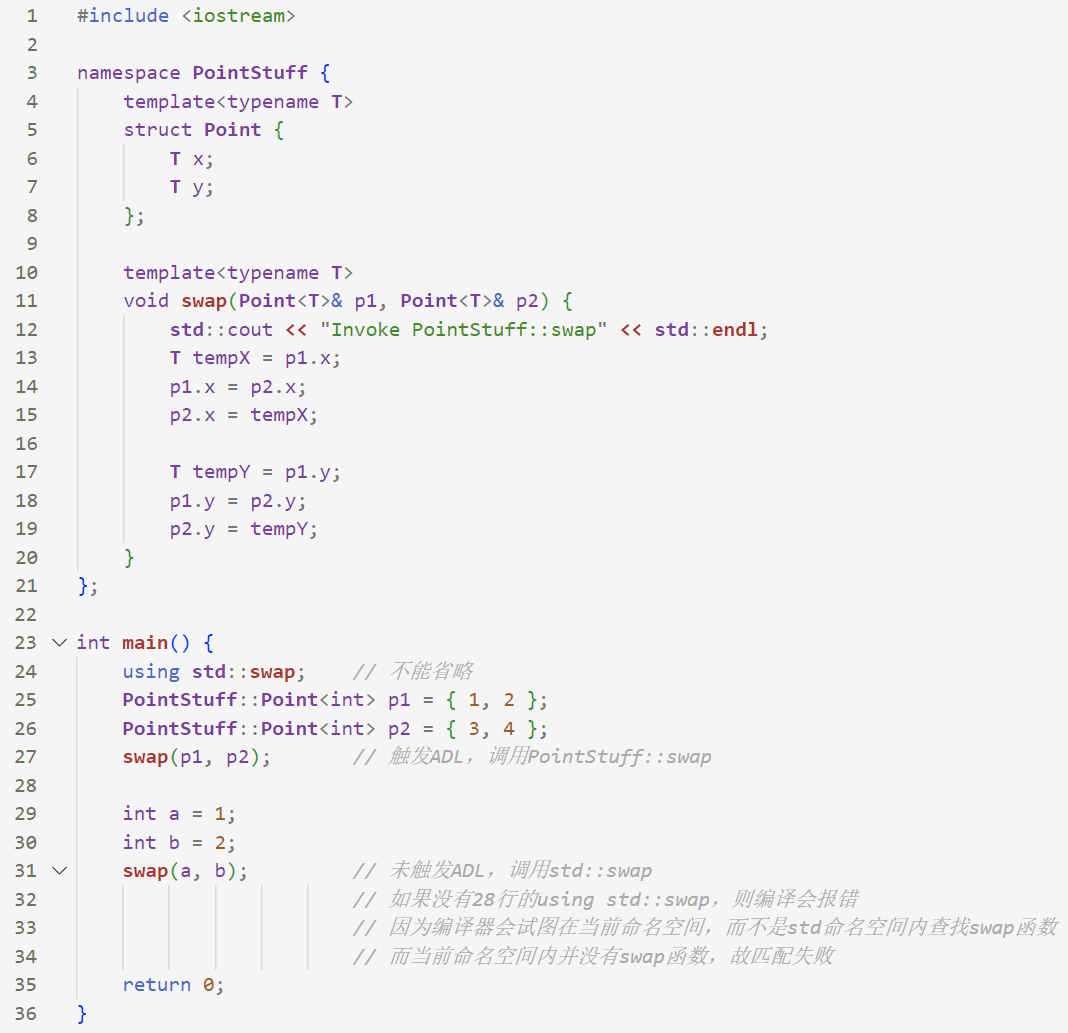
具体来说，如果调用函数时没有使用作用域解析符::，并且不同命名空间内都存在声明格式符合实参类型、顺序的函数，则编译器依照以下次序，从以下命名空间尝试匹配模板函数：

(1) 实参类型声明所在的命名空间。

(2) 用using引入的命名空间。

(3) 全局（默认）命名空间。

示例：



**4. 非类型模板参数（non-type template parameter）**

C++模板参数包括类型模板参数和非类型模板参数。前者十分常见，这里重点介绍后者。

非类型模板参数，表示需要用一个值，而不是一个类型来实例化模板。非类型模板参数实例化的时机，同样发生在编译时而非运行时，这与类型模板参数一样。

在大多数情况下，为了实现相同功能，可以不使用非类型模板参数，而是将参数作为模板类的const成员变量，然后在运行时初始化它。这样的代码可读性更好，但是在运行时才能进行检查。

非类型模板参数的另一个目的是，对于不同的实际模板参数，构造不同的类，表示它们“不是同一个类”。这在有些时候很有用。

比如，假设我们需要定义矩阵模板类。矩阵有行和列，并且只有行和列均相等的两个矩阵，才能进行加法运算。为了表达这种语义，我们可以将矩阵的行和列作为矩阵模板类的非类型模板参数。

这样，“4行3列的矩阵”和“1行3列的矩阵”，就是两个不同的类，所以它们无法进行矩阵加法运算。这一切检查都发生在编译时，并且无需额外的static\_assert。

文本

描述已自动生成

STL中的std::array，就是一个常见的带有非类型模板参数的类。

文本, 信件

描述已自动生成

最后，非类型模板参数在模板元编程（Template Metaprogramming, TMP）中是不可或缺的。

顺带一提，现代C++引入了constexpr，所以一些为了在编译时完成检查而写的非类型模板参数代码，在现代C++中，可以考虑改写为可读性更强的等价constexpr代码。

**5. 模板的声明和实现需要放在同一个文件（一般为头文件）中，不能分离**

通常，一个C++项目分为若干个cpp文件和h文件，每个cpp文件单独编译成每个目标文件，最终将每个cpp文件连接在一起组成最后的单一的可执行文件。这里最重要的点就是：**编译是相对于每个cpp文件而言的**。

C/C++所采用的方法是：**只要给出类的声明，就可以在本源文件中使用该类**。由于每个源文件都是独立的编译单元，在当前源文件中使用但未在此类的实现，就**假设在其他的源文件中实现好了**。

问题在于，模板是按需实例化的——只有在见到具体类型后，编译器才会在编译时实例化模板。

如果采用分离式编译，那么编译器遇到cpp文件中的实现代码时，其不知道具体的类型参数。此时，编译器假定其他的源文件中提供了具体的类型参数来实例化该模板，因此，编译器选择直接跳过。而其他文件实际上也并没有实例化该模板。结果是，**编译通过，但是这部分模板源代码，根本没有生成目标代码**。

而到了之后的连接环节，其他文件（如main.cpp）用到了这个模板时，由于找不到该模板实现所对应的二进制代码（因为编译时没有产生），所以连接会失败。

所以，模板的声明和实现需要放在同一个文件（一般为头文件）中，不能分离。