模板和标准模板库(STL)

闵卫

minwei@tedu.cn

----------------------

一、引论

1.泛型编程与非泛型编程

泛：广泛、一般、通用、普适

型：数据类型

非泛型的编程：

用与数据类型相关的解决方法，解决与数据类型无关的逻辑问题。

-------------------------------------

2.C/C++是静态类型语言

静态类型语言：运行时结构不可变的语言，如Java、C、C++。

动态类型语言：运行时代码可以根据某些条件改变自身结构的语言。  
主要动态语言：Object-C、C#、JavaScript、PHP、Python、Erlang。

------------------------------

静态语言的优缺点：

好处：性能优越。类型决定了数据的内存布局、编码方式和计算规则，而所有这一切全部在编译阶段决定，运行时无需处理任何与数据类型相关的问题，借以提升运行时性能。

----------------

坏处：不得不用与类型相关的方法去解决与类型无关的问题，阻碍程序员编写类型通用的算法和数据结构。

参见：typed.cpp

=====================================

解决方法一：普通宏定义

宏定义只是在预处理器的作用下，针对源代码的文本替换，其本身并不具备函数语义。因此借助于参数宏，即宏函数，可以在某种程度上摆脱静态类型的约束和限制，编写与类型无关的逻辑实现，但同时也丧失了类型的安全性。

参见：untyped.cpp

---------------

解决方法二：宏定义构建通用代码框架

利用宏定义构建通用代码框架，让预处理器将其扩展为针对不同类型的具体版本。将宏的一般性和函数的类型安全型完美地结合起来。该方法的局限和不足在于必须显式地扩展宏。

参见：macro.cpp

==============================================

1. 函数模板

模板是一种指导编译器生成函数的方法，一种底层二进制中不存在的抽象概念。

1.函数模板定义

template<typename 类型形参1，typename 类型形参2, ...>

返回类型 函数模板名(调用形参表){

函数体

}

函数模板的返回类型、形参表以及函数体中都可以使用该模板的类型形参。

---------------

2.函数模板的使用

（1）函数模板的实例化（使用函数模板时才实例化）：

函数模板名<类型实参1, 类型实参2, ...>(调用实参表);

1. 有时候函数模板的类型实参表或调用实参表可以为空，但尖括号和圆括号不能不写。

---------------------------

3.类型参数

1)类型形参：类型占位符，以字母或下划线开始，由字母、下划线或数字组成，不能包含特殊字符，也不能和关键字冲突。每个类型形参前面必须冠以typename/class关键字。

2)类型实参：必须是具体类型，既可以是基本类型，也可以是类类型，甚至可以是模板实例化后的类型。一个类型实参可不可以传递给一个类型形参，唯一的条件就是它必须满足具体函数的实现对该类型的要求(类类型，可能需要重载操作符)。

参见：targ.cpp

----------------------------

举例：

//函数模板定义

template<typename A, typename b, typename \_C>

A func(b arg){

\_C var;

//...(函数体)

}

//实例化函数模板

int i = func<int, double，Student>(1.76，Student("张飞", 22));

参见：ftmpl.cpp

--------------------------

4.二次编译(延迟编译)

（1）每个函数模板事实上至少要被编译两次。

（2）一次是在实例化之前，先检查函数模板本身，查看语法是否正确。

（3）另一次是在实例化期间，结合所使用的类型实参，再次检查模板代码，查看是否所有的调用都有效。

（4）但是只有在模板实例化时，才会进行第二次编译，此时才真正产生二进制形式的机器指令。

（5）作为第一次编译的结果，仅仅是在编译器内部形成一个用于描述该函数模板的数据结构，即所谓的模板内部表示。

1. 类型实参的隐式推断

(1)如果函数模板的模板参数(尖括号里的)，全都在该模板调用参数(圆括号里的)里出现，那么在调用该函数模板时，即使不显式指定模板参数，编译器也有能力根据调用参数的类型隐式推断出正确的模板参数，以获得与普通函数调用一致的语法表达。

举例：

相关： template<typename A, typename V> void foo(A x, V y){...}

不相关： template<typename A, typename V> void foo(A x){...}

----------------------

补充：非基本类型数据会根据语义推断

1. 数组名有时表示数组首地址，有时表示数组整体。
2. 当数组做形参时，数组名表示数组首地址；当数组的引用做形参时，数组名表示数组整体
3. 当字符串字面值做形参时，编译器会处理为首地址；当其引用做形参时，编译器会处理为一个字符数组.

举例："hello" ==> char const\*(PKc);引用"hello" ==> const char[6](A6\_c)

-----------------------

(2)如果编译器隐式推断的类型和程序设计者所期望的类型不一致，可能导致结果错误，这时就不要再使用隐式推断，或者也可以先做类型转换。

(3)以下三种情况，不能隐式推断，必须显式指定模板参数

a.不是全部模板参数都与调用参数的类型相关

template<typename A, typename V> void foo(A arg){ ...; V var; ... }

foo(123); // 错误

foo<int,string>(123);

b.隐式推断的同时不允许隐式类型转换

template<typename T> void foo(T x, T y){ ... }

foo(123, 4.56); // 错误

foo<double>(123,4.56);

foo((double)123,4.56); // 显式类型转换可以隐式推断

c.返回类型不能隐式推断

template<typename A,typename R> R foo(A arg){ ... }

int ret = foo(1.23); // 错误

int ret = foo<double,int>(1.23);

参见：ftmpl.cpp、deduction.cpp

----------------------------------------

6.重载

1)和普通函数一样，函数模板也可以重载，在重载解析的过程中，编译器会优先选择类型约束性较强的版本。

--------------------

注意：隐式推断并不是，推断模板参数与调用参数类型相同；而是推断模板参数类型，使之与调用实参匹配。

举例：

template<typename T>

T const& max(T const& x, T const& y){ ... } //①

template<typename T>

T\* const& max(T\* const& x, T\* const& y){ ... } //②

...

int a = 123, b = 456;

char const\* x = "AB";

char const\* y = "A";

cout << ::max(a, b) << endl; //①; T ==> int

cout << \*::max(&a, &b) << endl; //②；T ==> int,~~T ==> int\*~~

cout << ::max<>(x, y) << endl; //②; T ==> char const,~~T ==> char cosnt\*~~

------------------

2)模板函数和普通函数也可以构成重载，在重载解析的过程中，编译器会优先选择普通函数，除非函数模板能够产生具有更好匹配性的函数实例。

3)如果显式指定空的模板实参表，那么编译器就只在模板函数中挑选类型约束性较强者。

1. 显式指定的模板参数必须与所选择的重载版本中的调用参数的类型保持一致。

---------------------

类型一致的情况下，才选择约束性较强的。

举例：

template<typename T>

T const& max(T const& x, T const& y){ ... } //①

template<typename T>

T\* const& max(T\* const& x, T\* const& y){ ... } //②

...

char const\* x = "AB";

char const\* y = "A";

//显示指明了T ==> char const\*，结果只有代入①中，类型才与原数据匹配

cout << ::max<char const\*>(x, y) << endl; //①

---------------------

1. 即使是在函数模板的实例化函数中，编译器仍然优选选择普通函数，前提是该普通函数在第一次编译时可见(即出现在调用函数前)。

注解：一次编译时生成候选函数列表，只有被调用的普通函数写在调用者模板函数前面才可见。

1. 在重载函数模板时，应该尽可能把改变限制在参数的个数和类型的约束性上，对其引用属性最好始终如一，避免因为临时变量的产生而导致的非预期后果。

总结：同一个程序中，避免指针和引用混用。

原因：如果返回引用的函数调用了返回指针的函数，就会出现返回临时变量(指针)引用的情况。

参见：overload.cpp