C++ 模板 5: 类模板的继承和成员函数 Yinli 大三了 3 人赞同了该文章 基础 设计 技术 模板实例化 自动返回值类型 静态多态与动 可变参数模板 模板元論程 奇异递归模板 折叠表达式 类型萃取 表达式模板

已关注

模式与惯用法 依赖名称 constexpr if 特例化 - 篇文章介绍了关于这些模板的基本知识。这篇文章说明对类模板的继承和类模板成员函数的实 例化中的特性,这些特性可能会令你感到 surprise。 下面是第一个 surprise,至少我是 surprise 了: 类模板的继承成员函数不可用 个简单的例子:

// inheritance.cpp #include <iostream>

ublic: void func()
{ std::cout << "func\n"; class Derived : public Base oublic: void callBase()
{
 func(); };

std::cout << '\n';
Derived derived;
derived.callBase();
std::cout << '\n';</pre>

的。 File Edit View Bookmarks Serainer@linux:~> inheritance s Settings Help

#include <iostream>
template <typename T>
class Base

void func()

std::cout << "func\n";

template <typename T>
class Derived: public Base<T>

void callBase() func();

std::cout << '\n';
Derived<int> derive
derived.callBase();
std::cout << '\n';</pre>

我猜编译器的报错会让你 surprise。

void func2() const

void func3() const

int main()

上面的代码实现了类 Base 和 Derived 。Derived 是从 Base 公共派生出来,因此可以在其成员函数 callBase ((2) 行) 中使用类 Base 的方法 func 。程序的输出如下,没有什么要补充

rainer@linux:~>

rainer : bash

让 Base 成为类模板完全改变了程序行为。

File Edit View Bookmarks Settings Help
ratner@linux:->> g++ templateInher(tance.cpp -o templateInher(tance
templateInher(tance.cpp: In member function 'void Derived<):::callBase()':
templateInher(tance.cpp: I?I'd: error: there are no arguments to 'func' that depend o
n a templateInher tance.cpp: I?I'd: error: there are no arguments to 'func' that depend o
n a templateInher and tance there is a declaration of a templateInher and the available [-fpermissive]
func(): templateInheritance.cpp:17:14: note: (if you use '-fpermissive', G++ will accept you r code, but allowing the use of an undeclared name is deprecated) rainer@linux:-> ■ rainer : bash

错误信息中的 there are no arguments to 'func' that depend on a template parameter, so a declaration of 'func' must be available 给出了第一个提示。 func 是一个非依赖名称, 即它的名称不依赖于模板参数 ⊤。非依赖性名称在模板定义时被直找和绑定,只有依赖名称在模板实例化时被直找和绑定,因此,编译器不会在依赖 ⊤ 的基类 Base(T) 中直找,在类模板之外也 没有 func 这个名字。 这个过程叫做两阶段查找(Two-phase lookup)。第一阶段专门负责查找非依赖名称;第二阶段

负责查找依赖名称。 有三种方法可以将名称查询扩展到依赖的基类。下面的例子用到了这三种方法。 // templateInherita
#include <iostream>

template <typenam public: oid func1() const { std::cout << "func1()\n";

std::cout << "func3()\n"; template <typename T>
class Derived: public Base<T> ublic: using Base<T>::func2;
void callAllBaseFunctions() {
this->func1(); func2(); Base<T>::func3(); int main() { std::cout << '\n'; Derived<int> derived; derived.callAllBaseFunctions();

1. **使名称具有依赖性。**第 (1) 行的调用 this->func1 是依赖性的,这是隐式依赖。在这种情况 - IKI (1994年) TRANS (1994年) 第1(7) 1339年7 1415年 (日本学年) 1415年 (1994年) (1994年) (1994年) (1994年) (1994年) (1994年) (1994年) (1994年) (1

3. 完全限定地调用名称。 完全限定地调用 func3 (第(3)行) 会破坏虚调用,带来新的 应该用哪种方法? 一般来说,我更喜欢第一个,即 this->func1 。当基类需要重命名时,这个方 法也不会失效。 最后,程序输出如下: File Edit View Bookmarks Settings Heritainer@linux:~> templateInheritance2

rainer@linux:~>

void func() { std::cout << "func\n"; }
void func2(); // not defined (1)</pre>

int main()

std::cout << '\n';
Lazy<int> lazy;
lazy.func();
std::cout << '\n';</pre>

// LazyInstantiati #include **<cstddef>** class Array1

ublic: int getSize() cons return 10; vate: int elem[10];

ublic:

template <typename T, std::size_t N>
class Array2

rainer : bash 成员函数实例化的惰性规则 类模板的成员函数的实例化只在需要时发生。请看: // Lazy.cpp #include <iostream> nplate<class T> ruct Lazy

尽管 Lazy 类的成员函数 func2() (1) 只被声明而没有定义,但编译器接受了这个程序,因为没有必要定义 func2。 成员函数实例化过程的这种惰性有两个值得提到的特性。 节约资源 例如,当你实例化对各种类型的类模板 Array2 ,只有使用的成员函数被实例化了。对于非模板类 Array2 来说,这种惰性并不存在。请看这个用 C++ Insights 展示的例子。

std::size_t getSize() const
{ rivate: T elem[N]; }; int main() Array2<int, 5> myArr1; Array2<double, 5> myArr2; // (1) myArr2.getSize(); // (2) 类模板 Array2 的成员函数 getSize() 只对 myArr2 (1)进行了实例化。这个实例化是由调用 myArr2.getSize() (2) 引起的。 C++ Insights 展示了这一过程。以下截图中的关键是第 40 行和 59 行。 33 /* First instantiated from: 1
34 #ifdef INSIGHTS_USE_TEMPLATE
35 template<>
36 class Array2<int, 5>

public: inline std::size_t getSize() const;

/* Tirst instantiated from insights.cpp:28 */
eidef INSIGNTS_USE_TEMPLATE
templateO
class ArrayScdouble, 5> {

实例化类模板时,可以使用不支持所有成员函数的模板实参。只要不调用不支持的成员函数,编译

explicit Matrix(std::initializer_list<T> inList) : data(inList) {}

void printAll() const // (2)

for (const auto& d : data) std::cout << d << " ";

const Matrix<int> myMatrix1({ 1, 2, 3, 4, 5 });
myMatrix1.printAll(); // (3)

const Matrix<int> myMatrix2({ 10, 11, 12, 13 });
myMatrix2.printAll(); // (4)

std::cout << "\n\n";
const Matrix<Matrix<int>> myMatrix3({ myMatrix1, myMatrix2 });
// myMatrix3.printALL(); ERROR (5)

public: inline std::size_t getSize() con {

return 5UL;

private:
double elem[5];
public:
// inline Array2()

类模板的部分使用

#include <iostream>
#include <vector>

template <typename T>
class Matrix

std::vector<T> data;

std::cout << "\n\n";

器就不会报错。

ublic:

}; t main() std::cout << '\n';

类模板 Matrix (1)有一个类型参数 τ , 用 std::vector 保存数据, 并且可以通过 std::initalizer_list 进行初始化。 Matrix 支持成员函数 printAll() 来显示其所有成员。 (3)和 (4)显示了它的用法。 輸出操作符没有为 Matrix 重載。因此,虽然我们可以创建由其他 Matrix 对象作为成员的 myMatrix3 , 但不能显示它们(5)。 → ✓ rainer: bash — Konsole <> ✓ ∧ ⊗ File Edit View Bookmarks Settings Help rainer@seminar:~> classTemplatePartial

> 1 2 3 4 5 10 11 12 13 rainer@seminar:~>

启用 (5) 会导致一个相当冗长的错误信息,大约有 274 行:

Rie Edit Vew Bookmans Setings Help
rainer@seminar:-> g++ classTemplatePartial.cpp -o classTemplatePartial 2>&| | wc -l

cpp.com,原作者 Rair

知乎 @Yirki

下一篇文章将介绍关于别名模板和模板形参的内容。 导航

274 rainer@seminar:->

本文经授权翻译自 w

-篇

上一篇 原文