Technology Tarena 达内科技

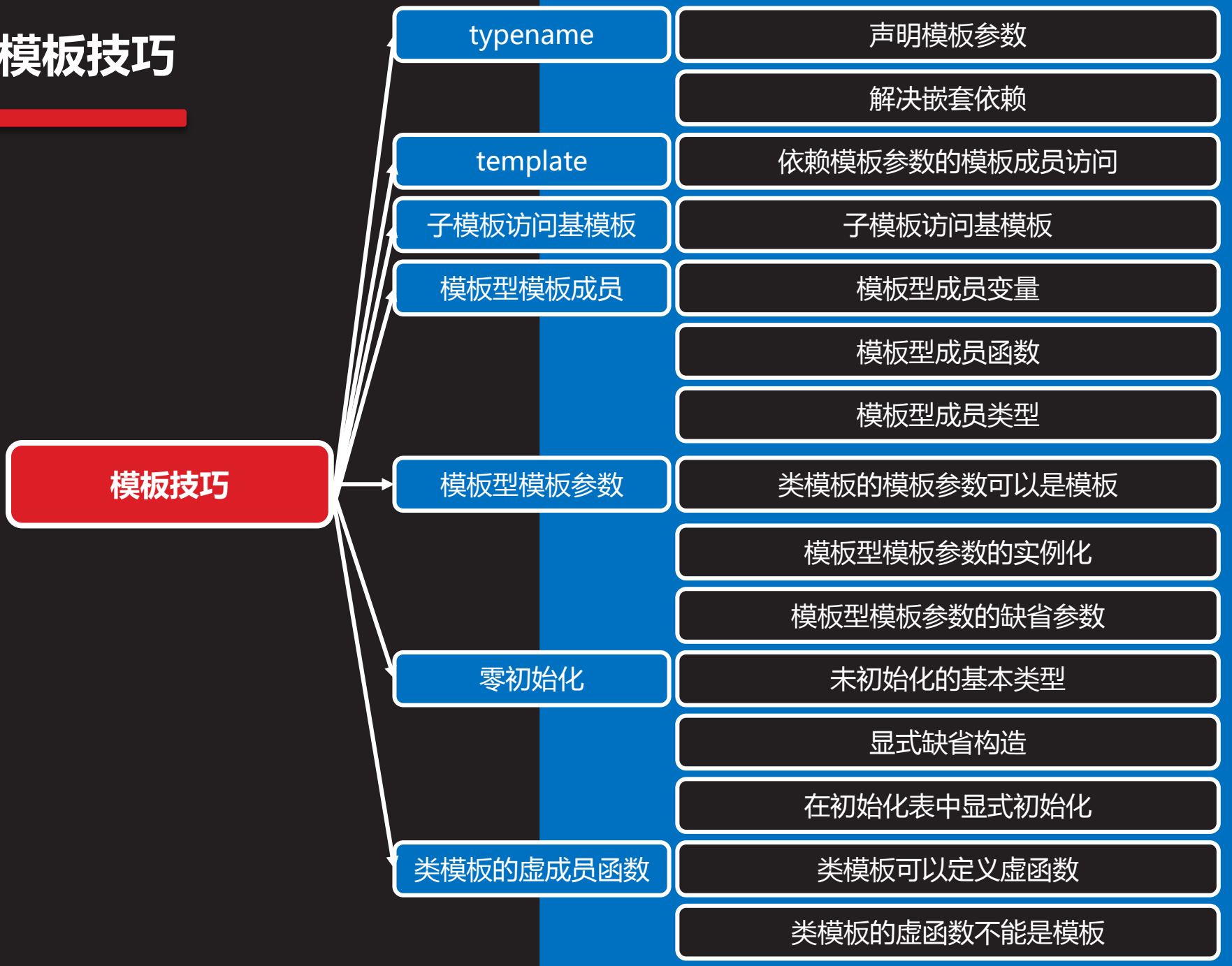
模板和STL

TEMPLATE & STL DAY02

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	模板技巧
	10:30 ~ 11:20	
	11:30 ~ 12:20	
下午	14:00 ~ 14:50	模板实战
	15:00 ~ 15:50	
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑

模板技巧


模板技巧
typename
声明模板参数
解决嵌套依赖
template
依赖模板参数的模板成员访问
子模板访问基模板
子模板访问基模板
模板型模板成员
模板型成员变量
模板型成员函数
模板型成员类型
模板型模板参数
类模板的模板参数可以是模板
模板型模板参数的实例化
模板型模板参数的缺省参数
零初始化
未初始化的基本类型
显式缺省构造
在初始化表中显式初始化
类模板的虚成员函数
类模板可以定义虚函数
类模板的虚函数不能是模板

Technology Tarena 达内科技

typename

声明模板参数

- 声明模板参数：template<typename T> ...，只有在这种语境下，typename关键字才可以和class关键字互换
  - class关键字
    - 声明类：class A { ... };
    - 声明模板参数：template<class T> ...
  - typename关键字
    - 声明模板参数：template<typename T> ...
    - 解决嵌套依赖：typename T::A a;
  - 无论是声明模板参数还是解决嵌套依赖，都不能使用struct关键字

等价

知识讲解

解决嵌套依赖

- 在第一次编译模板代码时，模板参数的具体类型尚不明确，编译器会把依赖于模板参数的嵌套类型理解为某个类的静态成员变量。因此当它看到代码中使用这样的标识符声明其它变量时，会报告错误，这就叫做嵌套依赖
  - class X { public: class A {}; }; // A是X的嵌套类型
  - template<typename T> void foo (void) { … T::A a … } // 错误
  - foo<X> ();
- typename关键字旨在告诉编译器，所引用的标识符是个类型名，可以声明变量，具体类型等到实例化时再定
  - template<typename T> void foo (void) { … typename T::A a … }

知识讲解

通过typename解决嵌套依赖

【参见：TTS COOKBOOK 】

课堂练习

- 通过typename解决嵌套依赖

Technology Tarena 达内科技

template

依赖模板参数的模板成员访问

- 在模板代码中，通过依赖于模板参数的对象、引用或指针，访问其带有模板特性的成员，编译器常常因为无法正确理解模板参数列表的左右尖括号，而报告编译错误
  - class A {
    public: template<typename T> void foo (void) { ... } };
  - template<typename T> void foo (T& r, T* p) {
    r.foo<int> (); p->foo<int> (); } // 错误
  - A a; foo (a, &a);
- 在模板名前面加上template关键字，意在告诉编译器其后的名称是一个模板，编译器就可以正确理解“<>”了
  - template<typename T> void foo (T& r, T* p) {
    r.template foo<int> (); p->template foo<int> (); }

依赖模板参数的模板成员访问

【参见：TTS COOKBOOK 】

课堂练习

- 依赖模板参数的模板成员访问

Technology Tarena 达内科技

子模板访问基模板

子模板访问基模板

- 在子类模板中直接访问那些依赖于模板参数的基类模板的成员，编译器在第一次编译时，通常会因为基类类型不明确而只在子类和全局作用域中搜索所引用的符号
  - template<typename T> class A {
    protected:
    void foo (void) { ... }
    void exit (int status) { .... } };
  - template<typename T> class B : public A<T> {
    public:
    void bar (void) {
    foo (); // 错误
    exit (EXIT_SUCCESS); } // 所调用并非基类exit成员
    };

知识讲解

子模板访问基模板（续1）

- 在子类模板中可以通过作用域限定符，或者显式使用this指针，迫使编译器到基类作用域中搜索所引用的符号
  - template<typename T> class B : public A<T> {
    public:
    void bar (void) {
    A<T>::foo ();
    this->exit (EXIT_SUCCESS);
    }
    };

知识讲解

子模板访问基模板

【参见：TTS COOKBOOK 】

课堂练习

- 子模板访问基模板

Technology Tarena 达内科技

模板型模板成员

模板型成员变量

- 类模板的成员变量，如果其类型又源自一个类模板的实例化类，那么它就是一个模板型模板成员变量
  - template<typename T> class List {
    public:
    void push_back (T const& elem) { ... }
    };
  - template<typename T> class Stack {
    public:
    void push (T const& elem) { m_list.push_back (elem); }
    private:
    List<T> m_list;
    };
  - Stack<int> si;

知识讲解

模板型成员函数

- 类模板的成员函数，如果除了类模板的模板参数以外，还需要其它模板参数，那么它就是一个模板型模板成员函数
  - template<typename T> class Stack {
    public:
    template<typename E>
    Stack (Stack<E> const& that) { ... }
    template<typename E>
    Stack<T>& operator= (Stack<E> const& rhs);
    };
  - template<typename T>
    template<typename E>
    Stack<T>& Stack<T>::operator= (
    Stack<E> const& rhs) { ... }

知识讲解

模板型成员类型

- 类模板的成员类型，如果除了类模板的模板参数以外，还需要其它模板参数，那么它就是一个模板型模板成员类型
  - template<typename X> class A {
    template<typename Y> class B {
    template<typename Z> class C;
    Y m_y; };
    X m_x; };
  - template<typename X>
    template<typename Y>
    template<typename Z>
    class A<X>::B<Y>::C {
    Z m_z; };

知识讲解

模板型模板成员

【参见：TTS COOKBOOK 】

课堂练习

- 模板型模板成员

Technology Tarena 达内科技

模板型模板参数

类模板的模板参数可以是模板

- 类模板的模板参数本身又可以是个模板
  - template<typename T> class List {
    public: void push_back (T const& elem) { ... } };
  - template<template<typename E> class C>
    class Stack {
    public:
        void push (int const& n) {
            m_cntr.push_back (n); }
    private:
        C<int> m_cntr;
    };
  - Stack<List> stack;

知识讲解

模板型模板参数的实例化

- 可以使用类模板的其它参数，实例化其模板型模板参数
  - template<typename T> class List {
    public: void push_back (T const& elem) { ... } };
  - template<typename T,
    template<typename> class C> class Stack {
    public:
    void push (T const& elem) {
    m_cntr.push_back (elem); }
    private:
    C<T> m_cntr;
    };
  - Stack<int, List> si;

知识讲解

模板型模板参数的缺省参数

- 模板型模板参数的模板参数也可以带有缺省值
  - template<typename T> class List {
    public: void push_back (T const& elem) { ... } };
  - template<typename T,
    template<typename> class C = List> class Stack {
    public:
    void push (T const& elem) {
    m_cntr.push_back (elem); }
    private:
    C<T> m_cntr;
    };
  - Stack<int> si;

知识讲解

模板型模板参数

【参见：TTS COOKBOOK 】

课堂练习

- 模板型模板参数

Technology Tarena 达内科技

零初始化

未初始化的基本类型

- 基本类型不存在缺省构造函数，未被显式初始化的局部变量都具有一个不确定的值
  - int var; // 未初始化(不确定的值)
- 包含(自定义或系统提供的)缺省构造函数的类，在未被显示初始化的情况下，都会有一个确定的缺省初始化状态
  - Student var; // 缺省初始化(调用缺省构造函数)
- 这样就会在模板的实现中产生不一致的语法语义
  - template<typename T> void foo (void) {
    T var; // 隐式缺省构造
    cout << var << endl; }
  - foo<int> (); // 不确定的值
  - foo<Student> (); // 确定的值

知识讲解

显式缺省构造

- 如果希望模板中所有参数化类型的变量，无论是类类型还是基本类型，都能以缺省方式获得初始化，就必须对其进行显示地缺省构造
  - template<typename T> void foo (void) {
    T var = T (); // 显式缺省构造
    cout << var << endl; }
  - void foo<int> (void) {
    int var = int (); // 用适当类型的0初始化
    cout << var << endl; } // 确定的值
  - void foo<Student> (void) {
    Student var = Student (); // 用缺省构造函数初始化
    cout << var << endl; } // 确定的值

知识讲解

在初始化表中显式初始化

- 对于类模板，可以在其缺省构造函数的初始化列表中，显式初始化其各个成员变量，无论它是类类型的还是基本类型的
  - template<typename T> class A {
    public:
    A (void) : m_var () {}
    T m_var;
    };
  - A<int> ai; // 用适当类型的0初始化m_var成员
    cout << ai.m_var << endl; // 确定的值
  - A<Student> as; // 用缺省构造函数初始化m_var成员
    cout << as.m_var << endl; // 确定的值

知识讲解

零初始化

【参见：TTS COOKBOOK 】

课堂练习

- 零初始化

Technology Tarena 达内科技

类模板的虚成员函数

类模板可以定义虚函数

- 类模板的普通成员函数可以是虚函数，即可以为类模板定义虚成员函数。和普通类的虚成员函数一样，类模板的虚成员函数亦可表现出多态性
  - template<typename PEN> class Rect {
    public:
    virtual void draw (PEN const& pen) { ... } };
  - template<typename PEN>
    class RoundRect : public Rect<PEN> {
    public:
    void draw (PEN const& pen) { ... } };
  - Rect<SolidPen>* rect =
    new RoundRect<SolidPen> (...);
    rect->draw (SolidPen (...));

知识讲解

类模板的虚函数不能是模板

- 无论是类还是类模板，其虚成员函数都不能是模板函数
  - 基于虚函数的多态机制，需要一个名为虚函数表的函数指针数组。该数组在类被编译或类模板被实例化的过程中产生，而此时那些模板形式的成员函数尚未被实例化，其入口地址和重载版本的个数，要等到编译器处理完对该函数的所有调用以后才能确定。成员函数模板的延迟编译阻碍了虚函数表的静态构建
  - template<typename PEN> class Rect {
    public:
    template<typename BRUSH>
    virtual void draw (PEN const& pen,
    BRUSH const& brush) { ... } // 错误
    };

知识讲解

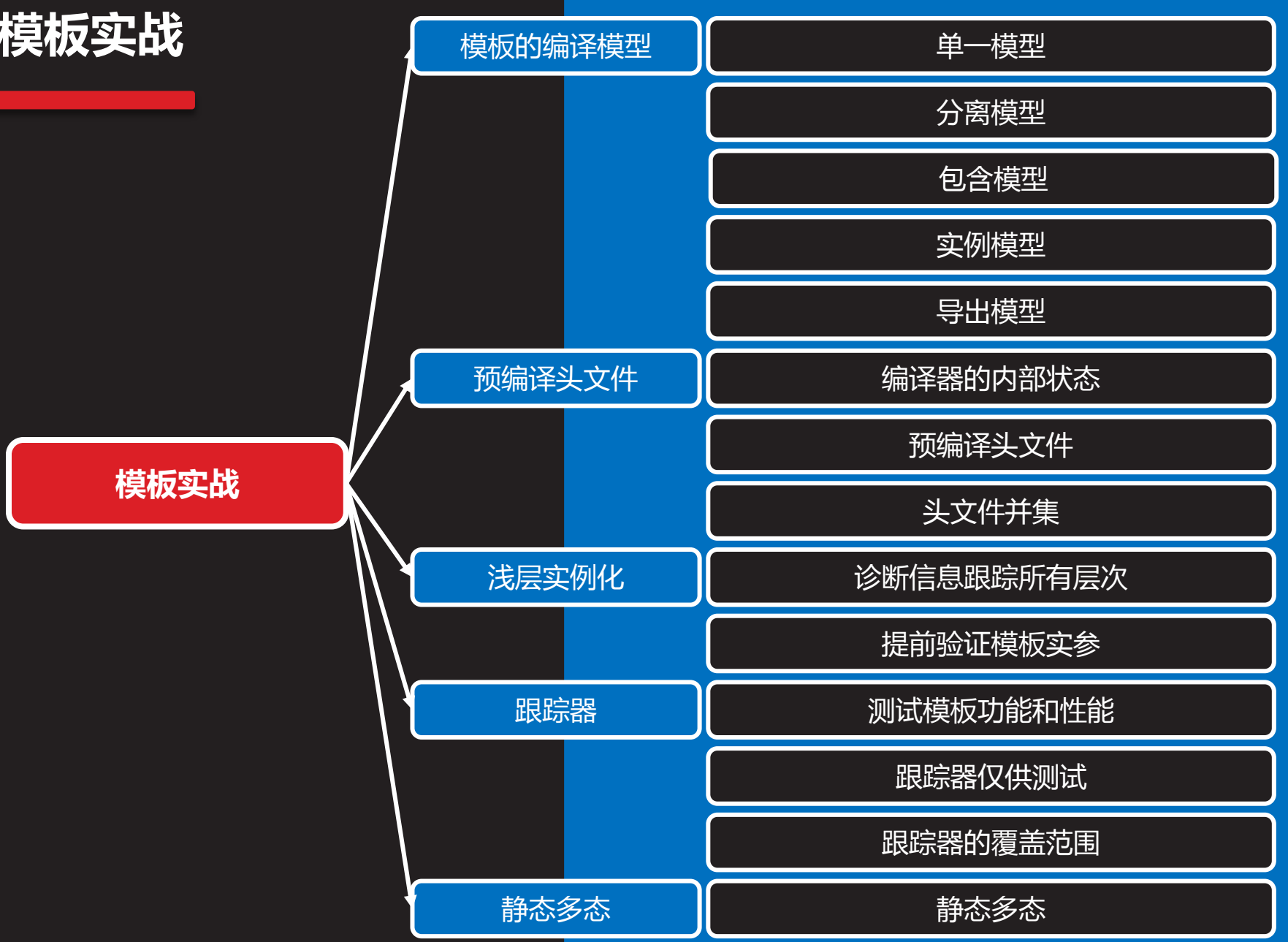
类模板的虚成员函数

【参见：TTS COOKBOOK 】

课堂练习

- 类模板的虚成员函数

模板实战


模板实战
模板的编译模型
单一模型
分离模型
包含模型
实例模型
导出模型
预编译头文件
编译器的内部状态
预编译头文件
头文件并集
浅层实例化
诊断信息跟踪所有层次
提前验证模板实参
跟踪器
测试模板功能和性能
跟踪器仅供测试
跟踪器的覆盖范围
静态多态
静态多态

Technology Tarena 达内科技

模板的编译模型

单一模型

- 将模板的声明、定义和实例化放在单一的编译单元中，无论编译还是链接，其结果总是对的
  - // 在声明模板的同时给出定义

```
template<typename T> class TypeOf {
public:
    ostream& operator>> (ostream& os) const {
        return os << typeid (T).name (); }
};
template<typename T> string typeOf (void) {
    return typeid (T).name ();
}
```

  - // 根据特定的模板实参对模板进行实例化

```
TypeOf<int> () >> cout << ' ' << typeOf<int> () << endl;
TypeOf<double> () >> cout << ' ' << typeOf<double> () << endl;
TypeOf<string> () >> cout << ' ' << typeOf<string> () << endl;
```

知识讲解

分离模型

- 每个C或C++源文件都是被单独编译的。编译器在编译模板定义文件时所生成的模板内部表示，此刻早已荡然无存。因此所有基于模板实例的类和函数，编译器只能假设它们被定义在其它模块中，并产生一个指向该(不存在的)定义的引用，期待链接器能在日后解决此问题。但事实上，由于模板的内部表示并没有真正化身为可执行的指令代码，链接器最终报告“未定义的引用”错误
- main.cpp
  - TypeOf<int> () >> cout << ' ' << typeOf<int> () << endl;
    TypeOf<double> () >> cout << ' ' << typeOf<double> ()
    << endl;
    TypeOf<string> () >> cout << ' ' << typeOf<string> ()
    << endl;

包含模型

- 把模板定义文件包含在模板声明文件的内部(声明之后)，即让模板的定义和声明都位于同一个头文件中
- 任何希望使用模板的程序都必须包含模板的声明文件，而模板定义文件亦因包含而内嵌于该声明文件，最终与模板的实例化代码同处一个编译单元
- 模板的声明、定义与实例化同处一个编译单元中，编译器有能力对模板进行正确地实例化，没有任何链接错误
- 包含模型会延长总体的编译时间，而且必须向模板用户提供模板的定义源文件

知识讲解

实例模型

- 在模板定义文件中使用实例化指示符，强制编译器在编译该文件时，即根据特定的模板实参对模板进行实例化
  - template class TypeOf<int>;
    template class TypeOf<string>;
  - template string typeOf<int> (void);
    template string typeOf<string> (void);
- 根据特定的模板实参对模板进行实例化的工作已在模板定义文件中完成。使用该模板时无需再行实例化，直接引用具体类型，调用具体函数即可
- 显式实例化的类型总是有限的，即便只考虑项目中有限类型的情况，也必须仔细跟踪每个需要实例化的类或函数，这对于大型项目而言，将成为十分繁琐的工作

知识讲解

导出模型

- 通过export关键字声明模板为导出，即可在其定义不可见的编译单元中，实例化该模板
  - export template<typename T> ...
- 编译器对导出型模板的内部表示会有更加持久的记忆，即便该模板的实例化与它的定义不在同一个编译单元中，亦能被正确地编译，不存在任何链接错误
- 导出模型在很大程度上增加了编译器的实现难度。截至目前，真正支持export关键字的编译器少之又少(仅Comeau C/C++和 Intel 7.x编译器支持)，GNU和Microsoft的C++编译器都不支持，而且在C++2011标准中，此特性已被废除，export关键字也被改作它用

知识讲解

模板的编译模型

【参见：TTS COOKBOOK 】

课堂练习

- 模板的编译模型

Technology Tarena 达内科技

预编译头文件

编译器的内部状态

- 当编译一个文件时，编译器从文件的开头一直扫描到文件的结束。对其所见到的每个标记(包括来自头文件的标记)，编译器都会更新其内部状态，以反应该标记所表达的语义。编译器的内部状态直接决定了其在目标文件中所产生的代码
- 如果有多个需要编译的文件，其前N行代码完全相同，那么编译器在编译第一个文件时，就可以把与这N行代码相对应的内部状态保存在一个文件中。待其编译剩下的文件时，先从这个文件中重新加载事先保存好的内部状态，然后从第N+1行开始编译。从一个文件中加载与N行代码相对应的内部状态，比实际编译N行代码快得多

预编译头文件

- 多个文件的前N行代码完全相同，这种情况更多地表现为，以相同的顺序包含一组相同的头文件，特别是那些几乎每个程序都会用到的标准头文件，如stdio.h或iostream等。先编译这组头文件，并将编译过程中所形成的内部状态保存在一个专门的文件中，然后在编译所有以包含这组头文件为前N行代码的文件时，直接从这个专门的文件中加载与头文件有关的信息，避免重复编译完全相同的内容，缩短编译时间，提高编译速度。这种编译技术称为预编译头技术。目前大多数C/C++编译器都支持该技术，而那个用于保存编译器内部状态的专门文件则称为预编译头文件(GNU编译器生成的.gch文件或Microsoft编译器生成的.pch文件)

知识讲解

头文件并集

- 在使用预编译头技术时，所包含的头文件应该是各模块所需头文件的并集。这对于每个具体模块而言可能会有部分冗余，但就整体而言要比只选择有用的头文件，能够获得更快的编译速度
- 更一般化的做法是，把那些包括标准头文件在内的，不会或很少修改的头文件及代码集中放在一个头文件中；编译该头文件，得到预编译头文件；然后在每个需要使用其中内容的文件的第一行包含该头文件，编译器会自动为其加载预编译头文件，而不是重新编译该头文件
  - gcc -c precompile.h -> precompile.h.gch

知识讲解

预编译头文件

【参见：TTS COOKBOOK 】

课堂练习

- 预编译头文件

Technology Tarena 达内科技

浅层实例化

诊断信息跟踪所有层次

- 底层模板往往是在实例化上层模板的过程中被实例化的。因此，一旦因为底层模板的错误而导致上层模板不能被正确地实例化，编译器所给出的诊断信息通常会包含对产生这个问题的所有层次的完整跟踪。程序员往往很难从这么多信息中快速找到症结所在
  - template<typename P> void zero (P const& p) { *p = 0; }
  - template<typename P> void core (P const& p) { zero (p); }
  - template<typename T> void middle (
    typename T::P const& p) { core (p); }
  - template<typename T> void shell (void) {
    typename T::P p; middle<T> (p); }
  - class Object { public: typedef int P; };
  - shell<Object> ();

知识讲解

提前验证模板实参

- 在程序中增加一些不可能执行到哑代码，只是为了在实例化上层模板时，提前验证一下所用模板实参能否满足底层模板所需要的操作，这种编程技巧称为浅层实例化
  - template<typename T> void shell (void) {
    class Shallow {
    void deref (T::P const& p) {
    *p = 0;
    }
    };
    typename T::P p;
    middle<T> (p);
    }

浅层实例化

【参见：TTS COOKBOOK 】

课堂练习

- 浅层实例化

Technology Tarena 达内科技

跟踪器

测试模板功能和性能

- 跟踪器是一个用户自定义的类，可以做为测试模板功能和性能的类型实参
  - class Tracer {
    public:
    Tracer (void);
    Tracer (int data);
    Tracer (Tracer const& that);
    Tracer& operator= (Tracer const& rhs);
    ~Tracer (void);
    private:
    int m_data;
    };

跟踪器仅供测试

- 跟踪器的定义有且仅有满足模板测试的功能
  - Tracer::Tracer (void) {
    cout << "缺省构造：" << this << endl; }
  - Tracer::Tracer (int data) : m_data (data) {
    cout << "有参构造：" << this << endl; }
  - Tracer::Tracer (Tracer const& that) : m_data (that.m_data) {
    cout << "拷贝构造：" << &that << "->" << this << endl; }
  - Tracer& Tracer::operator= (Tracer const& rhs) {
    cout << "拷贝赋值：" << &rhs << "->" << this << endl;
    m_data = rhs.m_data;
    return *this; }
  - Tracer::~Tracer (void) {
    cout << "析构函数：" << this << endl; }

知识讲解

跟踪器的覆盖范围

- 对于被跟踪模板的每个操作，跟踪器都应该有一个相对应跟踪动作
  - vector<Tracer> v1 (3);
    vector<Tracer> v2 (3, 10);
    v2.push_back (20);
    v2.pop_back ();
    v2.erase (v2.begin ());
    v2.resize (3);
    v2.resize (4);
    v2.reserve (10);
    v2.clear ();
    v2.push_back (30);
    v2 = v1;

知识讲解

跟踪器

【参见：TTS COOKBOOK 】

课堂练习

- 跟踪器

静态多态

静态多态

- 将同一个操作作用于不同的对象，却可以产生不同的结果，谓之多态。基于虚函数的多态通常被称为动态多态，而基于模板的多态则数据静态多态。相较于动态多态，静态多态具有更好的时间性能，而且代码更加简洁
  - class Rect { public: void draw (void) const { ... } };
  - class Circle { public: void draw (void) const { ... } };
  - template<typename Shape>
    void drawAny (Shape const& shape) {
        shape.draw (); }
  - drawAny (Rect (...));
    drawAny (Circle (...));

知识讲解

Technology Tarena 达内科技

静态多态

【参见：TTS COOKBOOK 】

课堂练习

- 静态多态

Technology Tarena 达内科技

总结和答疑