Technology Tarena 达内科技

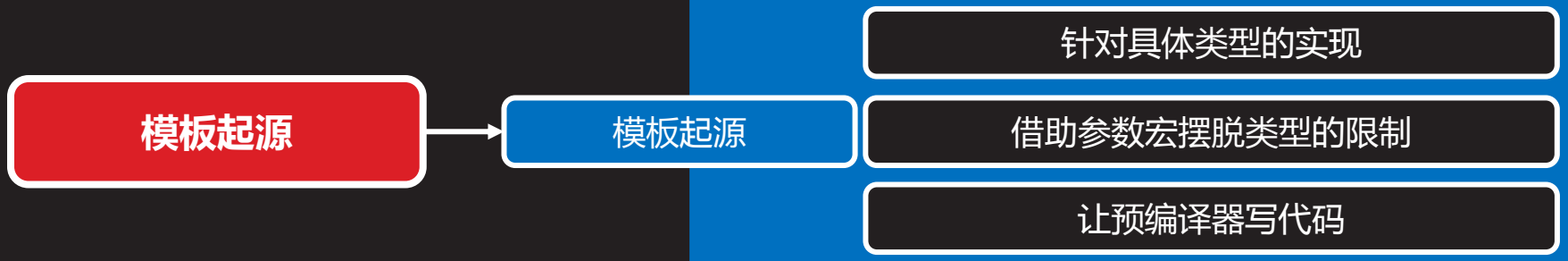
模板和STL

TEMPLATE & STL DAY01

内容

上午	09:00 ~ 09:30	模板起源
	09:30 ~ 10:20	
	10:30 ~ 11:20	函数模板
	11:30 ~ 12:20	
下午	14:00 ~ 14:50	类模板
	15:00 ~ 15:50	
	16:00 ~ 16:50	非类型模板参数
	17:00 ~ 17:30	总结和答疑

模板起源



Technology Tarena 达内科技

模板起源

针对具体类型的实现

- C/C++语言的静态类型系统，在满足效率与安全性要求的同时，很大程度上也成为阻碍程序员编写通用代码的桎梏。它迫使人们不得不为每一种数据类型编写完全或几乎完全相同的实现，虽然它们在抽象层面上是一致的
  - int max_int (int x, int y) {
    return x > y ? x : y;
    }
  - double max_double (double x, double y) {
    return x > y ? x : y;
    }
  - string max_string (string x, string y) {
    return x > y ? x : y;
    }

知识讲解

借助参数宏摆脱类型的限制

- 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(又名宏函数)可以在某种程度上使程序的编写者摆脱那些源于类型的约束和限制，但同时也因此丧失了类型的安全性
  - #define max(x, y) ((x) > (y) ? (x) : (y))
  - cout << max (123, 456) << endl;
  - cout << max (1.23, 4.56) << endl;
  - string a = "hello", b = "world";
    cout << max (a, b) << endl;
  - cout << max ("hello", "world") << endl; // 错误

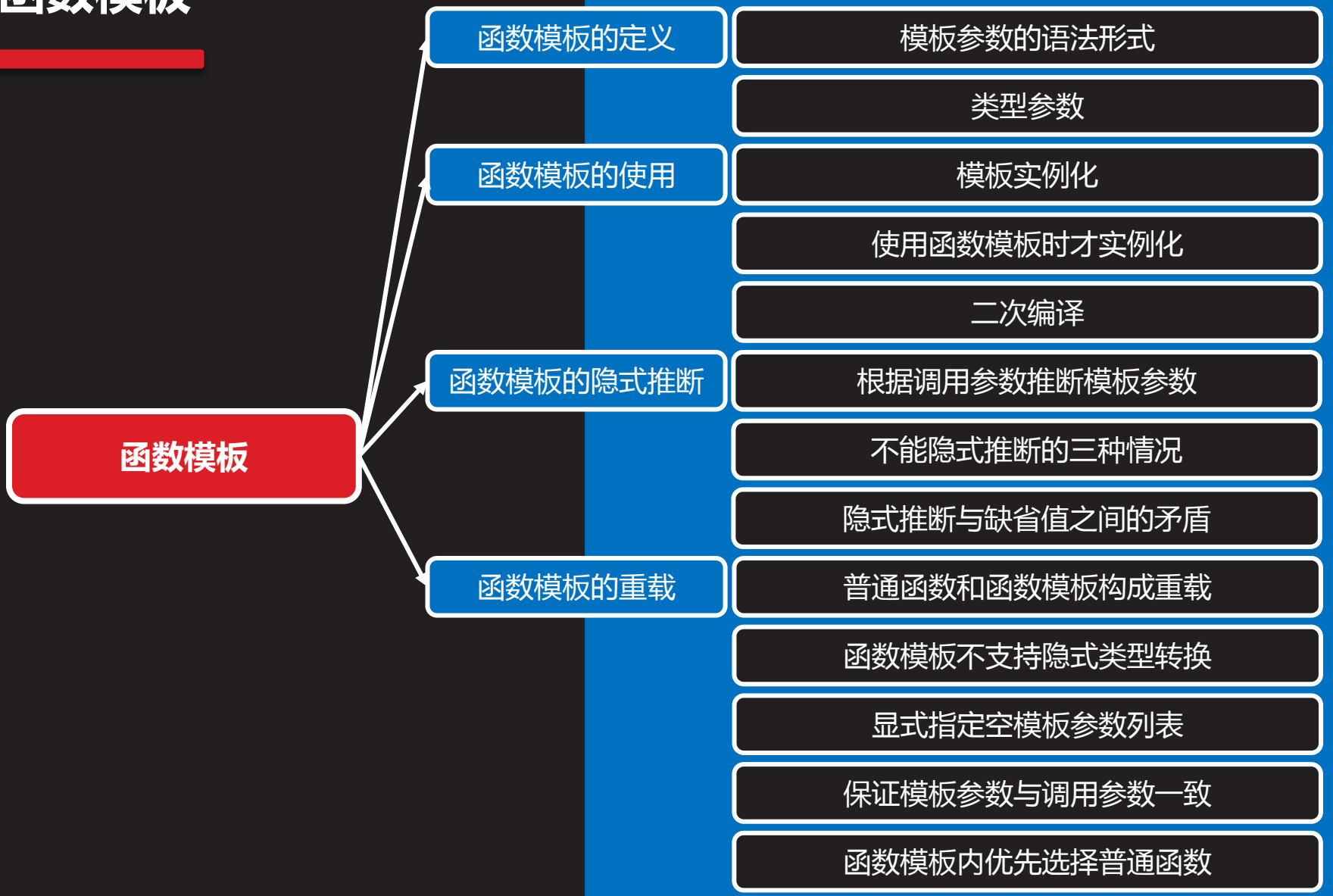
知识讲解

让预编译器写代码

- 利用宏定义构建通用代码的框架，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性完美地结合起来
  - #define MAX(T) T max_##T (T x, T y) { \
    return x > y ? x : y; }
  - MAX (int)
    MAX (double)
    MAX (string)
  - #define max(T) max_##T
  - cout << max(int) (123, 456) << endl;
    cout << max(double) (1.23, 4.56) << endl;
    cout << max(string) ("hello", "world") << endl;

知识讲解

函数模板


函数模板
函数模板的定义
模板参数的语法形式
类型参数
函数模板的使用
模板实例化
使用函数模板时才实例化
二次编译
函数模板的隐式推断
根据调用参数推断模板参数
不能隐式推断的三种情况
隐式推断与缺省值之间的矛盾
函数模板的重载
普通函数和函数模板构成重载
函数模板不支持隐式类型转换
显式指定空模板参数列表
保证模板参数与调用参数一致
函数模板内优先选择普通函数

Technology Tarena 达内科技

函数模板的定义

模板参数的语法形式

- 模板参数必须用如下形式的语法来声明
  - template <typename 类型参数1,
    typename 类型参数2, ...>
- 例如
  - template <typename A, typename b, typename _C>
    A function (b arg) {
    _C var;
    ...
    }

知识讲解

类型参数

- 可以使用任何标识符作为类型参数的名称，但使用“T”已经成为了一种惯例。类型参数“T”表示的是，调用者调用这个函数时所指定的任意类型
- 可以使用任何类型(基本类型、类类型等)实例化模板的类型参数，前提是所使用的类型必须能够满足该模板所需要的操作。比如用一个不支持“<”运算符的类型，实例化max模板的类型参数，将引发编译错误
  - template<typename T>
    T const& max (T const& x, T const& y) {
        return x < y ? y : x;
    }

知识讲解

Technology Tarena 达内科技

函数模板的使用

模板实例化

- 通常而言，并不是把模板编译成一个可以处理任何类型的单一实体，而是对于实例化模板参数的每种类型，都从模板产生出一个不同的实体。这种用具体类型代替模板参数的过程叫做实例化。它产生了一个模板的实例
  - template<typename T> T const& max (
    T const& x, T const& y) { return x < y ? y : x; }
  - int const& max (int const& x, int const& y) {
    return x < y ? y : x; }
  - double const& max (double const& x, double const& y) {
    return x < y ? y : x; }
  - string const& max (string const& x, string const& y) {
    return x < y ? y : x; }

使用函数模板时才实例化

- 只要使用函数模板，编译器就会自动引发这样一个实例化过程，因此程序员并不需要额外地请求对模板实例化
  - 函数模板名<类型实参1, 类型实参2, ...> (调用实参表);
- 例如
  - ::max<int> (123, 456);
  - ::max<double> (1.23, 4.56);
  - ::max<string> ("hello", "world");
- 函数模板的类型实参表放在尖括号中，调用实参表放在圆括号中，且类型实参表必须位于函数模板名和调用实参表之间
- 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

知识讲解

二次编译

- 每个函数模板事实上都被编译了两次
  - 一次是在实例化之前，先检查模板代码本身，查看语法是否正确
  - 另一次是在实例化期间，结合所使用的类型参数，再次检查模板代码，查看是否所有的调用都有效
  - 但是请注意，只有在第二次编译时，才会真正产生二进制形式的机器指令
  - 作为第一次编译的结果，仅仅是在编译器内部形成的一个用于描述该函数模板的数据结构，即所谓模板的内部表示

知识讲解

定义、使用函数模板

【参见：TTS COOKBOOK 】

课堂练习

- 定义、使用函数模板

Technology Tarena 达内科技

函数模板的隐式推断

根据调用参数推断模板参数

- 如果函数模板调用参数的类型相关于该模板的模板参数，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
  - template<typename T>
    void foo (T const& x, T const& y) {
    cout << "T = " << typeid (T).name () << endl; }
  - foo (123, 456); // T = i
  - double x = 1.23, y = 4.56;
    foo (x, y); // T = d
  - foo ("hello", "world"); // T = A6_c
    foo ("hello", "tarena"); // 错误

知识讲解

不能隐式推断的三种情况

- 以下三种情况，不能隐式推断，必须显式指定模板参数
  - 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);
  - 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
  - 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);

知识讲解

隐式推断与缺省值之间的矛盾

- 函数模板参数的隐式推断与缺省值之间存在矛盾，因此函数模板的模板参数不能带有缺省值
  - template<typename T = int>
    T const& max (T const& x, T const& y) {
        return x < y ? y : x;
    }
  - cout << ::max (1.23, 4.56) << endl; // 错误
- C++2011标准允许函数模板带有缺省模板参数，但依然会优先选择隐式推断的结果
  - g++ ... -std=c++0x // < GCC 4.8
    g++ ... -std=c++11 // ≥ GCC 4.8
  - cout << ::max (1.23, 4.56) << endl; // 4.56

隐式推断模板参数

【参见：TTS COOKBOOK 】

课堂练习

- 隐式推断模板参数

Technology Tarena 达内科技

函数模板的重载

普通函数和函数模板构成重载

- 普通函数和可实例化为该函数的函数模板构成重载关系。在其它条件都相同的情况下，编译器优先选择普通函数，除非函数模板能够产生具有更好匹配性的函数实例
  - char const* const& max (char const* const& x,
    char const* const& y) { ... } // ①
    template<typename T>
    T const& max (T const& x, T const& y) { ... } // ②
  - char const* x = "ABC";
    char const* y = "AB";
    cout << ::max (x, y) << endl; // ①
    cout << ::max (100, 200) << endl; // ②

知识讲解

函数模板不支持隐式类型转换

- 函数模板的隐式实例化不支持隐式类型转换，但普通函数支持。因此在参数传递过程中如需隐式类型转换，则编译器将优先选择普通函数
  - char const* const& max (char const* const& x,
    char const* const& y) { ... } // ①
    template<typename T>
    T const& max (T const& x, T const& y) { ... } // ②
  - char const* x = "ABC";
    char* y = "AB";
    cout << ::max (x, y) << endl; // ①

知识讲解

显式指定空模板参数列表

- 可以显式指定一个空的模板参数列表，明确告知编译器使用函数模板，但模板参数却由隐式推断决定。即便是在函数模板中选择，编译器也会尽可能选择类型约束性强的版本，即更特殊的版本
  - char const* const& max (char const* const& x,
    char const* const& y) { ... } // ①
    template<typename T>
    T const& max (T const& x, T const& y) { ... } // ②
    template<typename T>
    T* const& max (T* const& x, T* const& y) { ... } // ③
  - char const* x = "AB";
    char const* y = "ABC";
    cout << ::max<> (x, y) << endl; // ③

保证模板参数与调用参数一致

- 如果为函数模板显式指定了模板参数，那么所选择的重载版本必须能够保证模板参数与调用参数的类型相一致
  - 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 = "ABC";
    char const* y = "AB";
    cout << ::max<char const*> (x, y) << endl; // ①

知识讲解

函数模板内优先选择普通函数

- 即使是在函数模板的实例化函数中，编译器仍然坚持普通函数优先原则，前提是该普通函数在一次编译时可见
  - char const* const& max (char const* const& x,
    char const* const& y) { ... } // ①
    template<typename T>
    T const& max (T const& x, T const& y) { ... } // ②
    template<typename T>
    T const& max (T const& x, T const& y, T const& z) {
    return ::max (::max (x, y), z); } // ③
  - char const* x = "ABC";
    char const* y = "AB";
    char const* z = "A";
    cout << ::max (x, y, z) << endl; // ③->①

知识讲解

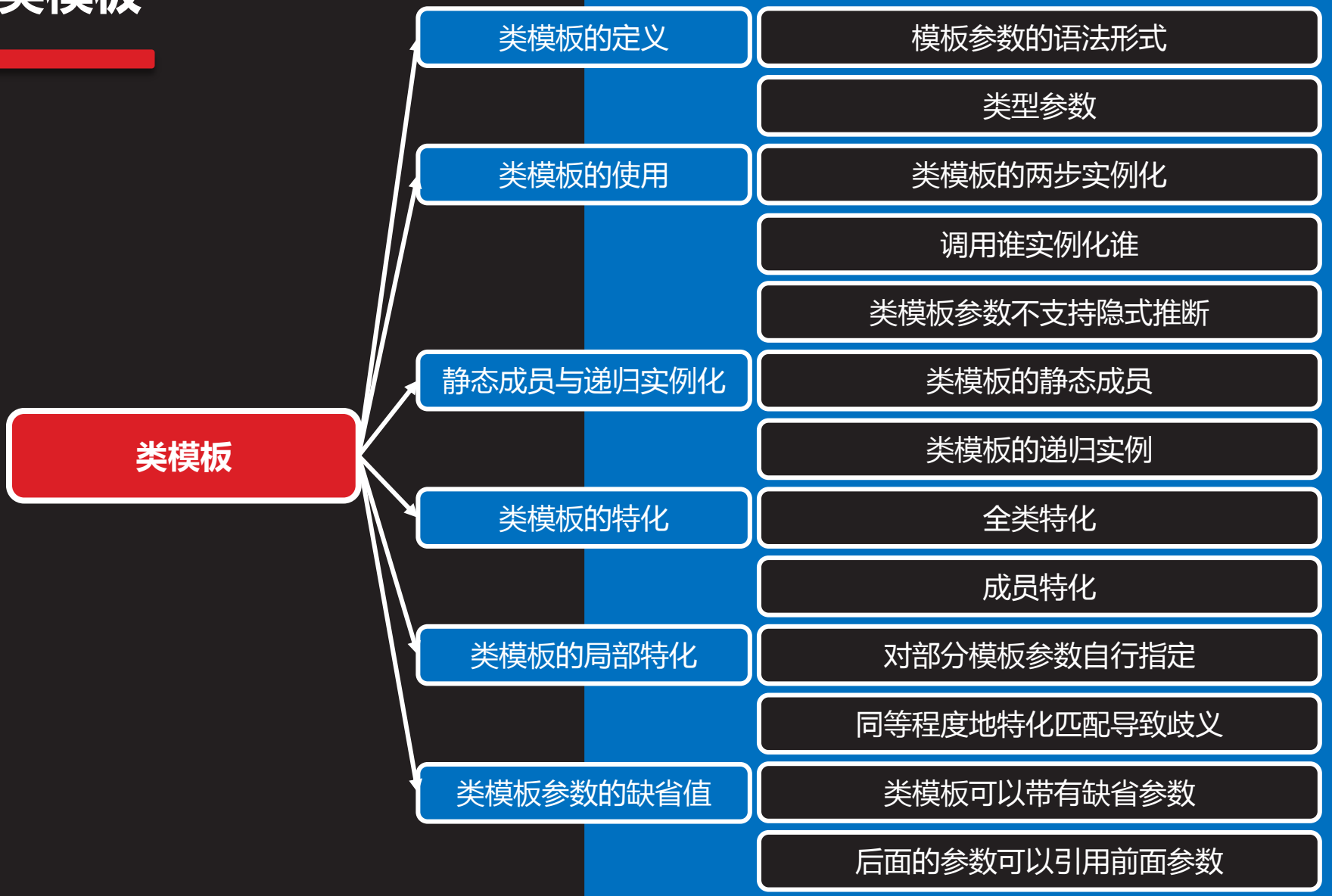
函数模板重载

【参见：TTS COOKBOOK 】

课堂练习

- 函数模板重载

类模板


类模板
类模板的定义
模板参数的语法形式
类型参数
类模板的使用
类模板的两步实例化
调用谁实例化谁
类模板参数不支持隐式推断
静态成员与递归实例化
类模板的静态成员
类模板的递归实例
类模板的特化
全类特化
成员特化
类模板的局部特化
对部分模板参数自行指定
同等程度地特化匹配导致歧义
类模板参数的缺省值
类模板可以带有缺省参数
后面的参数可以引用前面参数

Technology Tarena 达内科技

类模板的定义

模板参数的语法形式

- 模板参数必须用如下形式的语法来声明
  - template <typename 类型参数1,
    typename 类型参数2, ...>
- 例如
  - template <typename A, typename b, typename _C>
    class MyClass {
    public:
    A m_a;
    b foo (_C c);
    };

知识讲解

类型参数

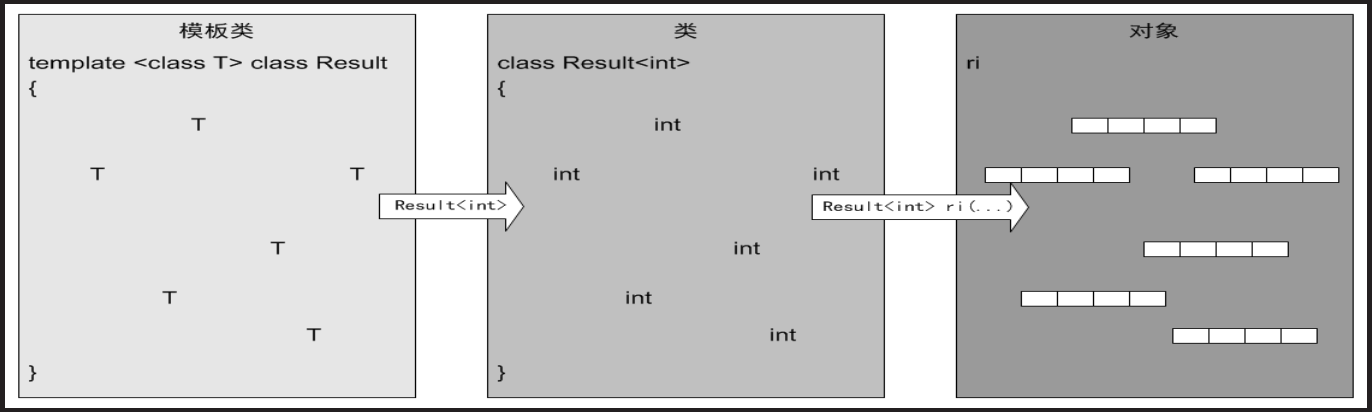
- 在类模板的内部，类型参数可以象其它任何具体类型一样，用于成员变量、成员函数、成员类型(内部类型)，甚至基类的声明
  - template<typename M, typename R, typename A,
    typename V, typename T, typename B>
    class MyClass : public B {
    M m_mem;
    R function (A arg) { ... V var ... }
    typedef T* pointer;
    };

Technology Tarena 达内科技

类模板的使用

类模板的两步实例化

- 从类模板到对象实际上经历了两个实例化过程
  - 编译期：编译器将类模板实例化为类并生成对象创建指令
  - 运行期：处理器执行对象创建指令将类实例化为内存对象



- 类模板本身并不代表一个确定的类型，既不能用于定义对象，也不能用于声明指针或引用。只有通过模板实参将其实例化为具体类以后，可具备类型语义
  - 类模板名<类型实参1, 类型实参2, ...>

知识讲解

调用谁实例化谁

- 类模板中，只有那些被调用的成员函数才会被实例化，即产生实例化代码。某些类型虽然并没有提供类模板所需要的全部功能，但照样可以实例化该类模板，只要不直接或间接调用那些依赖于未提供功能的成员函数即可
  - template<typename T> class Comparator {
    T const& max (void) const {
    return m_x < m_y ? m_y : m_x; }
    T const& min (void) const {
    return m_x > m_y ? m_y : m_x; } };
  - class Integer { bool operator< (Integer const& rhs); }
  - Comparator<Integer> ci (123, 456);
    cout << ci.max () << endl;

知识讲解

类模板参数不支持隐式推断

- 与函数模板不同，类模板的模板参数不支持隐式推断
  - template<typename T>
    class Comparator {
    public:
        Comparator (T const& x, T const& y) :
            m_x (x), m_y (y) {}
        ...
    private:
        int m_x, m_y; };
  - Comparator ci (123, 456); // 错误
  - Comparator<int> ci (123, 456);

知识讲解

定义、使用类模板

【参见：TTS COOKBOOK 】

课堂练习

- 定义、使用类模板

Technology Tarena 达内科技

静态成员与递归实例化

类模板的静态成员

- 类模板的静态成员变量，既不是一个对象一份，也不是一个模板一份，而是在该类模板的每个实例化类中，各有一份独立的拷贝，且为该类的所有实例化对象所共享
  - template<typename T> class A {
    public:
    static void print (void) {
    cout << &m_i << ' ' << &m_t << endl; }
    private:
    static int m_i;
    static T m_t; };
  - template<typename T> int A<T>::m_i;
    template<typename T> T A<T>::m_t;

知识讲解

类模板的静态成员变量

【参见：TTS COOKBOOK 】

课堂练习

- 类模板的静态成员变量

类模板的递归实例化

- 类模板的类型实参可以是任何类型，只要该类型能够提供模板所需要的功能
- 模板自身的实例化类亦可实例化其自身，谓之递归实例化。通过这种方法可以很容易地构建那些在空间上具有递归特性的数据结构
  - template<typename T>
    class Array {
        T m_array[3];
    };
  - Array<int> a1; // 一维数组
    Array<Array<int> > a2; // 二维数组
    Array<Array<Array<int> > > a3; // 三维数组

知识讲解

类模板的递归实例化

【参见：TTS COOKBOOK 】

课堂练习

- 类模板的递归实例化

Technology Tarena 达内科技

类模板的特化

全类特化

- 特化一个类模板可以特化该类模板的所有成员函数，相当于重新写一个针对某种特定类型的具体类
  - template<> class Comparator<char const*> {
    public:
    Comparator (char const* const& x,
    char const* const& y) : m_x (x), m_y (y) {}
    char const* const& max (void) const {
    return strcmp (m_x, m_y) < 0 ? m_y : m_x; }
    private:
    char const* const& m_x, &m_y;
    };
  - Comparator<char const*> cmp ("hello", "world");
    cout << cmp.max () << endl; // world

成员特化

- 类模板除了可以整体进行特化以外，也可以只针对部分成员函数进行特化
  - template<typename T> class Comparator {
    public:
    Comparator (T const& x, T const& y) :
    m_x (x), m_y (y) {}
    T const& max (void) const {
    return m_x < m_y ? m_y : m_x; }
    private:
    T const& m_x, &m_y; };
  - template<> char const* const& Comparator<
    char const*>::max (void) const {
    return strcmp (m_x, m_y) < 0 ? m_y : m_x; }

类模板的特化

【参见：TTS COOKBOOK 】

课堂练习

- 类模板的特化

Technology Tarena 达内科技

类模板的局部特化

对部分模板参数自行指定

- 类模板可以被局部特化，即一方面为类模板指定特定的实现，另一方面又允许用户对部分模板参数自行指定
  - // 基本版本

    template<typename A, typename B> class X {};
    - // 针对第二个模板参数取short的局部特化
      template<typename A> class X<A, short> {};
    - // 针对两个模板参数取相同类型的局部特化
      template<typename A> class X<A, A> {};
    - // 针对两个模板参数取某种类型指针的局部特化
      template<typename A, typename B> class X<A*, B*> {};
    - // 针对两个模板参数取某种类型引用的局部特化
      template<typename A, typename B> class X<A&, B&> {};
    - // 针对两个模板参数取某种类型数组的局部特化
      template<typename A, typename B> class X<A[], B[]> {};

知识讲解

同等程度地特化匹配导致歧义

- 如果多个局部特化同等程度地匹配某个声明，那么该声明将因二义性而导致歧义错误
  - X<int*, int*> x; // 错误
    - template<typename A> class X<A, A> {};
    - template<typename A, typename B> class X<A*, B*> {};
- 除非有更好的匹配
  - X<int*, int*> x;
    - template<typename A> class X<A*, A*> {};

知识讲解

Technology Tarena 达内科技

类模板的局部特化

【参见：TTS COOKBOOK 】

课堂练习

- 类模板的局部特化

Technology Tarena 达内科技

类模板参数的缺省值

类模板可以带有缺省参数

知识讲解

- 类模板的模板参数可以带有缺省值，即缺省模板实参
  - 实例化类模板时，如果提供了模板实参则用所提供的模板实参实例化相应的模板形参，如果没有提供模板实参则相应的模板形参取缺省值
- 例如
  - template<typename A = int, typename B = double, typename C = string> class X { ... };
  - X<char, short> x1 (...); // X<char, short, string> x1 (...)
    X<char> x2 (...); // X<char, double, string> x2 (...)
    X<> x3 (...); // X<int, double, string> x3 (...)
- 如果类模板的某个模板参数带有缺省值，那么它后面的所有模板参数必须都带有缺省值

后面的参数可以引用前面参数

- 类模板后面参数的缺省值可以引用前面参数的值
  - template<typename A, typename B = A*>
    class X { ... };
  - X<int> x; // 等价于X<int, int*> x
- 例如
  - template<typename T, typename C = vector<T> >
    class Stack {
    public:
    void push (T const& elem) {
    m_container.push_back (elem); }
    private:
    C m_container; };

带有缺省参数的类模板

【参见：TTS COOKBOOK 】

课堂练习

- 带有缺省参数的类模板

非类型模板参数

Technology Tarena 达内科技

数值形式的模板参数

普通数值作为模板参数

- 模板的参数并不局限于类型参数，普通数值也可以作为模板的参数，前面不要写typename，而要写具体类型
  - template<typename T, size_t C> class Array {
    public:
    T& operator[] (size_t i) {
    return m_array[i]; }
    T const& operator[] (size_t i) const {
    return const_cast<Array<T>&> (*this)[i]; }
    size_t capacity (void) const { return C; }
    private:
    T m_array[C]; };
  - Array<int, 10> array;
    for (int i = 0; i < array.capacity (); ++i) array[i] = i + 1;

知识讲解

非类型模板参数只能是常量

- 非类型模板参数只能是常量、常量表达式，以及带有常属性(const)的变量，但不能同时具有挥发性(volatile)
  - Array<int, 10> array;
  - Array<int, 3+7> array;
  - int const x = 3, y = 7;
    Array<int, x+y> array;
  - int const volatile x = 3, y = 7;
    Array<int, x+y> array; // 错误

知识讲解

类模板和函数模板的非类型参数

【参见：TTS COOKBOOK 】

课堂练习

- 类模板和函数模板的非类型参数

Technology Tarena 达内科技

总结和答疑