Modern C++ Camp

现代C++系统研发骨干特训营

李建忠 Boolan

Boolan
高端IT咨询与教育平台

现代C++系统研发骨干特训营

模块四、模板与泛型编程

模板与泛型编程

- 面向对象（编程范式）
  - 封装
  - 继承
  - 多态
  - 运行时抽象
- 泛型编程（编程范式）
  - 类模板
  - 函数模板
  - 概念
  - 编译时抽象

泛型编程应该成为正常编程活动的一部分

Bjarne Stroustrup

C++模板简介

- C++模板是一种编译时机制，在编译时生成具体的代码，使用实参将模板定义实例化为具体的类型或函数。C++支持两种模板：
  - 类模板
  - 函数模板

- 模板实例化时编译器会对实参类型进行检查，确保实参符合对模板参数的操作要求。 C++模板参数支持两种：
  - 类型参数，可隐式约束、也可显式约束
  - 值参数，编译时常量、或constexpr函数。不同值参数是不同类型（不允许是值浮点数、或者类对象：编译器不能确定值）。
  - 可为模板参数提供默认值

- 对模板参数进行显式约束，即C++ 20的概念（Concept）

模板类成员

- 普通成员：使用与主模板相同类型模板参数
  - 数据成员（变量、常量）
  - 成员函数
  - 静态成员（数据或函数）
  - 成员类型别名
  - 成员类型
- 成员模板（使用与主模板不同的类型参数）
  - 成员模板不能定义虚函数（模板实例化会导致链接器不断为虚表增加虚函数增项）
- 所有普通类的成员规则同样适用于模板类成员

模板实例化机制

- 数据成员——只要类型被使用，编译器会根据其数据成员、生成对应类型结构。
- 函数成员——选择性实例化
  - 非虚函数，如果实际调用到，则会生成代码；如果没有调用到，则不生成。
  - 虚函数，无论是否调用，总会生成代码。因为在运行时“有可能”调用到。
- 隐藏编译错误
  - 如果某些模板方法没有被调用，即使包含编译错误，也会被忽略。
- 强制实例化模板
  - 使用template class Array<int>; 来强制要求编译所有模板类函数成员，排除所有编译错误，无论是否调用到。

类型别名与模板别名

- 为模板类使用指定别名
  - 类型别名（alias type）：指定所有模板参数，得到完整类型
  - 模板别名（alias template）：指定部分参数，得到模板类型
  - 成员类型别名：类模板中通过定义类型别名，来定义“关联类型”
  - 别名和原始模板完全等价，包括使用模板特化时（但不支持特化别名）

- 优先使用using 而不是typedef
  - 两者都可以声明类型别名、成员类型别名
  - using 可以定义别名模板，而typedef不可以
  - using 可以免掉类型内typedef要求的typename前缀，和::type 后缀

模板参数类型自动推导

- C++模板编译时支持对类型参数的自动推导：
  - 普通函数参数
  - 类成员函数参数
  - 构造函数参数（C++ 17 支持），模板所有类型参数都有值

- 模板类型推导时：
  - 引用性被忽略：引用类型实参被当作非引用类型处理。
  - 转发引用时，左值参数按左值、右值参数按右值。
  - 按值传递时，实参中const/volatile修饰会被去掉。

模板特化

- 模板类型的特化指的是允许模板类型定义者为模板根据实参的不同，而定义不同实现的能力。
- 特化版本可以和一般版本拥有不同的外部接口和实现。
- 模板偏特化：也可以对部分模板参数进行特化定义，称为偏特化。
- 模板特化支持模板类、模板函数。

奇异递归模板模式

Curiously Recurring Template Pattern，简称CRTP，通过将基类模板参数设置为子类，从而实现静态多态（静态接口），或者扩展接口（委托实现）。

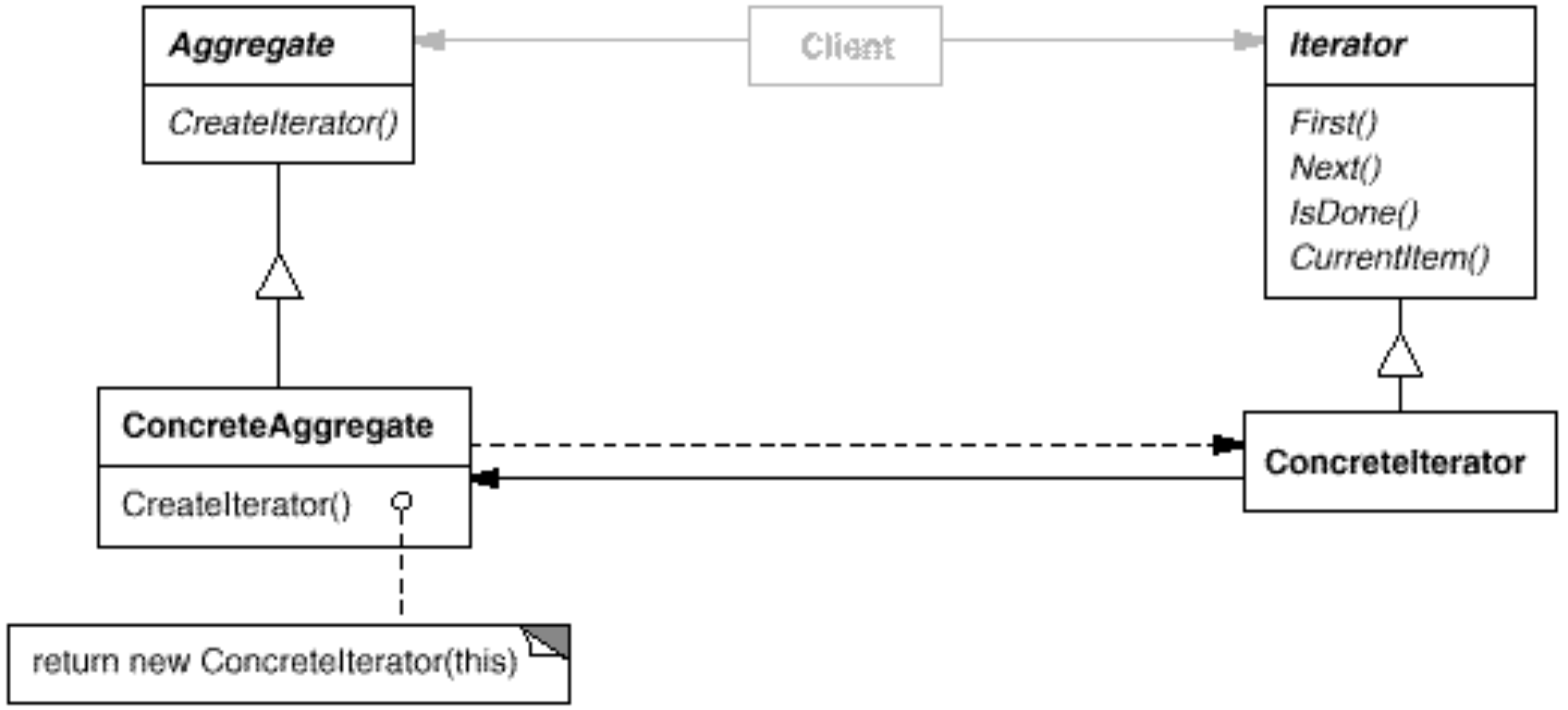
CRTP实现要点解析：

- class Sub : public Base<Sub> 通过模板参数，将子类类型在编译时注入基类，从而实现在基类中提前获取子类类型信息。
- static_cast<T*>(this) 将基类指针转型为模板子类T的指针
- Base类型为不完整类型，不能使用Sub参与内存布局，但可以在函数内使用（发生调用，模板编译时辨析即可）
- 删除对象，也要使用编译时多态进行删除，避免直接delete

泛型编程 = 面向概念编程

- 通过定义迭代器等一系列概念，作为容器对外“遍历”的泛型接口（概念），所有算法依赖概念、而非容器编程。从而实现泛型编程领域的“面向接口编程” == “面向概念编程”。

迭代器设计模式



STL中的迭代器概念

- 随机访问迭代器 random_access_iterator
- （*）、（->）、（[]）++、--、+、-、+=、-=、==、!=、<、>、<=、>=
- 双向迭代器bidirectional_iterator
  - （*）、（->）、++、--、==、!=
- 单向迭代器forward_iterator
  - （*）、（->）、++、==、!=
- 输入迭代器input_iterator
  - （*）、（->）、++、==、!=
- 输出迭代器output_iterator
  - （*）++

STL容器支持的迭代器

- vector 随机访问
- array 随机访问
- deque 随机访问
- list 双向迭代
- forward_list 单项迭代
- set/multiset 双向迭代
- map/multimap 双向迭代
- unordered_set/unordered_map 单向迭代器
- unordered_multiset/unordered_multimap 单向迭代器
- stack 不支持选代器
- queue 不支持迭代器

函数对象

- 函数对象（function object），又叫仿函数、函子（functor）、通过重载类的operator()调用操作符，实现将类对象当作函数调用的能力。
- 作为类对象，函数对象可以定义实例变量，并通过构造器参数来初始化，从而使得函数对象可以携带状态数据。
- 函数对象通常可以inline，其性能比函数指针要高。函数指针只能运行时辨析地址，进行间接调用、也无法内联，性能较差。
- 函数对象可以采用类模板的方式模板化，从而使得函数对象可以参与泛型编程。STL内大量地使用了函数对象作为算法策略（policy）

lambda表达式

- lambda表达式，等价于匿名函数对象，又称闭包（ closure ），更便捷、表达更直接。表达式要素：
  - 捕获列表（可空）
  - 参数列表（可选）
  - mutable修饰符，表达传值或传引用
  - noexcept（可选）
  - 返回值类型声明 ->（可选）
  - 表达式体 { ... }
- lambda表达式可接受参数、可返回值、可模板化、也可通过传值或传引用从闭包范围内访问变量。
- 编译器将lambda表达式编译为具名函数对象。

捕获列表：有状态 VS. 无状态

- lambda表达式，从闭包作用域捕获变量而获得状态，分传值、和传引用两种。捕获变量等价于 函数对象中的实例数据成员。
  - [=] 值捕获所有变量
  - [&] 引用捕获所有变量
  - [&x] 引用捕获x变量
  - [x] 值捕获 x
  - [=, &x] 默认通过值捕获，x变量通过引用捕获
  - [&, x] 默认通过引用捕获，x变量通过值捕获
  - [this] 捕获当前对象，可访问其所有公有成员
  - [=, x], [&, &x] 错误，重复指定
  - 注意，即便默认要求值捕获，全局变量总是使用引用捕获
  - 使用初始化捕获表达式表达move捕获（C++14）

函数适配器

- 函数适配器：接受函数参数，返回可调用该函数的函数对象。本质是函数对象。
- bind() 使用“额外实参”来绑定任意函数
- mem_fn() 绑定成员函数，适配为非成员函数（额外实参，仍需要bind）

- 下列适配器已经废弃，不再推荐使用
- bind1st, bind2nd
- ptr_fun, mem_fun, mem_fun_ref
- not1, not2

值语义 VS. 引用语义

- 模板函数、STL 容器默认提供值语义，插入/返回元素都使用拷贝构造（或移动，符合条件的话）。
- 如果提供引用语义，可使用如下两种方式： reference_wrapper 与 shared_ptr 。
- reference_wrapper内部封装指针，可引用栈对象、也可引用堆对象。可隐式转换为原生对象。但生命周期管理容易出问题。可借用 std::ref 与cref 将函数模板的 传值改为传引用。
- shared_ptr 共享对象，对象创建堆上，生命周期管理遵循RAII。

可调用构造与std::function

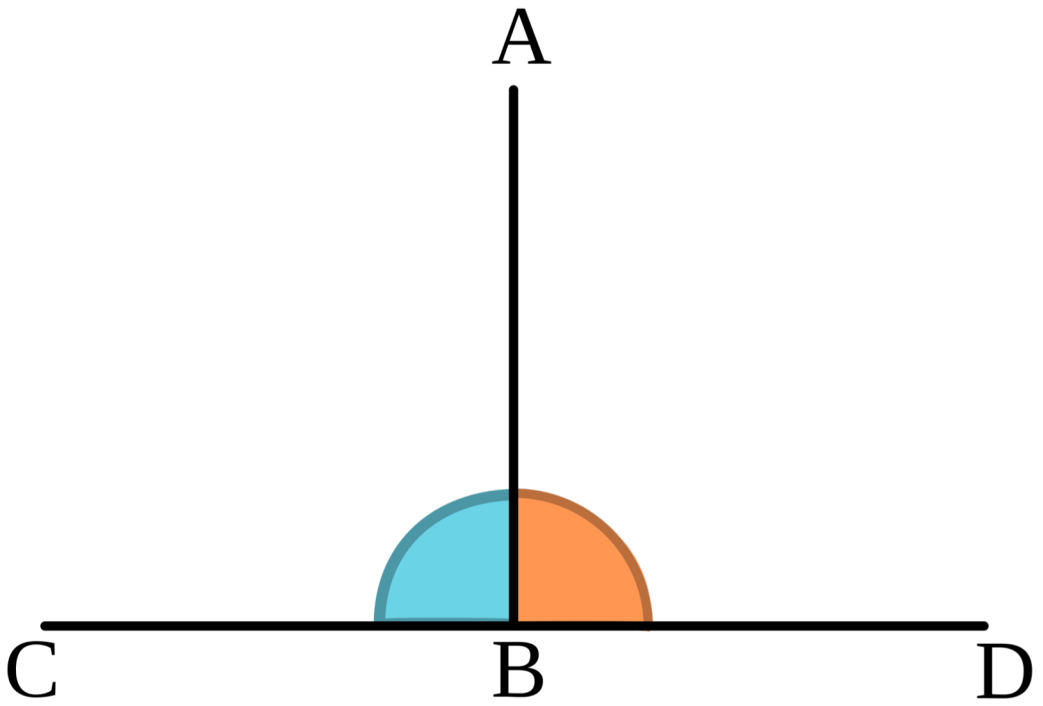
- std:function 可统一绑定所有符合调用约定的可调用构造：
  - 函数对象
  - 函数指针
  - 成员函数指针
  - Lambda表达式
  - bind、mem_fn函数适配器
- std::function 使用了 小对象栈、大对象堆的存储优化。
- 其默认大小为48个byte，存储大小一般比Lambda对象大。
- 其支持拷贝、赋值、移动等操作；拷贝是深拷贝，连同绑定对象状态一起拷贝。
- 内部存储函数指针，调用有运行时成本，不需要间接存储可调用对象，尽量避免使用。

函数对象、Lambda表达式最佳实践

- 函数对象 、Lambda表达式优于函数指针。对性能关键的应用，推荐使用自定义函数对象。
- 相对bind, mem_fn等函数适配器，优选Lambda表达式
- 对于引用语义，优选shared_ptr、而不是 reference_wrapper
- 显式列出捕获列表参数，避免默认捕获模式
- 如果捕获参数需要 move语义，使用初始化捕获表达式（C++14）
- 如果不需要多态绑定可调用构造，尽量避免使用std::function，其有较高存储成本、拷贝成本、调用成本。

软件设计的正交性

- 正交（ orthogonality ）意味着独立性，如果A的改变不影响B，那么A和B在设计上是正交的。
- 设计正交，即意味着设计的解耦（Decouple），通过解耦，消除不相关组件之间的影响。
- 设计不正交，即意味着耦合度高，牵一发而动全身，难以变更和修复。



策略设计（Policy）

- 将一个类或算法的设计分解为各种policy，找到正交分解点。将设计期的各种决定和约束条件留给policy决定。
- Policy 为泛型函数和类型提供可配置行为，基于行为提供正交设计的灵活性。通常为可调用构造，配置为模板参数。
  - 不需要有默认值，通常需要显式指定
  - 和其他模板参数通常成正交设计关系
  - 通常包含成员函数，也可以是类的静态成员模板
  - 可以聚合在平凡类内，或者模板类内

类型萃取（Trait）

- 通过附加属性，基于类型特征提供正交设计的灵活性。

- 聚合了相关各种相关类型和常量，一般不包含成员函数
- 可以是固定trait（不用模板参数化），也可是模板，可以构成Traits Template
- Trait参数通常依赖其他模板主参数
- 作为模板参数，通常有默认值
- 注意Traits与Policy的区别：Traits基于类型特征，Policy基于行为。

Traits与类型函数

- 类型函数：通常的函数称为 值函数。类型函数：接受类型作为参数，返回类型或者常量。类模板也可看作一种type function

- 元素类型Trait
- 返回值类型Trait
- 类型标签分发（Tag Dispatch）
- 判断式（Predicate Trait ）
- 类型转换函数Trait

变参模板与折叠表达式

- 变参模板（Variadic Template）：将模板参数定义为可接受任意数目、任意类型的实参；通过递归效应来实现编译时展开。
  - 变参模板完美转发
  - 变参表达式
  - 变参索引
  - 变参类模板： Tuple、 Variant;
  - 变参推导
  - 变参基类

- 折叠表达式：遍历参数包中所有元素，可简化变参模板的实现。

常用类型

- pair与结构化绑定
- tuple：一组异构元素序列，不止两个。
- variant，泛型版的联合数据结构，类型更安全
- optional：指定类型有值或无值（空指针）
- any：表达不限数量的可选类型中的一个