

案例与设计模式 I

(OOP)

刘知远

`liuzy@tsinghua.edu.cn`

`http://nlp.csai.tsinghua.edu.cn/~lzy/`

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 并发编程
- `thread`与主从模式
- `mutex`与互斥锁模式
- `async`、`future`、`promise`与异步
- 设计模式简介

设计模式

■ 行为型模式 (Behavioral Patterns)

- 关注对象行为功能上的抽象，从而提升对象在行为功能上的可拓展性，能以最少的代码变动完成功能的增减

■ 结构型模式 (Structural Patterns)

- 关注对象之间结构关系上的抽象，从而提升对象结构的可维护性、代码的健壮性，能在结构层面上尽可能的解耦合

■ 创建型模式 (Creational Patterns)

- 将对象的创建与使用进行划分，从而规避复杂对象创建带来的资源消耗，能以简短的代码完成对象的高效创建

本讲内容提要

- 14.1 案例介绍：语言集成查询LINQ解析器
- 14.2 迭代器（Iterator）模式
- 14.3 模板方法（Template Method）模式
- 14.4 策略（Strategy）模式

案例：语言集成查询LINQ

■ LINQ (Language Integrated Query) 即语言集成查询。

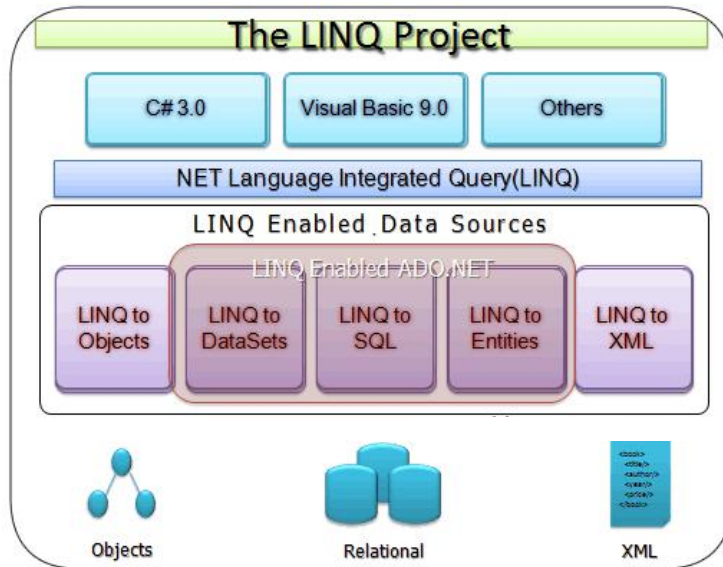
- LINQ是一组语言特性和API，让用户可以使用统一的接口对数据进行操作。
- LINQ常用于保存和检索来自不同数据源的数据（譬如来自XML、对象集合或者数据库），从而消除了编程语言和数据来源之间的不匹配。

案例：语言集成查询LINQ

- LINQ技术由微软提供在 .NET Framework 的编程框架中，目前支持C#以及Visual Basic语言。
- 根据数据源的不同，包括：
 - LINQ to Objects (对象的查询)
 - LINQ to XML (XML的查询)
 - LINQ to ADO.NET (数据库的查询)
 - LINQ to DataSets
 - LINQ to SQL
 - LINQ to entities

案例：语言集成查询LINQ

■ LINQ整体框架



- 本节课中，我们将一份**LINQ to Objects**迁移到**C++**上的简化实现作为案例，探讨背后开发过程中蕴含的设计模式

相关知识： Lambda函数

■ C++11中提供了对匿名函数的支持,称为 Lambda函数, 具体形式如下:

■ `[capture] (parameters) mutable -> return-type {statement}`

■ capture: 捕捉列表, 本节的例子中不使用捕捉列表。

■ (parameters): 参数列表。

■ mutable: 在默认的情况下, lambda函数总是返回一个const, 注明“mutable”关键字之后可以取消其常量性质。

■ ->return-type: 函数的返回值类型。可省略。

■ {statement}: 函数体。

```
[](int x) { return x % 2 == 0; }    // 判断x是否是偶数  
[](int x) { return x * x; }); // 返回x的平方
```


案例：语言集成查询LINQ

- C++ LINQ to Objects 实现功能

- from

- 接收不同的容器，返回一个自定义对象，该对象封装了该容器的迭代器

- select

- 输入Lambda函数，通过封装的迭代器对内部元素进行操作

- where

- 输入Lambda函数，通过封装的迭代器对内部元素进行筛选

案例：语言集成查询LINQ

■应用示例（from）

■可接受不同的输入（数组、容器等）

```
vector<int> v = {1, 2, 3, 4, 5, 6};  
  
for (auto x : from(v)) {  
    cout << x << " "; // 1 2 3 4 5 6  
}
```

```
double v[] = {1, 2, 3, 4, 5, 6};  
  
for (auto x : from(v)) {  
    cout << x << " "; // 1 2 3 4 5 6  
}
```

案例：语言集成查询LINQ

■应用示例（from）

■可接受不同的输入（数组、容器等）

```
struct person {  
    string name;  
    int age;  
};  
  
person zs = {"张三", 30};  
person ls = {"李四", 40};  
person ww = {"王五", 50};  
person persons[] = { zs, ls, ww };  
  
for (auto x : from(persons)) {  
    cout << x.age << " "; // 30 40 50  
}
```

案例：语言集成查询LINQ

■应用示例（from、where、select）

■支持语句链式执行

```
int v[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
  
auto q = from(v)  
         .where([](int x) {return x % 2 == 0;}) // 选择偶数  
         .select([](int x) { return x * x; }); // 进行平方操作  
  
for (auto x : q) {  
    cout << x << " "; // 4 16 36 64  
}
```

from的实现 与迭代器模式

案例：语言集成查询LINQ

■ from函数实现

- from函数返回自定义对象`linq_enumerable`，该对象封装了该容器的迭代器。这里只需要保存begin和end两个迭代器，便可以实现对于内部元素的迭代与操作
- 使用模板参数`TContainer`接受不同的输入容器
- `std::begin()`和`std::end()`函数返回容器的begin和end两个迭代器

```
template<typename TContainer>
auto from(const TContainer& c)
    ->linq_enumerable<decltype(std::begin(c))> {

    return
        linq_enumerable<decltype(std::begin(c))>(std::begin(c),
std::end(c));
}
```

案例：语言集成查询LINQ

■ from函数实现

- `linq_enumerable`是一个类模板，其类型由容器`c`的迭代器的类型决定，通过`decltype(std::begin(c))`获得。初始化时接受两个输入参数，即容器的`begin`和`end`两个迭代器
- 使用`auto`和`decltype`关键字决定实际返回类型

```
template<typename TContainer>
auto from(const TContainer& c)
    ->linq_enumerable<decltype(std::begin(c))> {

    return
        linq_enumerable<decltype(std::begin(c))>(std::begin(c),
std::end(c));
}
```

案例：语言集成查询LINQ

■ linq_enumerable 类实现

- 使用模板参数TIterator，构造时存储不同类型的迭代器

```
template<typename TIterator>
class linq_enumerable {
private:
    TIterator _begin;
    TIterator _end;
public:
    linq_enumerable(const TIterator& b, const TIterator& e) :
        _begin(b), _end(e) {}
    TIterator begin() const {
        return _begin;
    }
    TIterator end() const {
        return _end;
    }
};
```


案例：语言集成查询LINQ

■ linq_enumerable类实现

- 因为定义了begin()和end()函数，from函数返回的linq_enumerable对象可以通过for循环遍历

```
vector<int> v = {1, 2, 3, 4, 5, 6};  
  
for (auto x : from(v)) {  
    cout << x << " "; // 1 2 3 4 5 6  
}
```

```
double v[] = {1, 2, 3, 4, 5, 6};  
  
for (auto x : from(v)) {  
    cout << x << " "; // 1 2 3 4 5 6  
}
```

迭代器模式

■ 在from函数的设计中

- 无论from()的输入是哪种容器，代码都可以只根据容器返回的统一**迭代器**进行相关的操作
- 无需考虑容器内的数据类型与具体实现

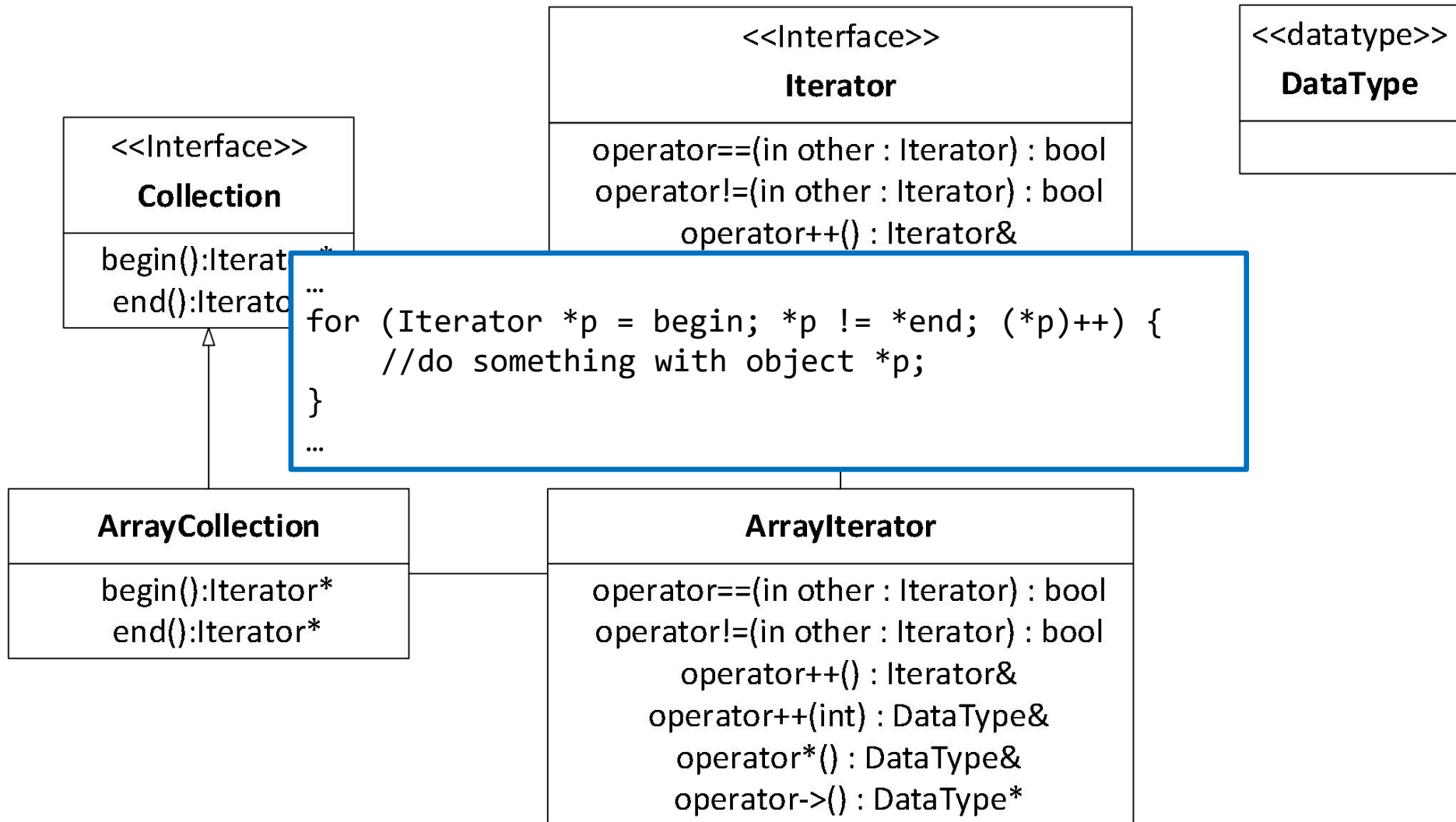
■ 这就是经典的**迭代器模式**

- 将上层数据操作算法与底层数据实现分离
- 接下来，我们将介绍迭代器模式的具体细节

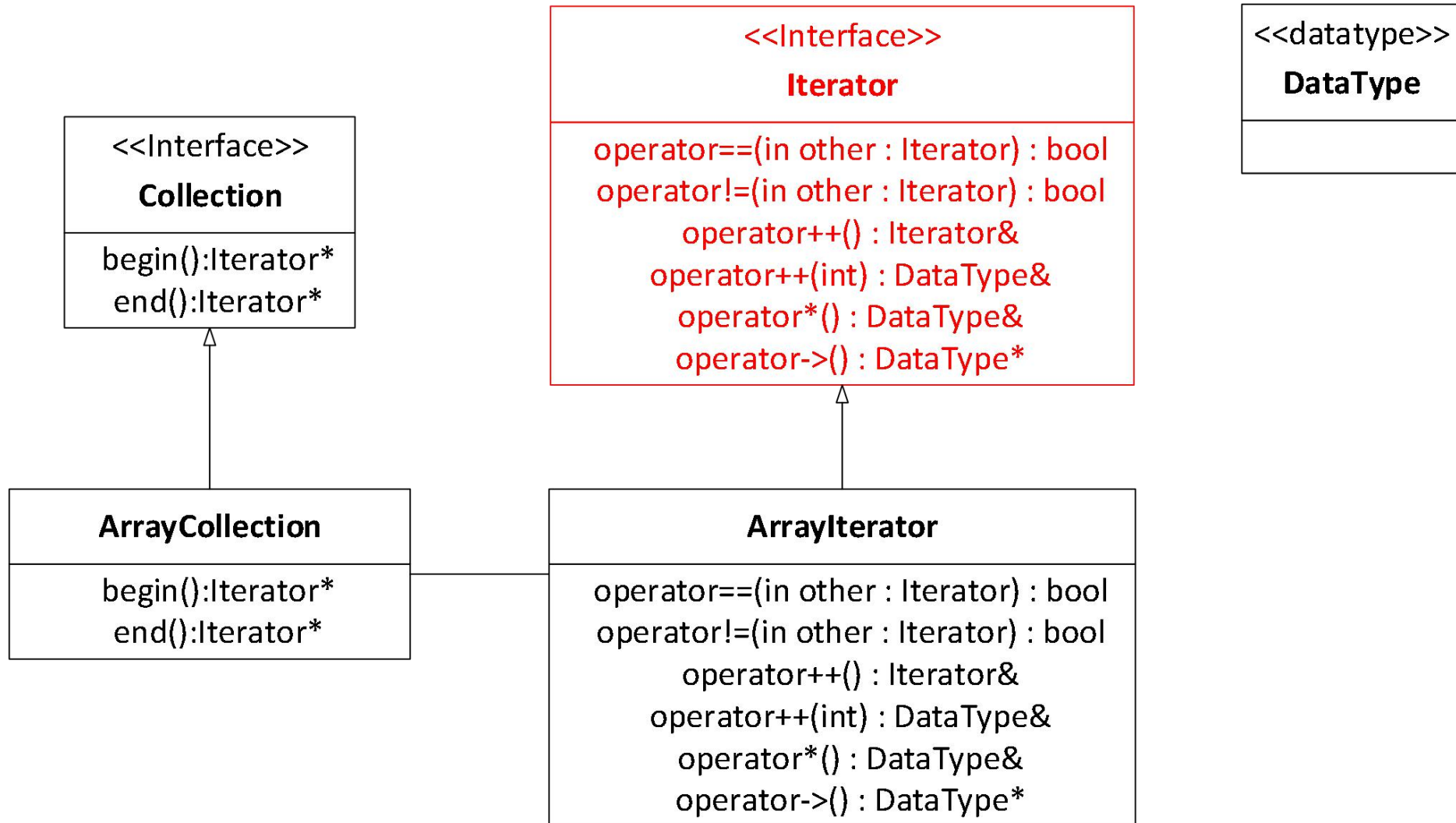
迭代器模式

- 提供一种方法顺序访问一个聚合对象中各个元素
- 又不需暴露该对象的内部表示——与对象的内部数据结构形式无关（数组还是链表）
- 具体实现相当于用模板方法（稍后介绍）构建迭代器和数据存储基类，为每种单独的数据结构都实现其独有的迭代器和存储类
- 但对于上层算法，算法的执行只依赖于抽象的迭代器接口，而无需关注最底层的具体数据结构

迭代器模式



实现Iterator基类



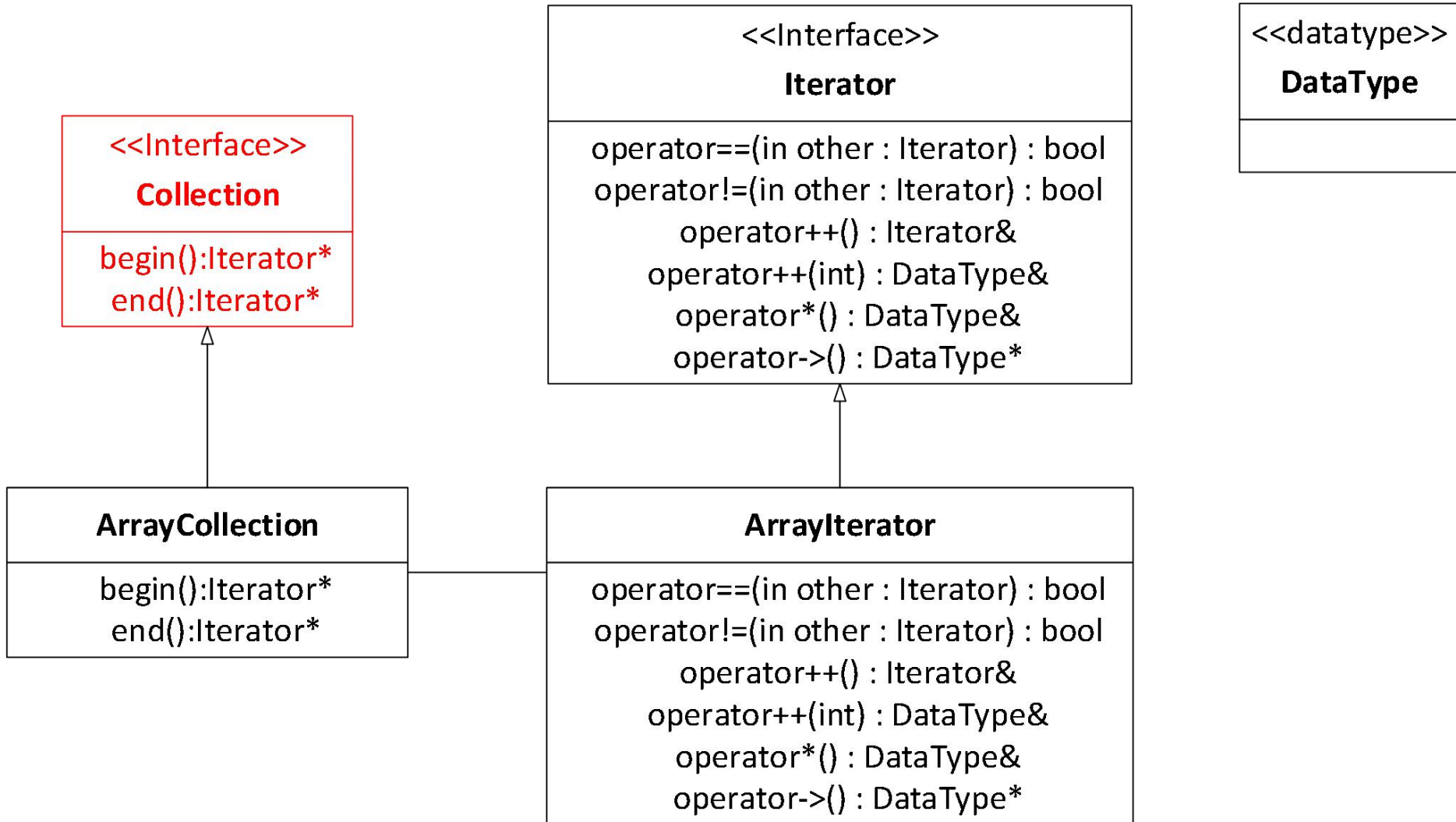
迭代器

- 把数据“访问”设计为一个统一接口，形成迭代器
- 这个迭代器可以套接在任意的数据结构上

//迭代器基类

```
class Iterator {  
public:  
    virtual ~Iterator() { }  
    virtual Iterator& operator++() = 0;  
    virtual float& operator++(int) = 0;  
    virtual float& operator*() = 0;  
    virtual float* operator->() = 0;  
    virtual bool operator!=(const Iterator &other) const = 0;  
    bool operator==(const Iterator &other) const {  
        return !(*this != other);  
    }  
};
```

实现Collection基类

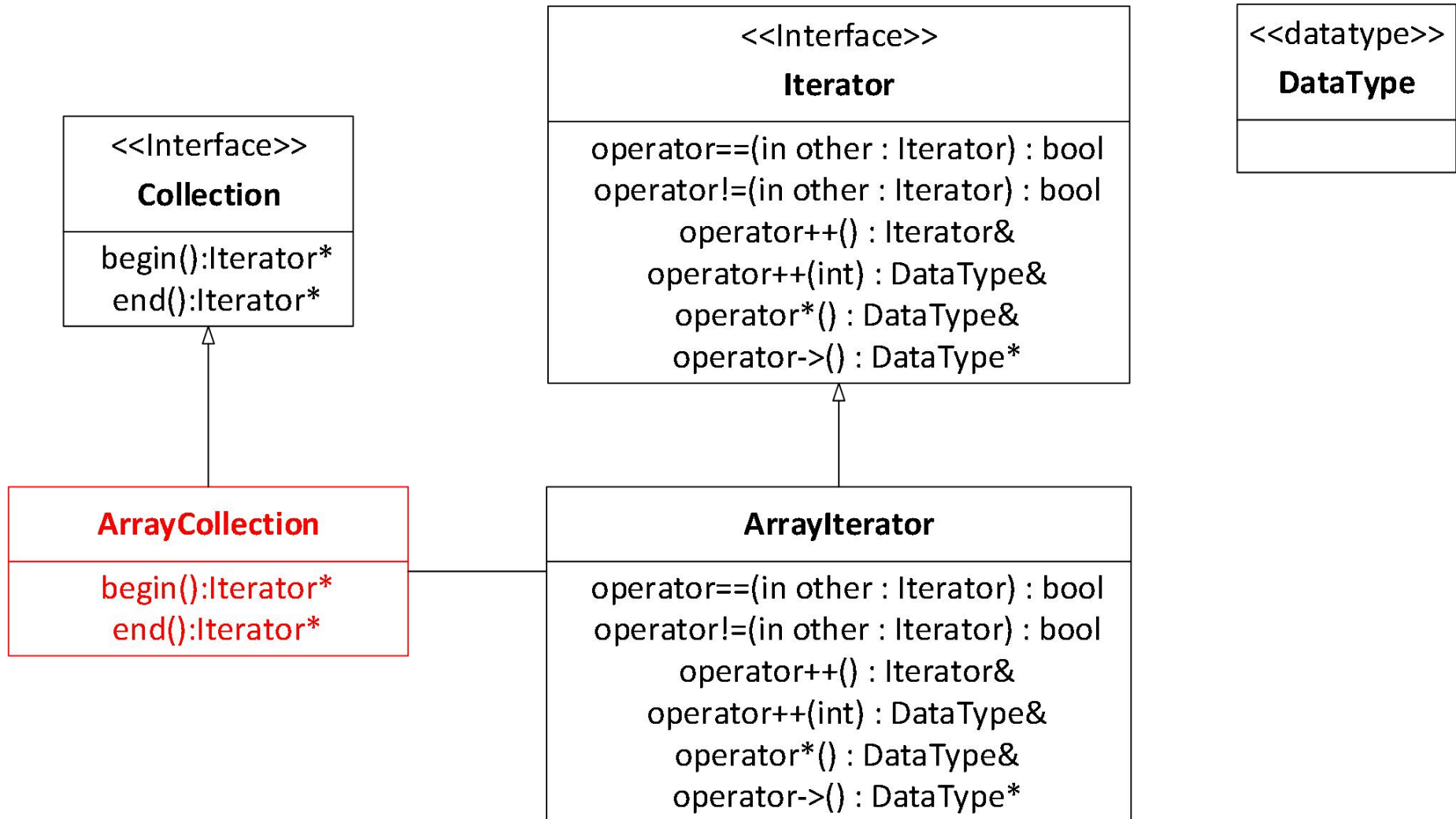


存储器

- 定义数据的存储结构基类Collection
- 需要给“存储”对象一个约束
 - 能够返回代表“头”和“尾”的迭代器
 - 使用“左闭右开区间”，即[begin, end)

```
class Collection {  
public:  
    virtual ~Collection() { }  
    virtual Iterator* begin() const = 0;  
    virtual Iterator* end() const = 0;  
    virtual int size() = 0;  
};
```


实现基于数组的Collection



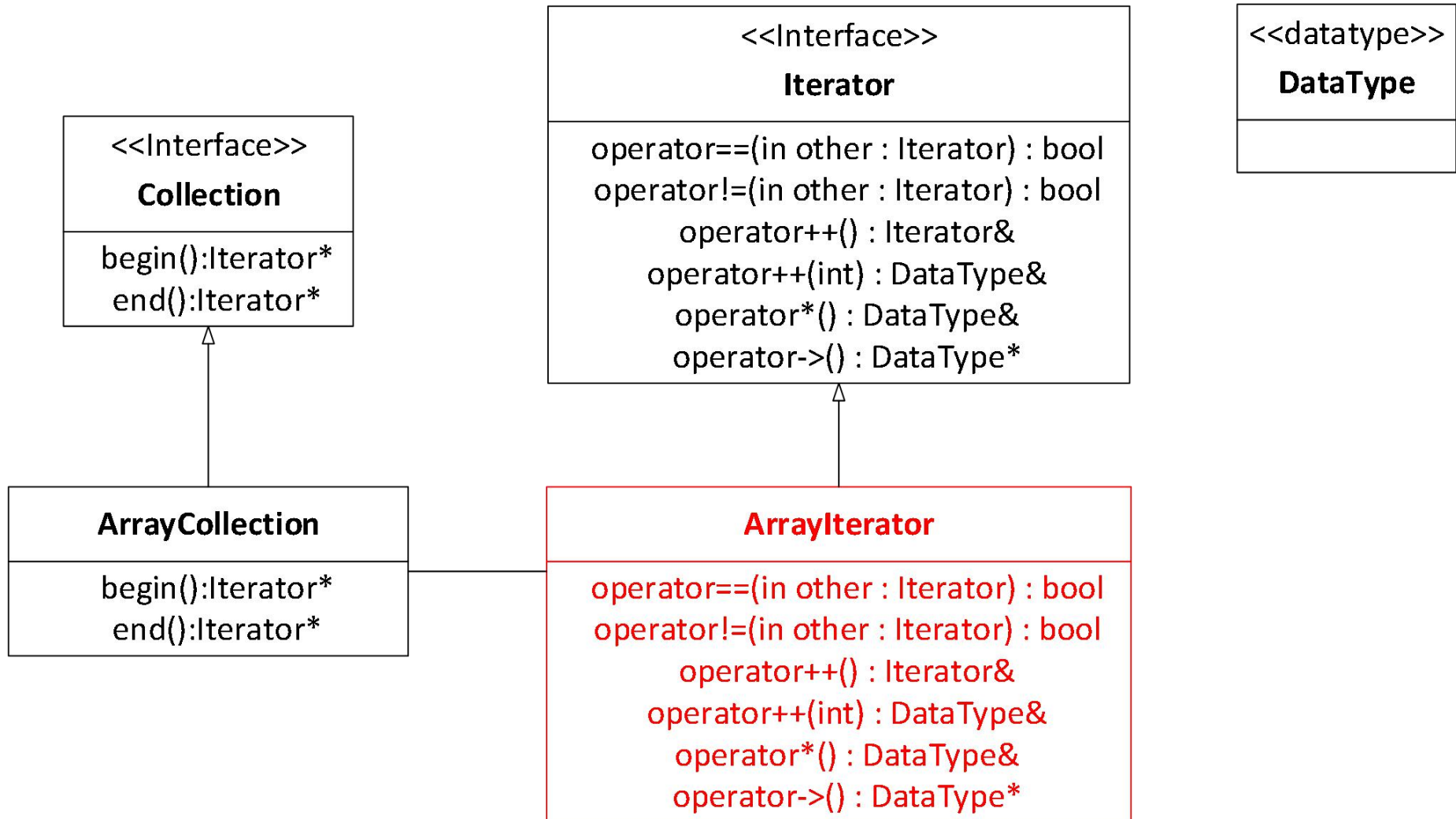
实现基于数组的Collection

```
class ArrayCollection : public Collection { //底层为数组的存储结构类
    friend class ArrayIterator; //friend可以使得配套的迭代器类可以访问数据
    float* _data;
    int _size;
public:
    ArrayCollection() : _size(10){_data = new float[_size]; }
    ArrayCollection(int size, float* data) : _size(size) {
        _data = new float[_size]; //开辟数组空间用以存储数据
        for (int i = 0; i < size; i++)
            *(_data+i) = *(data+i);
    }
    ~ArrayCollection() { delete[] _data; }
    int size() { return _size; }
    Iterator* begin() const;
    Iterator* end() const;
};

Iterator* ArrayCollection::begin() const { //头迭代器，并放入相应数据
    return new ArrayIterator(_data, 0);
}

Iterator* ArrayCollection::end() const { //尾迭代器，并放入相应数据
    return new ArrayIterator(_data, _size);
}
```

实现基于数组的Iterator



实现基于数组的Iterator

```
//继承自迭代器基类并配套ArrayCollection使用的迭代器
class ArrayIterator : public Iterator {
    float *_data;    //ArrayCollection的数据
    int _index;      //数据访问到的下标
public:
    ArrayIterator(float* data, int index) :
        _data(data), _index(index) { }
    ArrayIterator(const ArrayIterator& other) :
        _data(other._data), _index(other._index) { }
    ~ArrayIterator() { }
    Iterator& operator++();
    float& operator++(int);
    float& operator*();
    float* operator->();
    bool operator!=(const Iterator &other) const;
};
```

Iterator对Collection的数据访问

//迭代器各种内容的实现

```
Iterator& ArrayIterator::operator++() {  
    _index++; return *this;  
}
```

//因为是数组，所以直接将空间指针位置+1即可，可以思考下这里为什么返回float&，而不是Iterator

```
float& ArrayIterator::operator++(int) {  
    _index++;  
    return _data[_index - 1];  
}
```

//对data的内存位置取值

```
float& ArrayIterator::operator*() {  
    return *(_data + _index);  
}
```

```
float* ArrayIterator::operator->() {  
    return (_data + _index);  
}
```

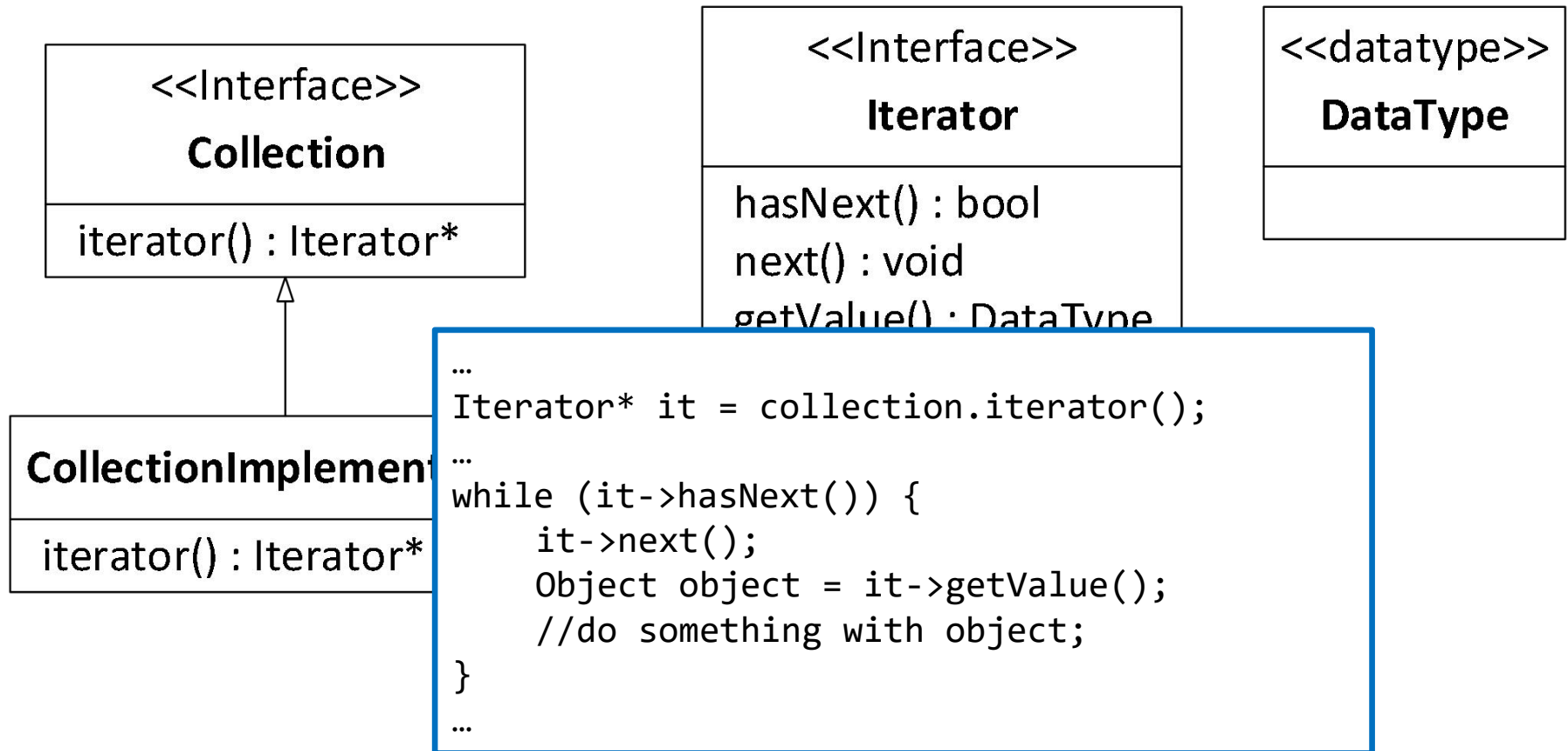
//判断是不是指向内存的同一位置

```
bool ArrayIterator::operator!=(const Iterator &other) const {  
    return (_data != ((ArrayIterator*)&other)->_data ||  
            _index != ((ArrayIterator*)&other)->_index);  
}
```

main()

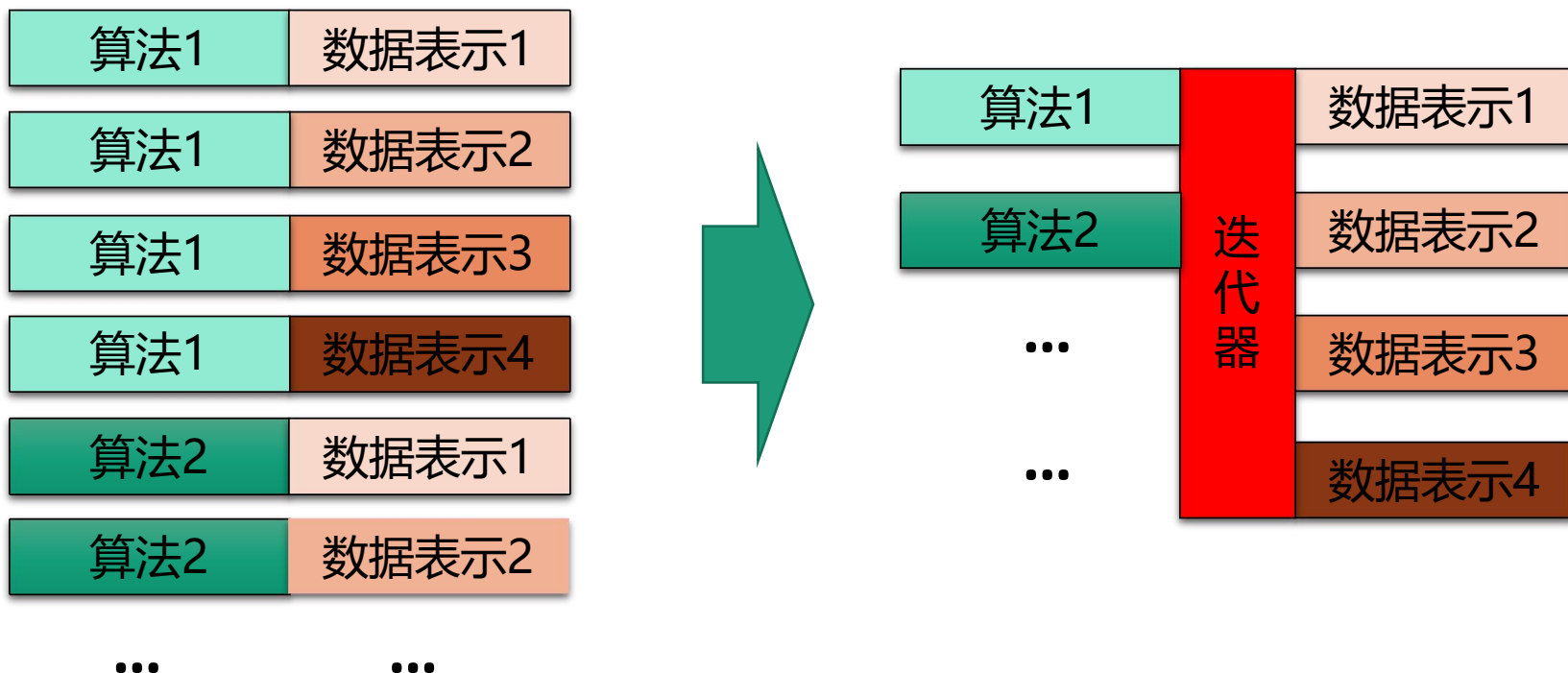
```
int main(int argc, char *argv[]) {  
  
    float scores[]={90, 20, 40, 40, 30, 60, 70, 30, 90, 100};  
    Collection *collection = new ArrayCollection(10, scores);  
  
    Iterator* begin = collection -> begin();  
    Iterator* end = collection -> end();  
    int passed = 0;  
  
    for (Iterator* p = begin; *p != *end; (*p)++) {  
        if (**p >= 60)  
            passed ++;  
    }  
    cout << passed << endl; // 5  
  
    return 0;  
}
```

另一种常见的迭代器模式



总结

- 迭代器模式实现了 **算法** 和 **数据存储** 的隔离
- 规避了为每一个算法和数据存储的组合均进行代码实现的巨大工作量



STL

- C++的STL中提供了大量的数据容器
- 这些容器背后支持的数据结构是不同的，但是类似的数据访问操作，如遍历、最大值、最小值.....
- STL繁多的数据结构均采用了类似的设计架构来抽象访问接口
- 可以阅读STL的具体实现代码来体会迭代器模式的特点

模板方法

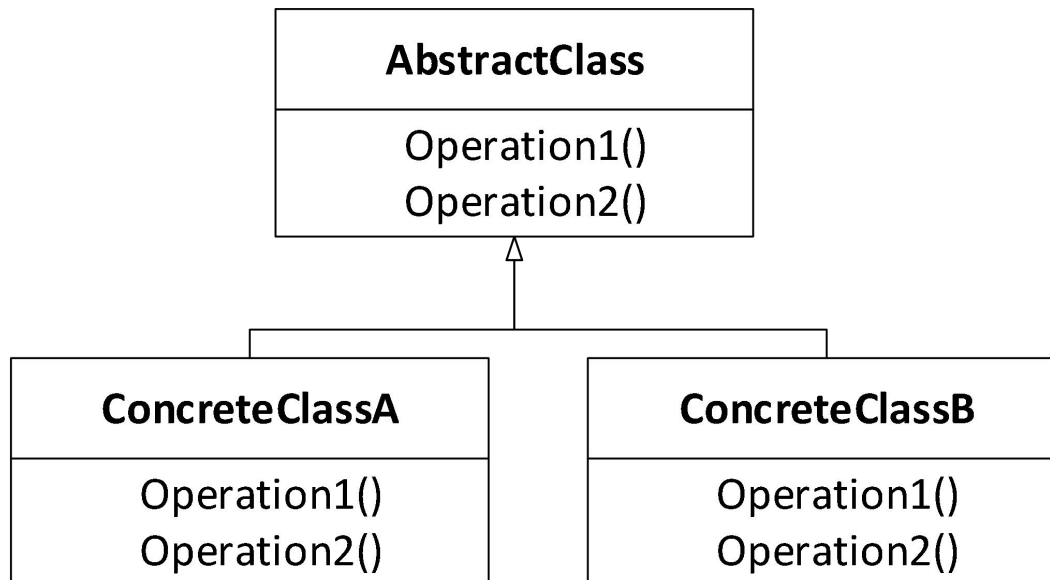
- 迭代器模式实现了 **算法** 和 **数据存储** 的隔离
- 一个迭代器底层的数据可以通过数组或者指针等不同的方式来存储
- 而针对不同的底层实现，上层的迭代器接口需保持一致，这里便使用到了 **模板方法**
- 模板方法可以通过C++的 **继承与多态** 或者 **template关键字** 来实现

模板方法

Template Method

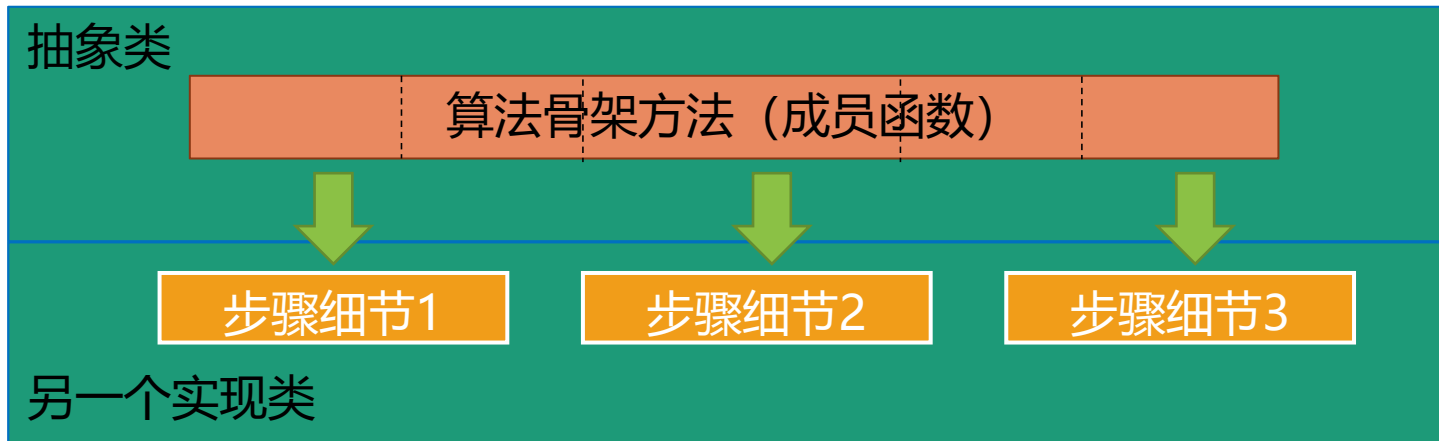
模板方法

- 在接口的一个方法中定义算法的骨架
- 将一些步骤的实现延迟到子类中
- 使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

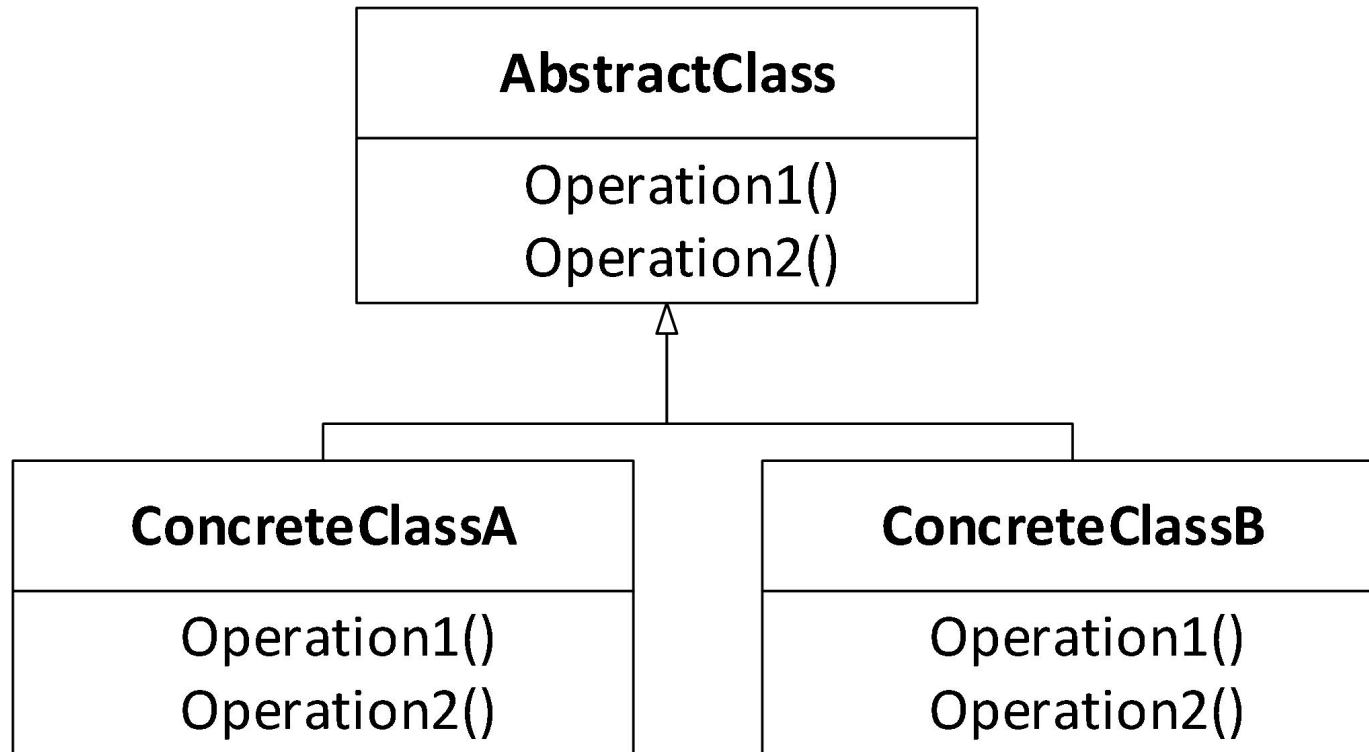


模板方法

- 抽象类（父类）定义算法的骨架
- 算法的细节由实现类（子类）负责实现
- 在使用时，调用抽象类的算法骨架方法，再由这个方法来根据需要调用具体类的实现细节
- 当拓展一个新的实现类时，重新继承与实现即可，无需对已有的实现类进行修改



实现示例



代码实现

```
class AbstractClass {  
public:  
    virtual void operation1() = 0;  
    virtual void operation2() = 0;  
  
    void run() { // 定义一个算法流程  
        operation1();  
        operation2();  
    }  
};
```

代码实现

```
class ConcreteA : public AbstractClass {
public:
    void operation1() { // 定义不同的操作
        cout << "ConcreteA::operation1" << endl;
    }
    void operation2() {
        cout << "ConcreteA::operation2" << endl;
    }
};

class ConcreteB : public AbstractClass {
public:
    void operation1() { // 定义不同的操作
        cout << "ConcreteB::operation1" << endl;
    }
    void operation2() {
        cout << "ConcreteB::operation2" << endl;
    }
};
```


代码实现

```
int main() {
    AbstractClass* absClass[] = {new ConcreteA(), new ConcreteB()};

    for (auto x: absClass) {
        x -> run();
        delete x;
    }

    return 0;
}

/* Output:
ConcreteA::operation1
ConcreteA::operation2
ConcreteB::operation1
ConcreteB::operation2
*/
```

针对接口编程

- 模板方法其实就是一种**针对接口编程**的设计
- 通过抽象出“**抽象概念**”，设计出描述这个抽象概念的**抽象类**，或称为“**接口类**”，这个类有一系列的（纯）虚函数，描述了这个类的“接口”
- 对这个接口类进行继承并实现这些（纯）虚函数，从而形成这个抽象概念的“**实现类**”——实现可以有很多种
- 在使用这个概念的时候，我们**使用接口类**来引用这个概念，而不直接使用实现类，从而避免实现类的改变造成整个程序的大规模变化

开放封闭原则


■ 模板方法很好的体现了开放封闭原则

- 对扩展开放，有新需求或变化时，可以方便地现有代码进行扩展，而无需整体变动
- 对修改封闭，新的扩展类一旦设计完成，可以独立完成其工作，同样不需要整体变动

■ 开放封闭原则的核心就是在结构层面上解耦，对抽象进行编程，而不对具体编程

- 抽象结构是简单与稳定的
- 具体实现是复杂与多变的

需求变化

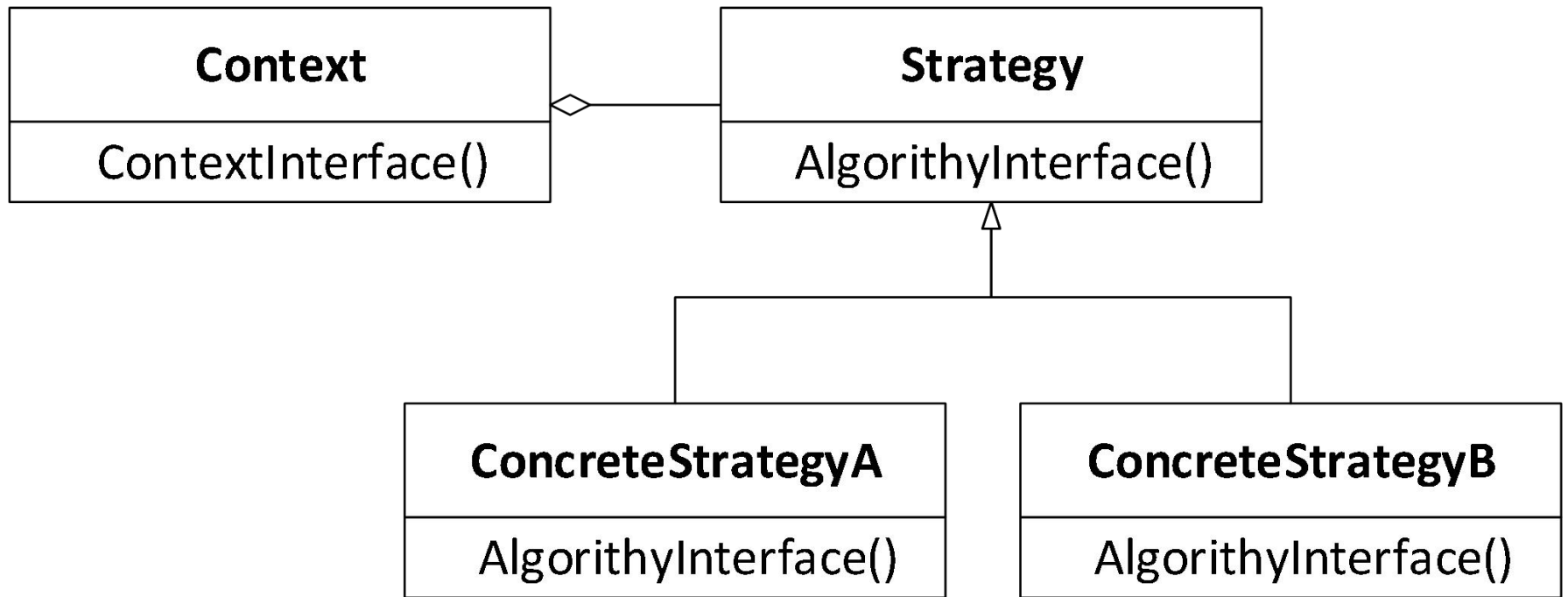
- 如果 `operation1()`、`operation2()` 这几个函数接口的实现方法互相独立
 - 假设
 - `operation1()` 有 n 种实现
 - `operation2()` 有 m 种实现
 - 使用模板方法，我们将需要实现 $n*m$ 个子类
 - 这样充斥大量冗余的实现方式是不可取的
- 

策略模式

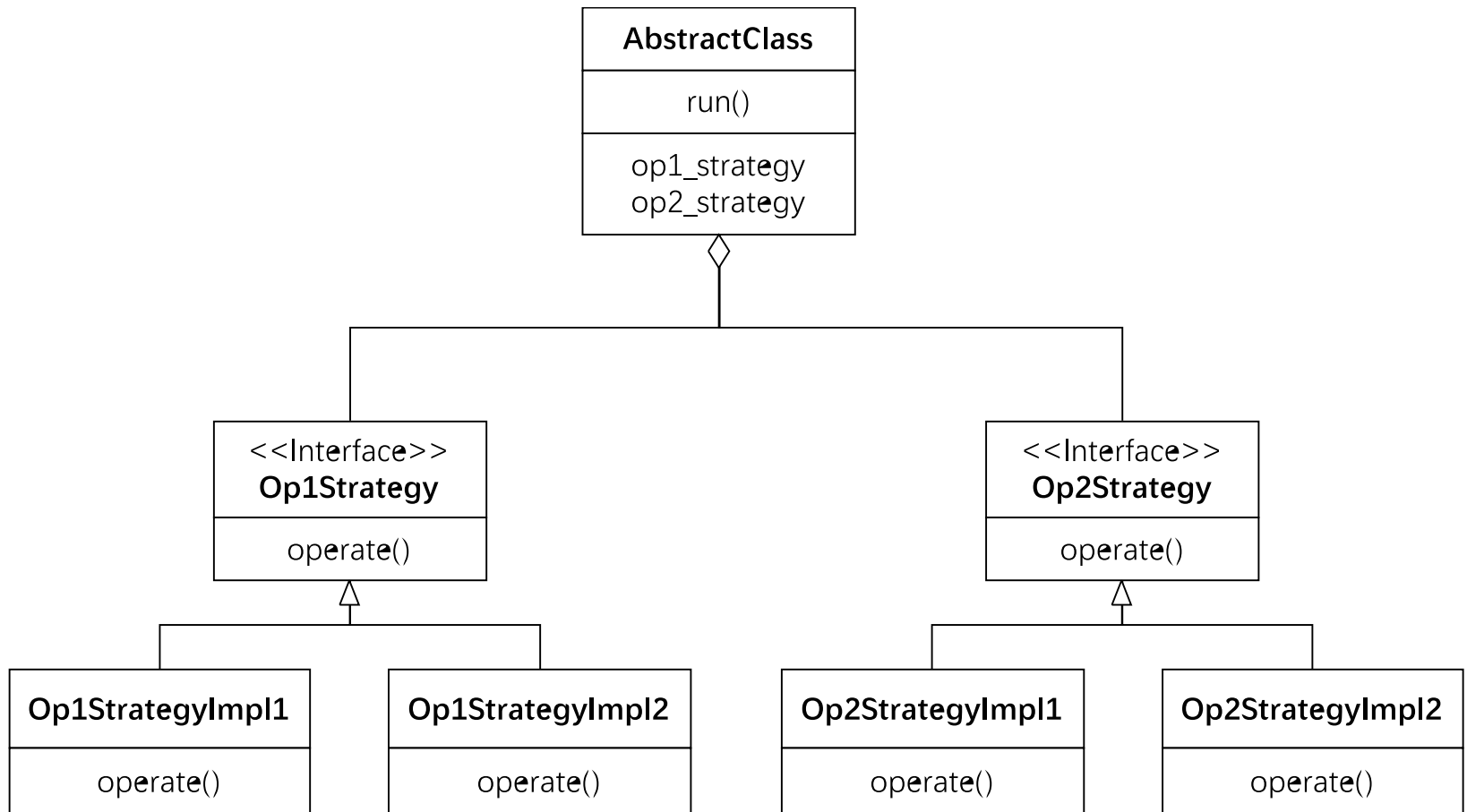
Strategy

策略 (Strategy) 模式

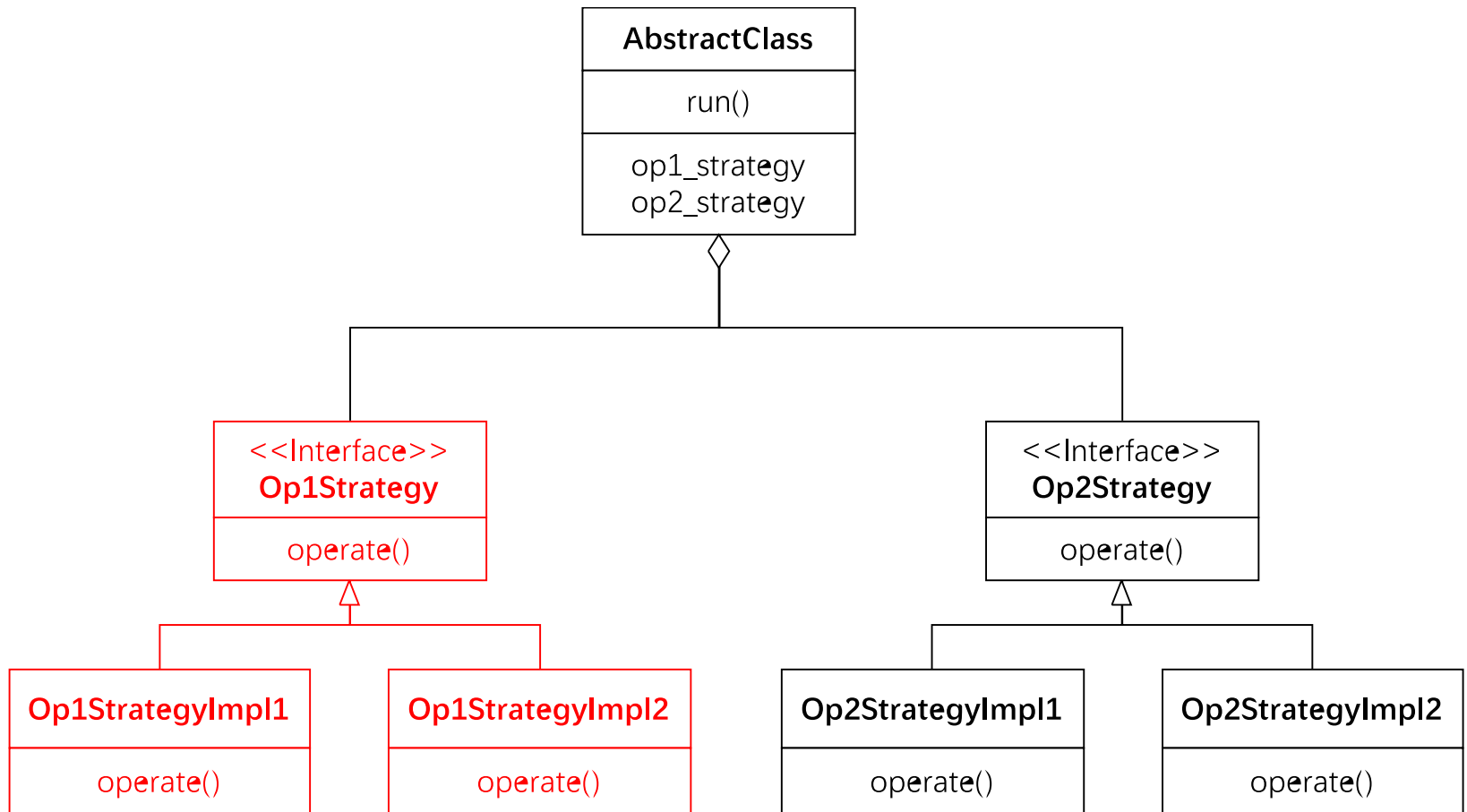
- 定义一系列算法并加以封装，使得这些算法可以互相替换



具体化到我们的问题



实现Op1Strategy



代码实现

// 操作1策略基类

```
class Op1Strategy {  
public:  
    virtual void operate() = 0;  
}
```

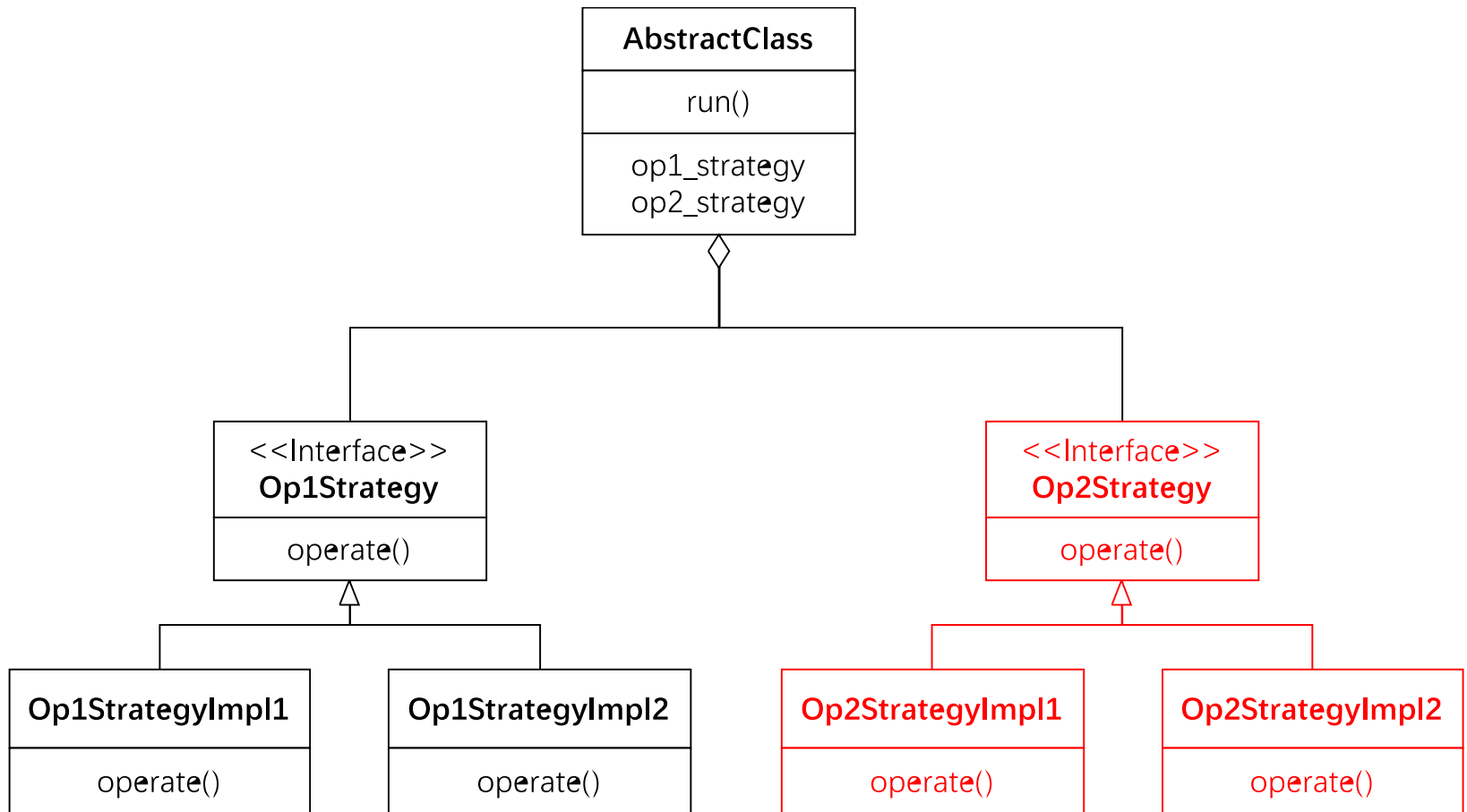
// 操作1具体实现1

```
class Op1StrategyImpl1: public Op1Strategy {  
public:  
    void operate() {  
        cout << "Operation1 Implementation 1" << endl;  
    }  
}
```

// 操作1具体实现2

```
class Op1StrategyImpl2: public Op1Strategy {  
public:  
    void operate() {  
        cout << "Operation1 Implementation 2" << endl;  
    }  
}
```

实现Op2Strategy



代码实现

// 操作2策略基类

```
class Op2Strategy {  
public:  
    virtual void operate() = 0;  
};
```

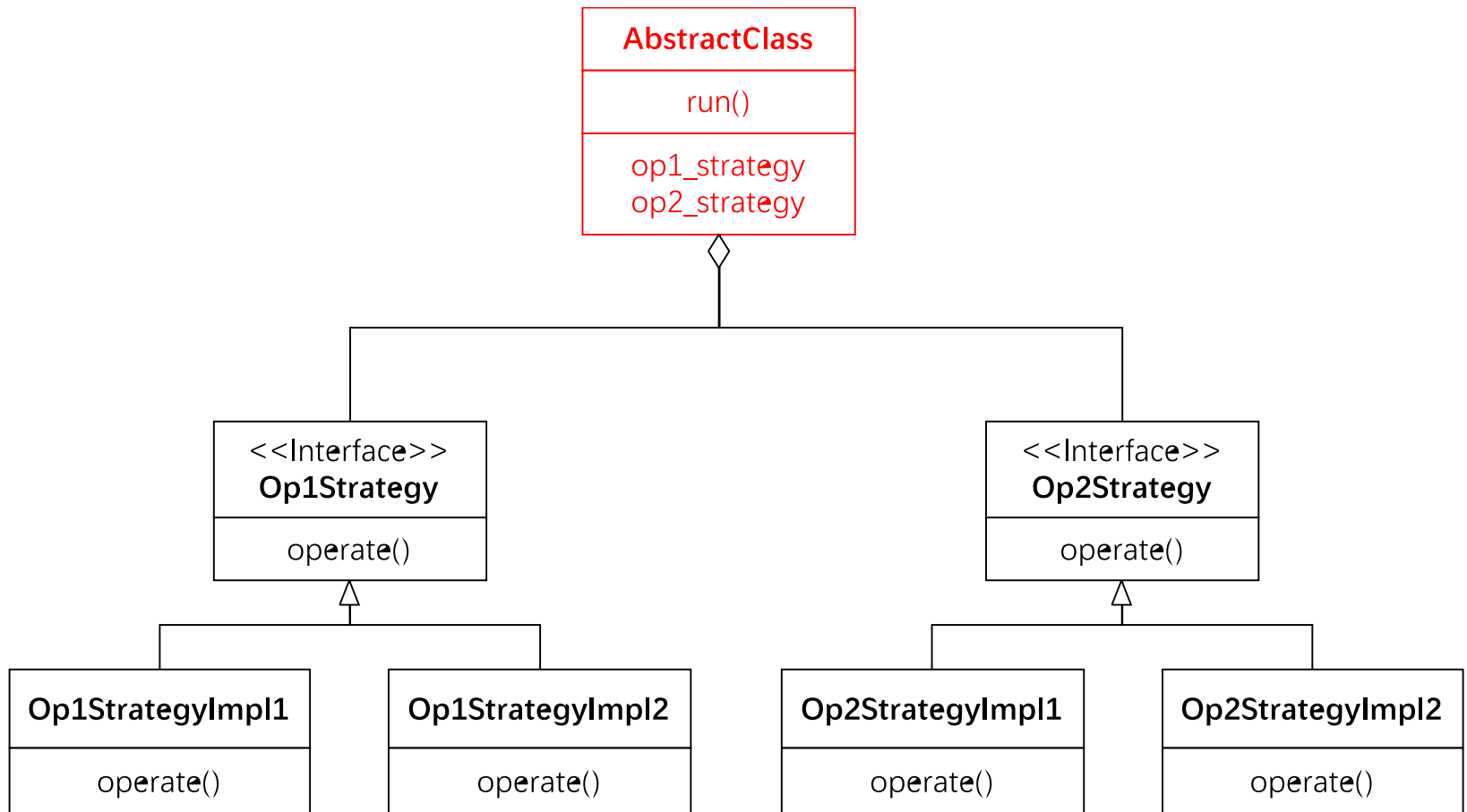
// 操作2具体实现1

```
class Op2StrategyImpl1: public Op2Strategy {  
public:  
    void operate() {  
        cout << "Operation2 Implementation 1" << endl;  
    }  
};
```

// 操作2具体实现2

```
class Op2StrategyImpl2: public Op2Strategy {  
public:  
    void operate() {  
        cout << "Operation2 Implementation 2" << endl;  
    }  
};
```

实现AbstractClass



代码实现

```
class AbstractClass {
public:
    // 各个策略类的组合
    AbstractClass(Op1Strategy* op1, Op2Strategy* op2) :
        op1_strategy(op1), op2_strategy(op2) {}

    // 定义操作流程
    void run() {
        op1_strategy->operate();
        op2_strategy->operate();
    }
private:
    // 获取不同的策略类
    Op1Strategy *op1_strategy;
    Op2Strategy *op2_strategy;
};
```

代码实现

```
int main() {  
    // 为每个策略选择具体的实现算法  
    Op1StrategyImpl1 op1_impl1;  
    Op2StrategyImpl2 op2_impl2;  
  
    // 构造函数  
    AbstractClass absClass(&op1_impl1, &op2_impl2);  
  
    // 运行算法流程  
    absClass.run();  
  
    return 0;  
}  
  
/* Output:  
Operation1 Implementation 1  
Operation2 Implementation 2  
*/
```

调用过程

```
int main() {  
    ...  
    absClass.run();  
}
```

统一的策略调用接口

```
void AbstractClass::run() {  
    op1_strategy->operate();  
}
```

```
void Op1StrategyImpl1::operate()  
{  
    ...  
}
```

```
void Op1StrategyImpl2::operate()  
{  
    ...  
}
```

现在的类数量

■ 假设

- operation1() 有 n 种实现
- operation2() 有 m 种实现

■ 我们需要实现

- 1个AbstractClass类
- 2个抽象策略类（接口）
- $n+m$ 个策略实现类（实现）

■ 策略模式极大的降低了代码冗余， $(n+m+2+1)$ vs $(n*m+1)$

单一责任原则

■ 策略模式很好的体现了单一责任原则

- 一个类（接口）只负责一项职责
- 不要存在多于一个导致类变更的原因

■ 如果一个类承担的职责过多，职责之间的耦合度会很大

- 职责的变化可能会削弱或者抑制这个类完成其他职责的能力
- 多变的场景会使得整体程序的设计遭受破坏，维护难度增大

■ 单一职责原则的核心就是在功能层面上解耦

模板方法VS策略

- 当我们需要给出Operation2的一个新的实现

模板方法

- 针对所有 operation1() ，都要实现一组新的子类
- 需要实现新子类（实现类）： **n** 个

策略

- 无需修改Op1Strategy和Op2Strategy，只要实现一个新的Op2StrategyImpl3实现类即可
- 需要实现新子类（实现类）： **1** 个

模板方法VS策略

模板方法

- 定义一个操作中的算法的骨架，而将具体实现步骤延迟到子类中。子类可以不改变算法的结构即可重定义该算法的某些特定步骤
- 优先**继承行为**，重视**功能的抽象与归纳**
- 优点：
 - 基类高度抽象统一，逻辑简洁明了
 - 子类之间关联不紧密时易于简单快速实现
 - 封装性好，实现类内部不会对外暴露
- 弊端：
 - 接口同时负责所有的功能（算法）
 - 任何算法的修改都导致整个实现类的变化

策略

- 定义一系列的算法，把它们一个个封装起来,使它们可相互替换。本模式使得算法可独立于使用它的客户而变化
- 优先**组合行为**，重视**功能的划分与组合**
- 优点：
 - 每个策略只负责一个功能，易于拓展。
 - 算法的修改被限制在单个策略类的变化中，任何算法的修改对整体不造成影响
- 弊端：
 - 在功能较多的情况下结构复杂
 - 策略组合时对外暴露，封装性相对较差

模板方法VS策略

- 模板方法和策略模式都是解决算法多样性对代码结构冲击的问题。业务相对简单时，策略模式和模板方法几乎等效。
- 模板方法更加侧重于逻辑复杂但结构稳定的场景，尤其是其中的某些步骤（部分功能）变化剧烈且没有相互关联。
- 策略模式则适用于算法（功能）本身灵活多变的场景，且多种算法之间需要协同工作。

案例：语言集成查询LINQ

- 继续我们的LINQ实现.....
- from函数之后，我们得到了一个ling_enumerable的对象，而后需要支持链式的函数操作
- select、where将定义成ling_enumerable的成员函数，每次操作后都将当前迭代器对象包装（装饰）一次，并重新返回一个 ling_enumerable 对象

案例：语言集成查询LINQ

■ select 函数实现

- 使用模板参数 **TFunction** 接受不同的输入函数
- 将当前对象中保留的 `_begin` 和 `_end` 迭代器重新包装成 **`select_iterator<TIterator, TFunction>`** 类型的迭代器。

```
template<typename TFunction>
auto select(const TFunction& f) const
    ->linq_enumerable<select_iterator<TIterator, TFunction>>
{
    return linq_enumerable<select_iterator<TIterator, TFunction>>(
        select_iterator<TIterator, TFunction>(_begin, f),
        select_iterator<TIterator, TFunction>(_end, f)
    );
}
```

案例：语言集成查询LINQ

■ select 函数实现

- **select_iterator<TIterator, TFunction>** 类构造函数中需要两个参数，一个是需要转换的迭代器，一个是当前的操作函数f

```
template<typename TFunction>
auto select(const TFunction& f) const
    ->linq_enumerable<select_iterator<TIterator, TFunction>>
{
    return linq_enumerable<select_iterator<TIterator, TFunction>>(
        select_iterator<TIterator, TFunction>(_begin, f),
        select_iterator<TIterator, TFunction>(_end, f)
    );
}
```

案例：语言集成查询LINQ

■ select_iterator 类实现

- 该类使用两个模板参数 **TIterator** 和 **TFunction**，分别保存相应的迭代器和函数

```
template<typename TIterator, typename TFunction>
class select_iterator
{
    typedef select_iterator<TIterator, TFunction> TSelf;
    // 用TSelf简化当前迭代器类型表示
private:
    TIterator iterator; // 保存迭代器
    TFunction f; // 保存操作函数
public:
    select_iterator(const TIterator& i, const TFunction& _f) :
        iterator(i), f(_f) {}
    .....
};
```


案例：语言集成查询LINQ

■ select_iterator 类实现

■ 除上述定义外，还需定义一些迭代器基本函数

```
template<typename TIterator, typename TFunction>
class select_iterator
{
    .....
public:
    .....
    TSelf& operator++() { // 迭代器自增操作
        ++iterator;
        return *this;
    }
    bool operator==(const TSelf& it) const { // 判断迭代器相等操作
        return it.iterator == iterator;
    }
    bool operator!=(const TSelf& it) const { // 判断迭代器不相等操作
        return it.iterator != iterator;
    }
};
```

案例：语言集成查询LINQ

■ select_iterator 类实现

- 决定select函数功能的取值操作
- 在取每个具体元素的值之前，需要使用函数f对其值进行处理后再返回
- 使用了auto和decltype来延迟确定取值操作的返回值类型

```
template<typename TIterator, typename TFunction>
class select_iterator
{
    .....
public:
    .....
    auto operator*()const -> decltype(f(*iterator)) // 迭代器取值操作
    {
        return f(*iterator);
    }
};
```

案例：语言集成查询LINQ

■select使用实例

```
int v[] = { 1, 2, 3, 4, 5 };  
  
auto q = from(v)  
        .select([](int x) { return x * x; }); // 进行平方操作  
  
for (auto x : q) {  
    cout << x << " "; // 1 4 9 16 25  
}
```

案例：语言集成查询LINQ

■ where 函数实现

- where 的实现与 select 类似，也是 `linq_enumerable` 类的一个成员函数
- 仍需定义 `where_iterator` 类
- 和 `select` 不同的功能实现体现在 `where_iterator` 类的函数实现中

案例：语言集成查询LINQ

■ where 函数实现

- where_iterator 的构造需要接受三个参数 **iterator**、**end** 和 **f**

```
template<typename TFunction>
auto where(const TFunction& f) const
    -> linq_enumerable<where_iterator<TIterator, TFunction>>
{
    return linq_enumerable<where_iterator<TIterator, TFunction>>(
        where_iterator<TIterator, TFunction>(_begin, _end, f),
        where_iterator<TIterator, TFunction>(_end, _end, f)
    );
}
```

案例：语言集成查询LINQ

■ where_iterator 类实现

- 该类使用两个模板参数 **TIterator** 和 **TFunction**，增加了一个 **end** 成员变量，为实际的 end 迭代器

```
template<typename TIterator, typename TFunction>
class where_iterator {
    typedef where_iterator<TIterator, TFunction> TSelf;
private:
    TIterator iterator;
    TIterator end;
    TFunction f;
public:
    .....
};
```

案例：语言集成查询LINQ

■ where_iterator 类实现

- 构造函数中，输入的iterator需要不断自增，直到找到符合f函数约束的元素
- 实现了初始化时的迭代器头部元素筛选

```
template<typename TIterator, typename TFunction>
class where_iterator {
    .....
public:
    where_iterator(const TIterator& i, const TIterator& e, const
TFunction& _f) : iterator(i), end(e), f(_f) {
        while (iterator != end && !f(*iterator)) {
            ++iterator;
        }
    }
    .....
};
```

案例：语言集成查询LINQ

■ where_iterator 类实现

- 在自增操作中，也需要找到首个满足f函数约束的元素
- 实现了迭代时的元素筛选

```
template<typename TIterator, typename TFunction>
class where_iterator {
    .....
public:
    TSelf& operator++(){
        if (iterator == end) return *this;
        ++iterator;
        while (iterator != end && !f(*iterator)) {
            ++iterator;
        }
        return *this;
    }
    .....
};
```


案例：语言集成查询LINQ

■ where_iterator 类实现

- 取值操作中，返回值的类型是迭代器内 **元素的类型**，定义 **iterator_type**
 - 将空指针 nullptr 强制转换为指向 TIterator 的指针
 - 对其解引用得到一个不存在的 TIterator 对象 *(TIterator*)nullptr
 - 再对迭代器对象进行解引用，即可得到迭代器元素
 - 最后对其使用 **decltype** 操作，得到元素类型

```
template<typename TIterator, typename TFunction>
class where_iterator {
    .....
    typedef decltype(**(TIterator*)nullptr) iterator_type;
public:
    iterator_type operator*() const {
        return *iterator;
    } .....
};
```

案例：语言集成查询LINQ

■ where_iterator 类实现

■ ==和!=的重载与select类似

```
template<typename TIterator, typename TFunction>
class where_iterator {
    .....
public:
    bool operator==(const TSelf& it) const {
        return it.iterator == iterator;
    }

    bool operator!=(const TSelf& it) const {
        return iterator != it.iterator;
    }
    .....
};
```

案例：语言集成查询LINQ

■where使用实例

```
int v[] = { 1, 2, 3, 4, 5 };  
  
auto q = from(v)  
         .where([](int x) { return x % 2 == 0; }); // 选择偶数  
  
for (auto x : q) {  
    cout << x << " "; // 2 4  
}
```

案例：语言集成查询LINQ

- 至此，简单的LINQ的C++实现便完成了。
- 在select和where函数的实现中，我们只需要使用迭代器进行相关的操作（如自增，取值操作），而具体的实现方式由各个迭代器内部进行相应实现
- 也就是说，select和where函数定义了一套算法**模板**，而不同的迭代器内部提供了具体的算法实现
- 这里具体使用了template关键字（如TIterator、TFunction）来实现**模板方法**

本节课

- 行为型设计模式关心对象之间的行为功能抽象，核心在于抽象行为功能中不变的成分，具体实现行为功能中变的成分，保证以尽可能少的代码改动完成功能的增减
- 迭代器模式抽象了数据访问方法，可以访问对象的元素但却不暴露底层实现，隔离具体算法与数据结构
- 模板方法归纳了一系列类的通用功能，在基类中将功能的接口固定，在子类中具体实现流程细节，使得新类的增加不对已有类产生影响
- 策略模式抽象了功能的选择与组合，隔离不同的功能使得相互之间不受影响，可以灵活支持算法或策略的变动

课后阅读

■ 《用 C++ 11 来实现 LINQ to Object》

- 除本讲提到的三个语句实现外，还提供了更加详细的 LINQ 语句实现
- 链接 代码

结束