Higher order procedures in C++

Presentation

Minjun Kim[1] Syed Asad Hussain[2]

[1]Computer Science Specialist - Software Engineering Stream
University of Toronto

[2]Computer Science Specialist - Information Systems Stream
University of Toronto

CSCC24, Winter 2024

Table of Contents

Introduction

- Pronounced "C plus plus"
- Object Oriented
- Super set of C
- Syntax similar to C and Java
- Complied Langauge: can use Clang, GNU Collection (GCC)...
- Statically typed
- Applications in:
  - Operating Systems
  - Games and Graphics
  - Embedded Systems
  - and more [1]

[1] https://www.softwaretestinghelp.com/cpp-applications/

Lambda Expressions in C++

General syntax of a lambda expression in C++ is as follows:

```
[capture_list] (parameters) mutable exception -> return_type {
    function_body
}
```

capture list*: list of variables that will be used in the function body but are defined outside the lambda expression

parameters: list of parameters, ex: (int a, float b,..), or empty ()

mutable: optional, specified whether the variables passed in the captured list are allowed to be modified

exception: optional, indicates whether an exception can be/cannot be thrown (noexcept)

Lambda Expressions: Capture List

- Specifies which variables are captured, similar to *closure*...but not quite!
- 4 ways to capture variables (from the enclosing environment)
  - `[]` - capture nothing
  - `[&]` or `[&varname,...]` - capture by reference
  - `[varname,...]` or `[=]` - capture by value
    - bound at time of deceleration
    - variable must be initialized
    - need to use mutable keyword if modifying captured variable
  - `[&varname1, varname2,...]` - both
- What NOT to do:
  - `[&, &varname]`, similarly for [=,...]

UNIVERSITY OF TORONTO

Lambda Expression Example

```
int main() {
    int c = [](int a, int b) -> int {return a + b;}(1, 2);
}
```

Lambdas in Haskell, Racket, Java, C++

- Closure:
  - Haskell, Racket: lexical scoping
  - C++: lexical scoping + more
  - Java: captures effectively final variables in scope
- Currying:
  - Haskell, Racket: support built-in currying
  - C++, Java: no built-in support, however, can model using different techniques such as functors and functional interfaces
- Enables creation of HOPs to various degrees in C++ and Java

Filter in C++

- Until much recently, there was no builtin function filter in c++
- In c++11: std::copy_if or std::remove_if
- In c++20: using std::views::filter in the ranges library
    - ranges, with views, are conceptually similar to java streams api and can be used in composition with the pipeline operator (|) to chain operations together

Two Simple Filter Examples

Using std::copy_if

```
int main() {
    std::vector<int> foo = {1, 2, 3, 4, 5};
    std::vector<int> evenFoo = {};

    std::copy_if(foo.begin(), foo.end(),
        std::back_inserter(evenFoo),
        [] (int x) {return x % 2 == 0})
}
```

Using ranges library

```
#include<ranges>
int main() {
     std::vector<int> foo = {1, 2, 3, 4, 5};

     auto evenFoo = std::views::filter(foo,
        [] (int x) {return x % 2 == 0});
}
```

Filter in Haskell, Racket, Java, C++

- Same purpose in all four languages, used on collections/containers
- Underlying representation remains unchanged
- Can be used in conjunction with other functions:
  - C++ : using views, ranges, pipe
  - Java: through streams api
  - Racket, Haskell functional programming languages
- Support lazy evaluation for filter

Introduction: Map in C++

- The concept of *mapping* over a collection to produce a new transformed collection is a cornerstone of functional programming.
- In C++, this functionality is captured by the `std::transform` algorithm in the Standard Template Library (STL).
- `#include <algorithm>`
- Let us examine `std::transform` and draw connections to *map* functions in Racket, Haskell, and Python, discussing the advantages and disadvantages of its use in C++.

Understanding `std::transform`

- `std::transform` applies a given function to a range of elements from one or two input ranges and stores the result in an output range.
- Commonly used with lambda expressions for inline transformations.
- Signature:
  `OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first, UnaryOperation unary_op);`
- It is overloaded to support both unary and binary operations.

Function Signature for std::transform

- Unary operation signature:
  `OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first, UnaryOperation unary_op);`
- Binary operation signature:
  `OutputIt transform(InputIt first1, InputIt last1, InputIt first2, OutputIt d_first, BinaryOperation binary_op);`

Parameter Explanation for std::transform

- `InputIt first1, last1`: Iterators specifying the beginning and end of the first input range.
- `InputIt first2`: An iterator to the beginning of the second input range.
- `OutputIt d_first`: An iterator to the beginning of the destination range where the results are stored.
- `UnaryOperation unary_op`: A unary function that is applied to each element in the input range.
- `BinaryOperation binary_op`: A binary function that is applied to pairs of elements from the two input ranges.

Example of std::transform [1/2]

```
vector<int> nums = {1, 2, 3, 4, 5};
vector<int> result;
transform(nums.begin(), nums.end(),
          back_inserter(result),
          [](int x) { return x * x; });

// result now holds {1, 4, 9, 16, 25}
```

- This example squares each number in the vector using a lambda expression.
- The back_inserter creates an insert iterator that adds new elements to the end of result.

Example of `std::transform` [2/2]

```
vector<int> list1 = {0, 5, 10};
vector<int> list2 = {1, 10, 20};
vector<int> result(list1.size());

transform(list1.begin(), list1.end(), list2.begin(),
result.begin(), [](int x, int y) { return 2 * (y - x); });

// result now holds {2, 10, 20}
```

- Applies a binary lambda expression to corresponding elements from each vector.
- Results are stored in a pre-sized `result` vector, ensuring adequate storage.

Mapping in Functional Languages

- Racket and Haskell treat *mapping* as a first-class operation, with `map` being a fundamental function.
- Racket Example: `(map (lambda (x) (* x x)) '(1 2 3 4 5))`
- Haskell Example: `map (\x -> x * x) [1, 2, 3, 4, 5]`
- Python also supports functional-style mapping with its `map` function.
- Python Example: `list(map(lambda x: x * x, [1, 2, 3, 4, 5]))`

Advantages of `std::transform` in C++

- Type safety at compile-time.
- Performance optimization opportunities provided by the compiler.
- Flexibility to use with any STL container supporting iterators.
- Can transform data in-place or into a new container.
- Overloaded versions for both unary and binary operations.

Disadvantages of `std::transform` in C++

- More verbose/convoluted syntax compared to functional languages.
- Requires understanding of iterators and other C++ specific concepts.
- Lack of built-in support for concurrency and parallelism in the standard version (prior to C++17).
- Error messages can be less informative due to template metaprogramming.

Conclusion: Map in C++

- `std::transform` is C++'s version of the map function, optimized for the language's strong typing.
- Not as concise as functional languages but allows for powerful and adaptable data manipulation.
- Understanding how to leverage `std::transform` can help one write more sophisticated, efficient, and legible code.

Introduction to Fold Operations

- Fold operations process a collection to accumulate a result.
- Fundamental in functional programming languages like Racket, Haskell, and also available in Python.
- Recall: both Haskell and Racket support fold operations natively:
  - **Haskell** uses `foldl` and `foldr` for left and right folds.
  - **Racket** provides similar functionality with `foldl` and `foldr`.
- **std::accumulate** for fold-left (`foldl`) from the Standard Template Library (STL).
- A combination of **std::accumulate** with **reverse iterators** or manual function implementation for fold-right (`foldr`).

UNIVERSITY OF TORONTO

Signature of std::accumulate

```
template< class InputIt, class T, class BinaryOperation >
T accumulate(InputIt first, InputIt last, T init,
             BinaryOperation op = std::plus<>());
```

- A template function found in the C++ Standard Library.
- Located in the `<numeric>` header file.

Parameters:

- **InputIt first, InputIt last**: iterators to the beginning and end of the sequence to be accumulated, specifying the range [`first`, `last`) of elements to be included in the accumulation.
- **T init**: initial value for the accumulation. The type `T` is determined by the type of this initial value, and is the return type of the function
- **BinaryOperation op** (optional): A binary function applied to accumulate the values. It takes two arguments of type `T` and returns a single value of type `T`.

Fold Left in C++

```
std::vector<int> nums = {1, 2, 3, 4, 5};

// Use std::accumulate to sum the elements of nums
int sum = std::accumulate(nums.begin(), nums.end(), 0);

// sum gets 15
```

- The `std::accumulate` function computes the sum of elements in `vec`, starting with an initial value of `0`.
- This is analogous to `foldl` in functional programming.

$$(((((0 + 1) + 2) + 3) + 4) + 5) = 15$$

Fold Left in C++

```
std::vector<int> nums = {1, 2, 3, 4, 5};

int product = std::accumulate(nums.begin(), nums.end(), 1,
                              std::multiplies<int>());

// product gets 120
```

- The `std::accumulate` function computes the product of elements in `nums`, starting with an initial value of `1`.
- This process is analogous to `foldl` in functional programming with multiplication.

$$((((1 \times 1) \times 2) \times 3) \times 4) \times 5 = 120$$

Simulating Fold Right in C++

C++ does not have a built-in `foldr` function. However, you can simulate `foldr` by reversing the range or using reverse iterators with `std::accumulate`.

Simulating foldr for Sum

```
std::reverse(nums.begin(), nums.end());
int sumR = std::accumulate(nums.begin(), nums.end(), 0);
```

Note: This method requires modifying the original container or creating a reversed copy.

$$1 + (2 + (3 + (4 + (5 + 0)))) = 15$$

UNIVERSITY OF TORONTO

Comparative Analysis

Comparing fold operations across C++, Haskell, and Racket reveals:

- Haskell and Racket offer more intuitive and concise syntax for fold operations
- C++'s `std::accumulate` provides fold functionality with more verbose syntax, as expected.
- The strong type system in Haskell helps with compile-time safety for fold operations, which is less achievable in C++.

Advantages of Using std::accumulate

- **Simplicity and Readability:** ‘std::accumulate‘ provides a straightforward way to implement fold-left operations, making code easy to read and understand.
- **Versatility:** It can be used with any operation that can be expressed as a binary function, allowing for flexibility in how data is processed and accumulated.
- **Type Safety:** As a part of the C++ Standard Library, ‘std::accumulate‘ benefits from C++’s strong type system, ensuring type safety at compile time.
- **Performance:** C++ compilers can optimize usage of ‘std::accumulate‘, potentially leading to faster execution compared to manual loop implementations.

Disadvantages of Using std::accumulate

- **No Native Fold-Right:** C++ does not provide a native function for fold-right operations within the Standard Library. This requires additional steps, such as reversing the container or crafting a custom fold-right function.
- **Efficiency Concerns with Reverse:** Using ‘std::accumulate‘ for fold-right by reversing the container can introduce overhead, potentially impacting performance.
- **Limited to Binary Operations:** ‘std::accumulate‘ is inherently designed for binary operations, which is not suitable for all use cases.
- **Initial Value Dependency:** The result of ‘std::accumulate‘ is dependent on the initial value, which may not always be intuitive, particularly in operations where the neutral element isn’t obvious.

Conclusion

- Lambda Expressions
- Filter Operations - std::views::filter
- Map Operations - std::transform
- Fold Operations - std::accumulate
- Pros and Cons
- Connection to CSCC24 material

References

1. Lambda Expressions in C++: https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170
2. Function Class in C++: https://learn.microsoft.com/en-us/cpp/standard-library/function-class?view=msvc-170
3. The auto keyword in C++: https://learn.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-170
4. filter_view Class in C++: https://learn.microsoft.com/en-us/cpp/standard-library/filter-view-class?view=msvc-170
5. Ranges in C++: https://learn.microsoft.com/en-us/cpp/standard-library/ranges?view=msvc-170
6. std::transform - cplusplus.com: https://cplusplus.com/reference/algorithm/transform/

References (cont'd)

7. C++ `std::transform` - cplusplus.com: `https://cplusplus.com/reference/algorithm/transform/`
8. `std::transform` - Microsoft Docs: `https://learn.microsoft.com/en-us/cpp/standard-library/algorithm-functions?view=msvc-170#transform`
9. C++ `std::accumulate` - cppreference.com: `https://en.cppreference.com/w/cpp/algorithm/accumulate`
10. What is the `std::accumulate` function in C++ - educative.io: `https://www.educative.io/answers/what-is-the-stdaccumulate-function-in-cpp`