Higher Order Procedures in C++

Introduction

What are Higher Order Procedures, why bother?

Higher Order Procedures (HOPs) are procedures that take other functions as arguments or return a function as a result. HOPs enable a programmer to reuse and abstract code to improve code readability and modularity. HOPs are at the core of the functional programming paradigm, which uses functions to build and maintain code. HOPs also create an opportunity for the programmer to abstract away from the fine-grained implementation of a function and focus on a high-level approach to solving a problem. Some examples of common HOPs are - Lambda functions, Map, Filter, and Fold.

What is so special about C++?

C++ is a very powerful multi-paradigm programming language that supports Object Oriented Programming, Generic Programming, and Functional Programming styles. While C++ does not have built-in support for HOPs, it provides programmers with features like lambda functions and function pointers which can be used for their implementation. The C++ Standard Library also provides developers with functions that are equivalent to a HOP as discussed below. C++ enables developers to directly control hardware resources which might be useful in applications where functional programming languages are used, especially if those applications are memory-intensive or time-sensitive.

Let us explore the functional programming paradigm using C++ below.

Lambda Functions

The term Lambda Function stems from Lambda Expressions used in Lambda Calculus. Lambda Functions are a way to create anonymous or name-less procedures. They are often used to improve code readability; save the developer time, space, and effort when a small function is to be used and doesn't require a separate definition. They also provide a minimalistic way to create closures, an extremely powerful tool in functional programming.

Lambda in C++

In C++, Lambda Functions are defined as follows:

```
auto multiply = [](int a, int b) -> int {return a*b;};
```

The `[]` is used to denote the beginning of the lambda function. The `(int a, int b)` is a parameter list for a lambda function and `-> int` specifies the return type of the lambda function. The body of the lambda function is enclosed in the curly braces, in this case `{return a*b;}` and the `auto` keyword is used to automatically deduce the type of the lambda function.

Let us look at some examples that use lambda functions in C++.

Lambda Example 1

```
// Program to add two numbers using a lambda function
int main()
{
    int x = 10;
    int y = 12;

    auto add = [](int a, int b) -> int {
        return a + b;
    };

    std::cout << "The sum of " << x << " and " << y << " is " << add(x, y) << std::endl;
    return 0;
}
```

The above program adds two integers 10 and 12 using a lambda function. The lambda function is returned into the add variable. `auto` is used for automatically detecting and assigning the type of a variable in C++.

Lambda Example 2

```
// Program to print a vector using a lambda function
int main()
{
    std::vector<int> my_vector = {1, 2, 3, 4, 5};

    auto print = [](const std::vector<int>& vec)
    {
        for(const auto& number : vec){
            std::cout << number << std::endl;
        }
    };

    print(my_vector);
    return 0;
}
```

The above program is used to print a vector in C++. It uses a lambda function returned to the `print` variable of `auto` type.

Map

Map is a higher-order procedure that applies the given procedure on every item in the given collection and returns a new collection of the same size where the new element is the result of applying that procedure to the corresponding element in the original list.

`std::transform` in C++

Declarations:

- Unary Operations

```
transform(Iterator inputBegin, Iterator inputEnd,
         Iterator OutputBegin,
         unary_operation)
```

- Binary Operations

```
transform(Iterator inputBegin1, Iterator inputEnd1,
         Iterator inputBegin2,
         Iterator OutputBegin,
         binary_operation)
```

Parameters:

- **inputBegin, inputBegin1, inputBegin2** - Input iterator pointing to the first position of a collection.
- **inputEnd, inputEnd1** - Input iterator pointing to the last position of a collection.
- **outputBegin** - Output operator pointing to the first position of the collection of results.
- **binary_operation, unary_operation** - Operations applied to the collection.

Return Value:

Returns an iterator pointing to the end of the transformed range.

Transform Example 1

```
// Program to add elements of two arrays using transform
int main()
{
    int first[] = { 1, 2, 3, 4, 5 };
    int second[] = { 10, 20, 30, 40, 50 };
    int results[5];

    std::transform(first, first + 5, second, results, std::plus<int>());

    for (int i = 0; i < 5; i++)
    {
      std::cout << results[i] << " ";
    }

    return 0;
}
```

The above program adds elements of two arrays using the transform function in C++. The program uses 3 arrays - `first`, `second`, and `results`. It essentially performs `first + second` and stores the result in `results`.

Transform Example 2

```
// Program to capitalize each character in a vector of strings
int main() {
    std::vector<std::string> words = {"please", "maintain", "silence!"};

    std::transform(words.begin(), words.end(), words.begin(),
                   [](std::string& str) {
                      std::transform(str.begin(), str.end(), str.begin(), toupper);
                      return str;
                    });

    std::cout << "Someone yelled: ";
    for (const auto& word : words) {
        std::cout << word << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

The above program uses transform to capitalize each character in a vector of strings. The program uses a lambda function which is applied on each word using transform, and itself applies another transform on each character of every word.

Filter

Filter is a higher-order procedure that operates on a collection of elements and add returns a new collection containing elements from the original collection that satisfy a given condition.

`std::copy_if` in C++

Declaration:

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
   OutputIterator result, UnaryPredicate pred);
```

Parameters:

- **first** − Input iterators to the initial positions of the searched sequence.
- **last** − Input iterators to the final positions of the searched sequence.
- **result** − Output iterator to the initial position in the new sequence.
- **pred** − Unary predicate which takes an argument and returns a bool value.

Return Value:

Returns an iterator pointing to the element that follows the last element written in the result sequence.

Copy If Example 1

```
// Program to filter out even numbers using copy_if

int main() {
    std::vector<int> numbers = {10, 11, 12, 13, 14, 15, 16};

    std::vector<int> filtered_numbers;
    std::copy_if(numbers.begin(), numbers.end(), std::back_inserter(filtered_numbers),
                 [](int i) { return i % 2 == 0; });

    std::cout << "The even numbers are: ";
    for (const auto& num : filtered_numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

The above program uses `copy_if` to filter out even numbers from a vector of integers. It does it by passing the beginning and the ending position of the vector along with a lambda function to filter numbers based on their divisibility by two.

Copy If Example 2

```
// Program to filter courses taught by Anya
int main() {
    std::vector<std::string> words = {"CSCA08", "CSCC24", "CSCC11", "CSCD84"};

    std::vector<std::string> p_words;
    std::copy_if(words.begin(), words.end(), std::back_inserter(p_words),
                 [](const std::string& word) { return word == 'CSCC24' || word ==
"CSCA08"; });
```

```
    std::cout << "Courses taught by Anya: ";
    for (const std::string& word : p_words)
    {
        std::cout << word << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

The above program uses `copy_if` to filter out words according to the given parameters in the lambda function. In this case, these words correspond to the courses at UTSC and the filtered words are courses taught by Anya in the past.

Fold

Fold is a higher-order procedure that takes a binary operator and an identity value and applies that binary operator to a collection of elements in an iterative fashion to get the final result.

`std::accumulate` in C++

Declaration:

```
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );
```

Parameters:

- **first**, **last** – the range of elements to apply the operation on.
- **init** - Identity value.
- **op** - Binary Operation

Return Value:

Returns a value after applying op to all items in the range (first, last) and identity in an iterative fashion.

Accumulate Example 1

```
// Program to calculate the product of numbers using accumulate
int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5};
    int product = std::accumulate(numbers.begin(), numbers.end(), 1, std::multiplies<int>
());
    std::cout << "The product of the numbers is: " << product << std::endl;
    return 0;
}
```

The above program uses `accumulate` to calculate the product of numbers from a vector of integers and store it in the variable product. It leverages the `std::multiplies` function to do the same.

Accumulate Example 2

```
// Program to create a sentence from strings in C++
int main() {
    std::vector<std::string> words{"CSCC24", " is ", "awesome", "!!\n"};
    std::string sentence = std::accumulate(words.begin(), words.end(), std::string(""));
    std::cout << "The sentence is: " << sentence << std::endl;
    return 0;
}
```

The above program uses `accumulate` to concatenate the words given by a vector of strings. It does so by using its default addition function and using the empty string as the identity.

Conclusion

In conclusion, higher-order procedures are a powerful feature in functional programming languages that allow functions to take other functions as arguments or return them as results. Although C++ does not have built-in support for higher-order procedures, it provides features like function pointers, lambda expressions, and templates that can be used to implement similar functionality. We explored some examples of higher-order procedures in C++, including lambda expressions, `std::transform`, `std::copy_if`, and `std::accumulate`, which allow us to write generic algorithms that can be used with different data types and structures, making our code more modular and flexible. Using these techniques, we can write more concise, efficient, and easier-to-maintain code.

References

- "Higher-order function." Wikipedia, Wikimedia Foundation, 27 Mar. 2023, en.wikipedia.org/wiki/Higher-order_function.
- Stroustrup, Bjarne. "The C++ Programming Language." Pearson Education, 2013.
- "Lambda Expressions in C++." Microsoft Docs, Microsoft Corporation, docs.microsoft.com/en

- "Lambda Expressions in C++." Microsoft Docs, Microsoft Corporation, docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-160.
- "std::transform." cppreference.com, cppreference, en.cppreference.com/w/cpp/algorithm/transform.
- "std::copy_if." cppreference.com, cppreference, en.cppreference.com/w/cpp/algorithm/copy_if.
- "std::accumulate." cppreference.com, cppreference, en.cppreference.com/w/cpp/algorithm/accumulate.