C++ Standard Library

Exploring Standard Library to Solve Common Tasks

Kyle Hurd

Example 1

Problem:

Given a list, generate a new list that contains all value squared.

Example:

Input: [1, 2, 3, 4]

Output: [1, 4, 9, 16]

Naive Approach

```cpp
std::vector<int> input{ 1, 2, 3, 4, };
std::vector<int> output;
for (const auto value : input) {
  output.push_back(value * value);
}
```

- Create new `std::vector<int>`.
- Iterate over all values in the `input` array.
- Push back to the `output` vector the `value` squared.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 1, 4, 9, 16 ]
```

Standard Library Approach

```cpp
std::vector<int> input{ 1, 2, 3, 4, };
auto output{ input };
std::transform(input.cbegin(), input.cend(), output.begin(), [](const auto& value) {
  return std::pow(value, 2);
});
```

- Create new `std::vector<int>` using `auto`.
- Utilize `std::transform` to transform all values in the input array.
- Utilize `std::pow` to square each value in the input container.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 1, 4, 9, 16 ]
```

Example 2

Problem:

Given a list, generate a new list that contains all value squared if and only if that value is odd.

Example:

Input: [1, 2, 3, 4]
Output: [1, 2, 9, 4]

Naive Approach

```cpp
std::vector<int> input{ 1, 2, 3, 4, };
std::vector<int> output;
for (const auto value : input) {
  if (value % 2 == 1) {
    output.push_back(value * value);
  }
  else {
    output.push_back(value);
  }
}
```

- Create new `std::vector<int>`.
- Iterate over all values in the `input` array.
- Push back the value squared if it is odd; otherwise just push back the value.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 1, 2, 9, 4 ]
```

Standard Library Approach

```cpp
std::vector<int> input{ 1, 2, 3, 4, };
auto output{ input };
std::transform(input.cbegin(), input.cend(), output.begin(), [](const auto& value) {
  return value % 2 == 1 ? std::pow(value, 2) : value;
});
```

- Create new `std::vector<int>` using `auto`.
- Utilize `std::transform` to transform all values in the input array.
- Utilize `std::pow` to square each value in the input container if it is odd.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 1, 4, 9, 16 ]
```

Example 2

Problem:

Given a list of integers, return the sum of all numbers in the list.

Example:

Input: [1, 2, 3, 4]

Output: 10

Naive Approach

```
std::vector<int> input{ 1, 2, 3, 4, };
int output{};
for (const auto value : input) {
  output += value;
}
```

- Create new `int` for the result.
- Iterate over all values in the `input` array.
- Augment assign the values into the output variable.

```
❯ g++-14 main.cpp
❯ ./a.out
10
```

Standard Library Approach

```cpp
std::vector<int> input{ 1, 2, 3, 4, };
const auto result{ std::accumulate(input.cbegin(), input.cend(), 0) };
```

- Utilize `std::accumulate` to sum all of the values in the input array.

```
❯ g++-14 main.cpp
❯ ./a.out
10
```

Standard For Each

Situation:

We have a method `send_request` which takes a `Request` object and handles sending a request.

```
void send_request(const Request& request);

std::vector<Request> requests = ...;
for (const auto& request : requests) {
  send_request(request);
}
```

Standard For Each

In cases where the function being called matches the container perfectly,
we can **promote reusability** and a functional approach by utilizing `std::for_each`

```
void send_request(const Request& request);

std::vector<Request> requests = ...;
std::for_each(requests.cbegin(), requests.cend(), send_request);
```

Standard For Each - Unique Use Cases

For situations where you have to maintain state, `std::for_each` is particularly powerful.

```
// Function object that counts even numbers
struct CountEven {
    int count = 0;
    void operator()(int n) {
        if (n % 2 == 0) {
            ++count;
        }
    }
};

const auto container = { 1, 2, 3, 4, 5, 6, 7, 8, };
const auto counter{ std::for_each(std::cbegin(container), std::cend(container), CountEven()) };
```

```
❯ g++-14 main.cpp
❯ ./a.out
Number of even elements: 4
```

Standard For Each - Parallel For Loops!

Before `C++17`, to parallelize a for loop, you would have to utilize a tool like `OpenMP`. Although `OpenMP` will most likely outperform the standard library, we have easy access to parallelizing for loops when it is needed for a performance bump.

Parallel For Loops!

Task: Double the Values in a Large Container

```
std::vector<int> container(1'000'000, 1);
for (auto& num : container) {
  num *= 2;
}
```

- Simple and effective. Iterates over all values and multiplies them by 2.
- **Problem**: Container is very large.

Approach 1: OpenMP

```
#pragma omp parallel for
for (int i = 0; i < container.size(); ++i) {
  container.at(i) *= 2;
}
```

- Efficient.
- Can't use range based for loops.

Approach 2: Parallel std::for_each

```
std::for_each(std::execution::par, container.begin(), container.end(), [](auto& num) {
  return num * 2;
});
```

- Also efficient.
- Built into library.
- Can be a one line change.

Example 3:

```
constexpr auto counter_limit{ 500 };
int counter{};

while (counter < counter_limit) {
  if (valid_state) {
    const auto count{ counter };
    counter = 0;
    handle_counts(count);
  }
  ++counter;
  check_a();
  check_b();
  // ...
}
```

- While the counter is below the counter limit, continuously check if the "state" is valid.
- Otherwise, perform checks.
- If valid, capture the counter and reset to 0.

Standard Library Approach

```
while (counter < counter_limit) {
  if (valid_state) {
    handle_counts(std::exchange(counter, 0));
  }
  ++counter;
  check_a();
  check_b();
  // ...
}
```

- `std::exchange` allows you to return the initial value and set the value in one call.

Example 4:

Problem:

Given a list of numbers, correct all of them such that all numbers are within the range 0 - 100

Example:

Input: [-40, 24, 99, 100, 110, 114, 0]
Output: [0, 24, 99, 100, 100, 100, 0]

Naive Approach

```cpp
std::vector<int> numbers{ -40, 24, 99, 100, 110, 114, 0, };
for (auto& number : numbers) {
  if (number < 0) {
    number = 0;
  }
  if (number > 100) {
    number = 100;
  }
}
```

- Iterate over all numbers by reference.
- If the current number exceeds the lower or upper bound, make it the upper or lower bound.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 0, 24, 99, 100, 100, 100, 0 ]
```

Standard Library Approach

```cpp
std::vector<int> numbers{ -40, 24, 99, 100, 110, 114, 0, };
std::for_each(numbers.begin(), numbers.end(), [](auto& value) {
  return std::clamp(value, 0, 100);
});
```

- Convert range based for loop to `std::for_each` loop.
- Create a lambda utilizing `std::clamp`.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 0, 24, 99, 100, 100, 100, 0 ]
```

Example 5

Problem:

Given a list of scores, determine if all scores are between the values 0 and 100.

Example:

Input: [32, 69, 42]
Output: true

Input: [0, 42, 101]
Output: false

Naive Approach

```cpp
std::vector<int> scores{ 32, 69, 42, };
bool result{ true };
for (const auto score : scores) {
  if (score < 0 || score > 100) {
    result = false;
  }
}
```

- Iterate over all scores in container.
- If any score is not between 0 - 100 inclusive, set `result` to false.

```
❯ g++-14 main.cpp
❯ ./a.out
true
```

Standard Library Approach

```cpp
std::vector<int> scores{ 32, 69, 42, };
const auto result = std::all_of(scores.cbegin(), scores.cend(), [](const auto score) {
  return score >= 0 && score <= 100;
});
```

- `std::all_of` to check if all scores in the container are between 0-100 inclusive.

```
❯ g++-14 main.cpp
❯ ./a.out
true
```

Example 5

Problem:

Given a container of numbers, erase all numbers that are even.

Example:

Input: [1, 1, 3, 4, 6, 8]

Output: [1, 1, 3]

Naive Approach

```cpp
std::vector<int> numbers{ 1, 1, 3, 4, 6, 8 };
auto it = numbers.begin();
while (it != numbers.end()) {
  if (*it % 2 == 0) {
    it = numbers.erase(it);
  }
  else {
    ++it;
  }
}
```

- Iterate over all items in the container.
- `std::erase` returns a valid iterator to replace the invalidated one.
- If erase was not called, you are safe to increment the pointer.

```
❯ g++-14 main.cpp
❯ ./a.out
[ 1, 1, 3 ]
```

Standard Library Approach

```
std::vector<int> numbers{ 1, 1, 3,4, 6, 8, };
numbers.erase(std::remove_if(numbers.begin(), numbers.end(), [](const auto number) {
  return number % 2 == 0;
}), numbers.end());
```

- `std::remove_if` moves all items set for removal to the back of the container.
- `std::erase` erases all values between the returned iterator of `std::remove_if` and `numbers.end()`

```
❯ g++-14 main.cpp
❯ ./a.out
[ 1, 1, 3 ]
```

Example 6

Problem:

You have a JSON configuration file that you want to load into your application.

The JSON file consists of many types of values:

```
{
  "name": "Kyle Hurd",
  "age": 25,
  "is_married": false,
  "percent_done": 88.3
}
```

Naive Approach

```
class Config {
public:
  std::string get_value(const std::string& key) {
    return m_data.at(key);
  }
private:
  std::unordered_map<std::string, std::string> m_data;
};

Config config;
const auto age{ std::stoi(config.get_value("age")) };
const auto is_married{ config.get_value("is_married") == "true" };
```

- Store all keys and values as `std::string`

Standard Library Approach

```
class Config {
public:
  template <typename T>
  T get_value(const std::string& key) {
    return std::any_cast<T>(m_data.at(key).second);
  }
private:
  std::unordered_map<std::string, std::any> m_data;
};

Config config;
const auto name{ config.get_value<std::string>("name") };
const auto age{ config.get_value<int>("age") };
const auto is_married{ config.get_value<bool>("is_married") };
```

- Store all values as `std::any`, casting them to their appropriate type when accessing the data.