MTH 4300/4299: Programming and Computer Science II

Lecture 09: Recursion; Iterators and Algorithms

Jaime Abbariao

▒▒ Recursion with references (tail recursion)

In the recursive functions we've written so far, the result is built up as the calls return. For example, in `factorial`, the multiplication happens after the recursive call comes back:

```
return n * factorial(n - 1);  // must wait for factorial(n-1) to return
```

An alternative pattern is tail recursion: instead of waiting for the recursive call to return and then doing more work, you pass the accumulated result forward as a parameter (by reference or by value). The recursive call is the very last thing the function does.

▒▒ Tail recursion with reference parameters

We can use a reference parameter to accumulate the result as we recurse. This avoids building up a chain of pending operations on the call stack.

```cpp
#include <iostream>

void factorial_helper(int n, int& result) {
    if (n <= 0) {
        return;
    }
    result *= n;
    factorial_helper(n - 1, result);
}

int main() {
    int result = 1;
    factorial_helper(5, result);
    std::cout << result << std::endl;  // 120
    return 0;
}
```

Here `result` is passed by reference. Each call multiplies into it directly -- no waiting for return values. The recursive call `factorial_helper(n - 1, result)` is the last thing the function does (this is what makes it "tail recursive").

▒▒ Comparing the two approaches

Standard recursion (builds result on the way back up):

```
factorial(4)
  -> 4 * factorial(3)
       -> 3 * factorial(2)
            -> 2 * factorial(1)
                 -> 1 * factorial(0)
                      returns 1
                 returns 1
            returns 2
       returns 6
  returns 24
```

Tail recursion with reference (builds result on the way down):

```
factorial_helper(4, result=1)
  result = 1 * 4 = 4
  -> factorial_helper(3, result=4)
       result = 4 * 3 = 12
       -> factorial_helper(2, result=12)
            result = 12 * 2 = 24
            -> factorial_helper(1, result=24)
                 result = 24 * 1 = 24
                 -> factorial_helper(0, result=24)
                      returns (base case)
```

The answer is ready as soon as we hit the base case -- no unwinding needed.

▒▒ Exercise: Tail-recursive sum of array

Rewrite `array_sum` using a reference parameter to accumulate the result.

```
void array_sum(int arr[], int index, int size, int& total);
```

Start with `total = 0`, then call `array_sum(arr, 0, 5, total)`.

Test with `{3, 7, 1, 9, 2}`. Expected output: `22`

```cpp
#include <iostream>

void array_sum(int arr[], int index, int size, int& total) {
    if (index >= size) {
        return;
    }
    total += arr[index];
    array_sum(arr, index + 1, size, total);
}

int main() {
    int arr[] = {3, 7, 1, 9, 2};
    int total = 0;
    array_sum(arr, 0, 5, total);
    std::cout << total << std::endl;  // 22
    return 0;
}
```

▒▒ Exercise: Tail-recursive reverse string

Write a function that reverses a string in place using recursion and references.

```
void reverse_string(std::string& s, int left, int right);
```

- Base case: if `left >= right`, return
- Recursive case: swap `s[left]` and `s[right]`, then recurse with `left + 1` and `right - 1`

Test with `"hello"`. Expected output: `olleh`

▒▒ Exercise: Tail-recursive reverse string (Solution)

```cpp
#include <iostream>
#include <string>

void reverse_string(std::string& s, int left, int right) {
    if (left >= right) {
        return;
    }
    char temp = s[left];
    s[left] = s[right];
    s[right] = temp;
    reverse_string(s, left + 1, right - 1);
}

int main() {
    std::string s = "hello";
    reverse_string(s, 0, s.size() - 1);
    std::cout << s << std::endl;  // olleh
    return 0;
}
```

Notice how the reference parameter `std::string& s` lets each recursive call modify the same string. Without the reference, each call would get its own copy and the swaps would be lost.

▒▒ Exercise: Tail-recursive count occurrences

Write a function that counts how many times a value appears in an array, using a reference parameter for the count.

```
void count_occurrences(int arr[], int index, int size, int target, int& count);
```

Test with {1, 3, 2, 3, 5, 3, 7} and target = 3. Expected output: 3

▒▒ Exercise: Tail-recursive count occurrences (Solution)

```cpp
#include <iostream>

void count_occurrences(int arr[], int index, int size, int target, int& count) {
    if (index >= size) {
        return;
    }
    if (arr[index] == target) {
        count++;
    }
    count_occurrences(arr, index + 1, size, target, count);
}

int main() {
    int arr[] = {1, 3, 2, 3, 5, 3, 7};
    int count = 0;
    count_occurrences(arr, 0, 7, 3, count);
    std::cout << count << std::endl;  // 3
    return 0;
}
```

Iterators

So far we've worked with raw arrays and used an index to access elements:

```cpp
int arr[] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
    std::cout << arr[i] << std::endl;
}
```

This works, but it ties us to arrays. What if we want to loop over a different kind of container — like a linked list, or a set? Those don't support index-based access with `[]`.

An iterator is a generalization of an index: an object that points to an element in a container and knows how to move to the next one.

▒▒▒ std::vector

Before working with iterators, we need a container. `std::vector` is the most common container in C++ — think of it as a resizable array.

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {10, 20, 30, 40, 50};

    std::cout << v[0] << std::endl;  // 10
    std::cout << v.size() << std::endl;  // 5

    v.push_back(60);  // add to the end
    std::cout << v.size() << std::endl;  // 6

    return 0;
}
```

You can use `[]` just like an array, but `std::vector` also gives you iterators.

▒▒ Iterator basics

Every standard container provides two iterators:

- `v.begin()` — points to the first element
- `v.end()` — points one past the last element (a sentinel, not a valid element)

Given an iterator `it`, you can:

- `*it` — dereference it to get the value it points to
- `++it` — advance it to the next element
- `it != v.end()` — check if you've reached the end

```
std::vector<int> v = {10, 20, 30, 40, 50};
std::vector<int>::iterator it = v.begin();

std::cout << *it << std::endl;  // 10
++it;
std::cout << *it << std::endl;  // 20
```

▒▒ Looping with iterators

We can rewrite our index-based loop using iterators:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {10, 20, 30, 40, 50};

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

The type `std::vector<int>::iterator` is verbose. We can use `auto` to let the compiler infer it:

```
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << std::endl;
}
```

▒▒ Range-based for loop

The range-based `for` loop you've already seen is just shorthand for iterator-based looping:

```
for (int x : v) {
    std::cout << x << std::endl;
}
```

This is exactly equivalent to the iterator loop on the previous slide. The compiler rewrites it as a loop from `begin()` to `end()` under the hood.

Use the range-based `for` when you just need to read each element. Use iterators directly when you need more control — for example, when you need to know where you are, or when you want to erase or insert elements while iterating.

▒▒ Modifying elements through an iterator

Since `*it` gives you a reference to the element, you can assign through it:

```cpp
#include <iostream>
#include <vector>

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

    for (auto it = v.begin(); it != v.end(); ++it) {
        *it = *it * 2;  // double each element in place
    }

    for (int x : v) {
        std::cout << x << " ";  // 2 4 6 8 10
    }
    std::cout << std::endl;

    return 0;
}
```

If you only need to read (not modify), use a `const` iterator:

```
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    std::cout << *it << " ";
}
```

▒▒ Exercise: sum with iterators

Write a function `int vector_sum(std::vector<int>& v)` that uses an iterator-based loop (not a range-based `for`) to sum all elements.

Test with `{3, 7, 1, 9, 2}`. Expected output: `22`

```cpp
#include <iostream>
#include <vector>

int vector_sum(std::vector<int>& v) {
    int total = 0;
    for (auto it = v.begin(); it != v.end(); ++it) {
        total += *it;
    }
    return total;
}

int main() {
    std::vector<int> v = {3, 7, 1, 9, 2};
    std::cout << vector_sum(v) << std::endl;  // 22
    return 0;
}
```

▒▒ Algorithms

The <algorithm> header provides a large set of ready-made functions that operate on ranges defined by iterators. Instead of writing the same loops over and over, you can call a standard function that is well-tested and clearly communicates your intent.

```
#include <algorithm>
```

We'll look at the most common ones: std::find, std::count, std::min_element, std::max_element, std::reverse, and std::sort.

▒▒▒ std::find

`std::find` searches for the first occurrence of a value and returns an iterator to it. If not found, it returns `end()`.

By hand:

```
int index = -1;
for (int i = 0; i < v.size(); i++) {
    if (v[i] == target) {
        index = i;
        break;
    }
}
```

With `std::find`:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {10, 20, 30, 40, 50};

    auto it = std::find(v.begin(), v.end(), 30);

    if (it != v.end()) {
        std::cout << "Found: " << *it << std::endl;  // Found: 30
    } else {
        std::cout << "Not found" << std::endl;
    }

    return 0;
}
```

▒▒▒ std::count

`std::count` counts how many elements in a range equal a given value.

By hand:

```
int total = 0;
for (int i = 0; i < v.size(); i++) {
    if (v[i] == target) {
        total++;
    }
}
```

With `std::count`:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 3, 2, 3, 5, 3, 7};

    int total = std::count(v.begin(), v.end(), 3);
    std::cout << total << std::endl;  // 3

    return 0;
}
```

std::min_element and std::max_element

These return an iterator to the smallest or largest element in a range.

By hand:

```
int max = v[0];
for (int i = 1; i < v.size(); i++) {
    if (v[i] > max) {
        max = v[i];
    }
}
```

With `std::max_element` (and `std::min_element`):

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {3, 7, 1, 9, 2};

    auto max_it = std::max_element(v.begin(), v.end());
    auto min_it = std::min_element(v.begin(), v.end());

    std::cout << "Max: " << *max_it << std::endl;  // Max: 9
    std::cout << "Min: " << *min_it << std::endl;  // Min: 1

    return 0;
}
```

▒▒▒ std::reverse

`std::reverse` reverses the elements of a range in place.

By hand:

```
int left = 0, right = v.size() - 1;
while (left < right) {
    int temp = v[left];
    v[left] = v[right];
    v[right] = temp;
    left++;
    right--;
}
```

With `std::reverse`:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

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

    std::reverse(v.begin(), v.end());

    for (int x : v) {
        std::cout << x << " ";  // 5 4 3 2 1
    }
    std::cout << std::endl;

    return 0;
}
```

▒▒▒ std::sort

`std::sort` sorts the elements of a range in ascending order.

By hand (bubble sort):

```
for (int i = 0; i < v.size() - 1; i++) {
    for (int j = 0; j < v.size() - 1 - i; j++) {
        if (v[j] > v[j + 1]) {
            int temp = v[j];
            v[j] = v[j + 1];
            v[j + 1] = temp;
        }
    }
}
```

With `std::sort`:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {5, 2, 8, 1, 9, 3};

    std::sort(v.begin(), v.end());

    for (int x : v) {
        std::cout << x << " ";  // 1 2 3 5 8 9
    }
    std::cout << std::endl;

    return 0;
}
```

The pattern

Notice that every algorithm follows the same pattern:

```
std::algorithm_name(container.begin(), container.end(), ...);
```

This is why iterators matter: the algorithm doesn't care whether you're working with a `std::vector`, a `std::string`, or any other container. As long as it provides `begin()` and `end()`, the algorithm works.

For example, `std::reverse` works just as well on a `std::string`:

```
std::string s = "hello";
std::reverse(s.begin(), s.end());
std::cout << s << std::endl;  // olleh
```

▒▒▒ Exercise: putting it together

Given the vector `{5, 2, 8, 2, 9, 3, 2, 7}`, use functions from `<algorithm>` to:

1. Count how many times `2` appears
2. Find the maximum element
3. Sort the vector
4. Print the sorted vector in reverse

Expected output:

```
Count of 2: 3
Max: 9
Sorted:    2 2 2 3 5 7 8 9
Reversed:  9 8 7 5 3 2 2 2
```

▒▒ Exercise: putting it together (Solution)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {5, 2, 8, 2, 9, 3, 2, 7};

    std::cout << "Count of 2: " << std::count(v.begin(), v.end(), 2) << std::endl;
    std::cout << "Max: " << *std::max_element(v.begin(), v.end()) << std::endl;

    std::sort(v.begin(), v.end());
    std::cout << "Sorted:   ";
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;

    std::reverse(v.begin(), v.end());
    std::cout << "Reversed: ";
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;

    return 0;
}
```

▒▒ Lab: Mini Cipher

A secret message has been encoded in two steps:

1. Each character was shifted forward by 3 (e.g. `'h'` → `'k'`, `'e'` → `'h'`)
2. The entire message was reversed

Here is the encoded message:

```
std::vector<char> message = {'v', 'v', 'd', 'o', 'f', ' ', 'r', 'o', 'o', 'h', 'k'};
```

Decode it by undoing the steps in reverse order:

1. Reverse the vector
2. Shift each character back by 3 using an iterator-based loop
3. Print the decoded message character by character
4. Use `std::find` to locate the first `'o'` in the decoded message and print its position

Expected output:

```
Decoded: hello class
First 'o' at position: 4
```