Scientific Programming with C++

CONTAINERS AND ITERATORS

Learning objectives

- After this lecture and related assignments, you will…
  - be introduced to containers in the C++ standard library
  - know how containers differ in function and usage
  - have an idea of what iterators are and how they are used

Arrays: C-style array example

```
// c-style array
std::string users[5] = { "Steve", "Timothy", "Susan", "Michael", "Karim" };

// 2D array
double dataTable[2][4] = {
    {0.21, 0.66, 0.58, 0.13},
    {-0.5, 15.0, 0.01, 540.3}
};

// 3D array
int numbers[2][5][3] = {
    {
        {1, 2, 3}, {4, 6, 8}, {10, 13, 16}, {19, 23, 27}, {31, 36, 41}
    },
    {
        {-11, -567, -4}, {-15, 0, 1}, {5654, 34, -253}, {0, 0, 0}, {0, -2, 0}
    }
};

std::cout << users[3] << std::endl;
users[3] = "Danny";
std::cout << users[3] << std::endl;
std::cout << dataTable[0][3] << std::endl;
std::cout << numbers[1][2][0] << std::endl;
```

```
Michael
Danny
0.13
5654
```

Containers

- Variables and objects can be stored in C-style arrays
- However, the C++ Standard Library (namespace std) offers a variety of containers for storing objects
  - Implemented as class templates
    - Allows the use of various data types with them
  - Containers provide a clean interface
    - Similar member functions across different containers
- Different types optimized for different tasks
  - Sequence containers
  - Associative containers
  - Unordered associative containers
  - Container adaptors

Sequence containers

- Allow sequential access
- Static contiguous array (std::array)
  - Fixed-size array
- Dynamic contiguous array (std::vector)
  - Resizable array
  - The most convenient container for most applications
- Double-ended queue (std::deque)
  - Fast insertion and deletion at beginning and end
- Singly-linked list (std::forward_list)
- Doubly-linked list (std::list)

Example: sequence containers

```
// standard library array
std::array<std::string, 5> users = { "Steve", "Timothy", "Susan", "Michael", "Karim" };

// note the extra brackets here
std::array<std::array<double, 4>, 2> dataTable = { {
    {0.21, 0.66, 0.58, 0.13},
    {-0.5, 15.0, 0.01, 540.3}
} };

// simpler to write with vectors
std::vector<std::vector<double>> dataTableWithVectors = {
    {0.21, 0.66, 0.58, 0.13},
    {-0.5, 15.0, 0.01, 540.3}
};

std::cout << users[3] << std::endl;
std::cout << dataTable[0][1] << std::endl;
std::cout << dataTableWithVectors[0][1];
```

```
Michael
0.66
0.66
```

Associative containers

- Elements are referenced by their key instead of their position
- Set (std::set)
  - The key of an element is the value of that element
- Map (std::map)
  - The value of an element is mapped to a key
    - Key value and mapped value are separate (compare to set)
- Multiple-key set (std::multiset)
- Multiple-key map (std::multimap)

Example: associative containers

```
// create a set
std::set<char> alphabet;
// insert elements into set
alphabet.insert('A');
alphabet.insert('B');
alphabet.insert({ 'C', 'D' });

// create a map with key-value pairs
std::map<std::string, unsigned int> studentGrades =  { {"Steve", 1}, {"Nieve", 3} };
// insert a new element to the map
studentGrades.insert({ "Michael", 0 });

// print all elements in the set
for (char letter : alphabet) {
    std::cout << letter;
}
std::cout << std::endl;
// print some values of the studentGrades map with corresponding keys
std::cout << studentGrades["Nieve"] << std::endl;
std::cout << studentGrades["Michael"] << std::endl;
```

```
ABCD
3
0
```

Container adaptors

- LIFO stack (stack)
  - Last in, first out: elements are inserted and removed from one end
    - Think of stacking pancakes on a plate
      - The last pancake you pile on the stack is the first you eat
- FIFO queue (queue)
  - First in, first out: elements are inserted in one end and removed from the other
    - Think of a queue of people in the university restaurant
      - The first person to get in the queue is the first to get their food
- Priority queue (priority_queue)
  - Elements are ordered in decreasing magnitude (largest is first)

Example: stack container

```
// let's cook a stack of pancakes
std::stack<std::string> pancakePlate;
pancakePlate.push("first cooked pancake");
pancakePlate.push("second cooked pancake");
pancakePlate.push("third cooked pancake");

// at this point, let's remove the top pancake
pancakePlate.pop();

// then cook two more
pancakePlate.push("fourth cooked pancake");
pancakePlate.push("final cooked pancake");

// now let's eat the pancakes starting from the top
while (pancakePlate.size() > 0) {
    std::cout << "eating " << pancakePlate.top() << std::endl;
    pancakePlate.pop();
}
```

```
eating final cooked pancake
eating fourth cooked pancake
eating second cooked pancake
eating first cooked pancake
```

Example: queue container

```
// let's populate a queue to the school canteen
std::queue<std::string> schoolCanteenQueue;
// Billy arrives first
schoolCanteenQueue.push("Billy");
// then Bob
schoolCanteenQueue.push("Bob");
// and finally Jack
schoolCanteenQueue.push("Jack");

// let them have lunch as the canteen opens
while (schoolCanteenQueue.size() > 0) {
    std::cout << schoolCanteenQueue.front() << " got his lunch" << std::endl;
    schoolCanteenQueue.pop();
}
```

```
Billy got his lunch
Bob got his lunch
Jack got his lunch
```

Iterators

- Object that points to an element in a container
  - Can be used to iteratively access the elements of a container
- Different containers support different iterators (or none)
- Usage (some examples)
  - Functions begin() and end()
    - Note: end() points to one past the last element
  - Operators
    - ++ moves to the next element
    - -- moves to the previous elements
  - Loops
    - Explicit for loop with functions
    - Range-for loop (uses iterators internally, no need to dereference)

Example: iterators

```
std::vector<int> numbers = { 1, 4, 7, -3, -2, 0, 5 };

// iterators act like pointers; dereference to get the underlying value
for (std::vector<int>::iterator iter = numbers.begin(); iter != numbers.end(); ++iter) {
    std::cout << *iter << std::endl;
}

// operators can be used manually too
std::vector<int>::iterator iter;
iter = numbers.end();
std::cout << "Last element: " << *(--iter) << std::endl;
```

```
1
4
7
-3
-2
0
5
Last element: 5
```

Example: range-for loop

```
std::vector<int> numbers = { 1, 4, 7, -3, -2, 0, 5 };

// gets a copy of the element; cannot change original
for (int number : numbers) {
    number = 0;
}
std::cout << numbers[0] << std::endl;

// gets a constant reference the element
for (const int& number : numbers) {
    //number = 0; // modification would throw an error
}

// gets a modifiable reference to the element
for (int& number : numbers) {
    number = 0;
}
std::cout << numbers[0] << std::endl;
```

```
1
0
```

Iterators: Why bother?

- Instead of using iterators, you could just go through a container in a for loop and stop when iteration variable reaches the size of the container
  - This is only effective is the size() function for that container is effective
- Iterators are standardized, performant and give direct access to container elements

Iterator categories

- Input
  - Single-pass iteration, no element is accessed more than once
- Output
  - Single-pass iteration, cannot use the access operator but can use the assignment operator
- Forward
  - Single-pass iteration, properties of both input and output iterators
- Bidirectional
  - Can move in both directions
- Random-access
  - Can access any element, even without moving sequentially
  - Most powerful

Summary

- Select the best container for your problem
  - Makes programming easier and improves program performance
  - If unsure, go with std::vector
- LIFO containers: new elements added to the front/top
  - std::stack
- FIFO containers: new elements added to the back/bottom
  - std::queue
- Iterators
  - An effective but not the only way to access elements in containers
  - A lot of functionality "under the hood" through functions the user calls