C++ 11 and the Standard Library: Containers, Iterators, Algorithms

Comp Sci 1575 Data Structures
Q: Why did C++ decide not to go out with C?  
A: Because C has no class.
1 C++ 11
   uniform initialization
   range-based for
   auto
   nullptr

2 Data Structures Overview

3 STL
   History
   Containers
   Iterators
1 C++ 11
   uniform initialization
   range-based for
   auto
   nullptr

2 Data Structures Overview

3 STL
   History
   Containers
   Iterators
```cpp
#include <iostream>

int main()
{
    int value0 = 5; // C++ 98
    int value1(5); // C++ 98

    int value{5}; // C++ 11
    std::string a{"hello"}; // C++ 11

    int value; // C++ 98
    // value() // looks like a function, error
    int value2{}; // default to 0; C++ 11

    int i = 3.99; // i gets 3; no warning/error
    int k{3.99}; // i gets 3; warning

    return 0;
}
```
#include <iostream>

int main()
{
    // C++ 98
    int arr[] = {1, 2, 3, 4, 5};

    // C++ 11 optional
    int arr[] = {1, 2, 3, 4, 5};
    return 0;
}

This is consistent with non-arrays now.
1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr

2. Data Structures Overview

3. STL
   - History
   - Containers
   - Iterators
#include <iostream>

int main()
{
    int fibonacci[] = {0, 1, 1, 2, 3, 5, 8};
    for (int number : fibonacci)
        std::cout << number << ' , ';
    return 0;
}
range-based for loops: modify element

```cpp
#include <iostream>

int main()
{
    int fib_plus[] = {0, 1, 1, 2, 3, 5, 8};

    for (int &number : fib_plus)
        number++;

    for (int number : fib_plus)
        std::cout << number << ',';

    return 0;
}
```
Fast and “safe” range-based for loops

```cpp
#include <iostream>

int main()
{
    int fibonacci[] = {0, 1, 1, 2, 3, 5, 8};

    for(const int &number : fibonacci)
        std::cout << number << ',';

    return 0;
}

Avoid copy
```
1. **C++ 11**
   - uniform initialization
   - range-based for
   - `auto`
   - `nullptr`

2. **Data Structures Overview**

3. **STL**
   - History
   - Containers
   - Iterators
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto d = 5.0; // 5.0 is a double literal
    auto i = 1 + 2; // evaluates to an integer
    auto sum = add(5, 6); // add() returns int
    return 0;
}
#include <iostream>

int main() {
    auto fibonacci[] = {0, 1, 1, 2, 3, 5, 8};

    for(auto number : fibonacci)
        std::cout << number << ' , ';

    return 0;
}

**Warning:** For-each doesn’t work with pointers to an array, and thus with arrays passed to functions.
1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr

2. Data Structures Overview

3. STL
   - History
   - Containers
   - Iterators
nullptr should generally be used instead of NULL

NULL is a “manifest constant” (a #define of C) that’s actually an integer that can be assigned to a pointer because of an implicit conversion.

nullptr is a keyword representing a value of self-defined type, that can convert into a pointer, but not into integers.

```cpp
#include <iostream>

int main() {
    int i = NULL; // OK
    // int i = nullptr; // Error, no convert to int
    int* p = NULL; // ok, int converted into ptr
    int* p = nullptr; // ok
    // suppose you have two functions in overload:
    void func(int x);
    void func(int* x);
}
```

Now, if you call func(NULL), you are actually calling the int variant, NULL being an int. But func(nullptr) will call the pointer variant, nullptr not being an int.
1 C++ 11
   uniform initialization
   range-based for
   auto
   nullptr

2 Data Structures Overview

3 STL
   History
   Containers
   Iterators
## Data structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Worst</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>Search</td>
</tr>
<tr>
<td>Array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Stack</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Queue</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Singly-Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Doubly-Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Skip List</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Hash Table</td>
<td>N/A</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Cartesian Tree</td>
<td>N/A</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>B-Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Red-Black Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Splay Tree</td>
<td>N/A</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>KD Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
</tbody>
</table>

**Color key:**
- Red: $O(\log(n))$
- Yellow: $O(n)$
- Green: $O(n \log(n))$
## Sorting algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Timsort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\Omega(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Tree Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n(\log(n))^2)$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$\Omega(nk)$</td>
<td>$\Theta(nk)$</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
</tr>
<tr>
<td>Cubesort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
</tbody>
</table>
1 C++ 11
   uniform initialization
   range-based for
   auto
   nullptr

2 Data Structures Overview

3 STL
   History
   Containers
   Iterators
1 C++ 11
   uniform initialization
   range-based for
   auto
   nullptr

2 Data Structures Overview

3 STL
   History
   Containers
   Iterators
STL is now just part of standard namespace

- Old diagram of the Standard Template Library (STL)
Main pillars of the STL

- **Containers** manage storage space for elements and provide member functions to access them. Implemented as templated classes, with flexibility of types as elements.
- **Algorithms** act on containers, and perform operations like initialization, sorting, searching, and transforming of the contents of the aforementioned containers.
- **Iterators** step through elements of collections of objects in containers or subsets of containers. Pointer-like iterators can traverse many container classes in modularly.
Check out the code: container with iterators and algorithms

A dynamic C-like array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object.

- Random access is done in constant, $O(1)$ time.
- Insertion or removal of elements at the back takes average time $O(1)$, amortized constant time. Removing the last element takes only constant time, because no re-sizing happens.
- Insertion or removal of elements at the beginning or “middle” is linear in distance to the end of the vector $O(n)$.

See: Intro_vector.cpp
1. C++ 11
   uniform initialization
   range-based for
   auto
   nullptr

2. Data Structures Overview

3. STL
   History
   Containers
   Iterators
Containers

- Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queues, lists, and stacks.
- Classes of containers, each of which is designed to support a different set of operations:
  1. sequence containers
  2. associative containers
  3. un-ordered associative containers
  4. container adaptors (modify above)
- Containers manage storage space that is allocated for their elements and provides member functions to access them, either directly or through iterators (objects with properties similar to pointers).
- The set of containers have at least several member functions in common with each other, and share functionalities.
For a comprehensive list, see:

Simple utilities: pair, tuple

Not in the “Containers” but “Utilities”

How to choose your container?

- Order is important?
  - No: Last In, First Out
  - Yes: Best In, First Out

- Last In, First Out?
  - No: Best In, First Out
  - Yes: Keep Sorted Elements

- Keep Sorted Elements?
  - No: Insert/erase at middle
  - Yes: Insert/erase at front

- Insert/erase at front?
  - Yes: Frequent Traversals
  - No: Persistent Positions

- Persistent Positions?
  - Yes: Size varies
  - No: Allow Duplicates

- Allow Duplicates?
  - No: Separate Key / Value?
    - Yes: unordered_set
    - No: unordered_map
  - Yes: Separate Key / Value?
    - Yes: unordered_multiset
    - No: unordered_multimap

- Size varies?
  - Yes: List
  - No: Vector (sorted)

- Vector (sorted)?
  - No: Deque
  - Yes: Main Purpose?
    - In-order Traversals?
      - Yes: vector (sorted)
      - No: Look-up Keys
    - Look-up Keys?
      - Yes: vector
      - No: deq
1. C++ 11
   - uniform initialization
   - range-based for
   - auto
   - nullptr

2. Data Structures Overview

3. STL
   - History
   - Containers
   - Iterators
An iterator can be imagined as a pointer to a given element in the container, with overloaded operators to provide a subset of well-defined functions normally provided by pointers:

- **Operator** `*` Dereferencing the iterator returns the element that the iterator is currently pointing at.
- **Operator** `+ +` Moves the iterator to the next element in the container. Most iterators also provide **Operator** `− −` to move to the previous element.
- **Operator** `==` and **Operator** `!=` Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
- **Operator** `=` Assign the iterator to a new position (typically the start or end of the container’s elements). To assign the value of the element the iterator is point at, dereference the iterator first, then use the assign operator.
- Where does forward_list go?
### Iterators

Each category of iterator is defined by the operations that can be performed on it. Any type that supports the necessary operations can be used as an iterator, e.g., pointers support all of the operations required by RandomAccessIterator, so pointers can be used anywhere a RandomAccessIterator is expected.

<table>
<thead>
<tr>
<th>category</th>
<th>properties</th>
<th>valid expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>all categories</td>
<td><em>copy-constructible, copy-assignable and destructible</em></td>
<td>X b(a);</td>
</tr>
<tr>
<td></td>
<td>Can be incremented</td>
<td>b = a;</td>
</tr>
<tr>
<td></td>
<td>Supports equality/inequality comparisons</td>
<td>a == b</td>
</tr>
<tr>
<td></td>
<td>Can be dereferenced as an <em>rvalue</em></td>
<td>a != b</td>
</tr>
<tr>
<td></td>
<td>Supports arithment operators + and -</td>
<td>*a-&gt;m</td>
</tr>
<tr>
<td></td>
<td>Can be dereferenced as an <em>lvalue</em> (only for mutable iterator types)</td>
<td>*a = t</td>
</tr>
<tr>
<td></td>
<td>Can be decremented</td>
<td>*a++ = t</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>default-constructible</td>
<td>X a;</td>
</tr>
<tr>
<td></td>
<td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td>
<td>X()</td>
</tr>
<tr>
<td></td>
<td>Supports arithmetic operators + and -</td>
<td>{ b=a; *a++; *b;</td>
</tr>
<tr>
<td>Random Access</td>
<td>Supports inequality comparisons (&lt;, &gt;, &lt;= and =&gt;) between iterators</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>Supports compound assignment operations += and -=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supports offset dereference operator ([ ])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can be incremented</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can be decremented</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supports inequality comparisons (&lt;, &gt;, &lt;= and =&gt;) between iterators</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supports compound assignment operations += and -=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supports offset dereference operator ([ ])</td>
<td></td>
</tr>
</tbody>
</table>
Each container includes at least 4 member functions for the operator= to set the values of named LHS iterators.

- `begin()` returns an iterator to the first element.
- `end()` returns an iterator one past the last element.
- `cbegin()` returns a const (read-only) iterator to the first element.
- `cend()` returns a const (read-only) iterator one past the last element.

`end()` doesn’t point to the last element in the list. This makes looping easy: iterating over the elements can continue until the iterator reaches `end()`, and then you know you’re done.
```cpp
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<int> ints {1, 2, 4, 8, 16};
    std::vector<std::string> fruits {"orange", "apple", "raspberry"};
    std::vector<char> empty;

    // Sums all integers in the vector ints (if any), printing only the result.
    int sum = 0;

    for (auto it = ints.cbegin(); it != ints.cend(); it++)
        sum += *it;

    std::cout << "Sum of ints: " << sum << "\n";

    // Prints the first fruit in the vector fruits, without checking
    std::cout << "First fruit: " << *fruits.begin() << "\n";

    // checks
    cout << empty.empty();
    if (empty.begin() == empty.end())
        std::cout << "vector 'empty' is indeed empty.\n";

    // Alternative syntax
    auto it1 = ints.begin();
    auto it2 = std::begin(ints);
}
Container have different iterator invalidation rules

Each container has different rules for when an iterator will be invalidated after operations on the container:
http://en.cppreference.com/w/cpp/container

<table>
<thead>
<tr>
<th>Category</th>
<th>Container</th>
<th>After insertion, are...</th>
<th>After erasure, are...</th>
<th>Conditionally</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>iterators valid?</td>
<td>references valid?</td>
<td>iterators valid?</td>
</tr>
<tr>
<td>Sequence containers</td>
<td>array</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vector</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>deque</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>forward_list</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Associative containers</td>
<td>set</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>multiset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>map</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>multimap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unordered associative containers</td>
<td>unordered_set</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>unordered_multiset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unordered_map</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unordered_multimap</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C++ 11
uniform initialization
range-based for
auto
nullptr
Data
Structures
Overview
STL
History
Containers
Iterators