Vector, Iterators & the STL

CS 150 – C++ Programming I
Lecture 16

Overloaded Output Operators

- For any user-defined type you can overload most of the C++ operators to work with that type
- Syntax for a binary operator (+, ==, >, etc)
  - `T operator?(const T& lhs, const T& rhs)`
- Overloaded output operator syntax:
  - `ostream& operator<<(ostream&, const T&)`
  - Return the `ostream` argument after writing to it
- Exercise: write output operator for `Car` type
  - `Manufacturer, model, mpg MPG`

Creating vector Objects

- A C++ standard library list-like container
  - Homogeneous: store things of the same type
  - Not fixed size – grows and shrinks as needed
  - Need to `#include <vector>`
- Specify base type when creating variables
  - `vector<int> v1;` `// empty list of int`
  - `vector<double> v2(10);` `// 10 initialized 0`
  - `int a[] = {1, 2, 3};` `// C++ 98`
    `vector<int> v4{a, a + 3};`
  - `vector<int> v3{1, 2, 3};` `// C++ 11`
- Exercise: initialize some vectors (*vinit.cpp*)

Accessing vector Elements

- Access elements with `at()` or `[ ]` just like `string`
  - `cout << v2.at(0) << endl;  // safe (checked)`
  - `cout << v2[1] << endl;  // not safe`
- Operations on *`vector`* objects
  - Number of elements? `size()`
    - Store size in *`vector::size_type`* or *`size_t`*
  - Add element to end of the *`vector`*: `v1.push_back(3);`
  - Remove last element with `pop_back()`
  - Aggregate: compare and assign using `==` and `=`

Vectors & Functions

```
double average(const vector<double>& v) // 1.
{
    size_t len = v.size();   // 2.
    if (len == 0) return 0.0/0.0; // or nan("")
    double sum = 0.0;
    for (auto e: v) sum += e;  // 3.
    return sum / len;
}
```

1. Pass by reference (or const&) never by value
2. Use size() function, save in size_t
3. Use range-based loops when you can (C++ 11)

Templates & Output

- Imagine a *print()* function for *vector*
  - `void print(ostream& out, const vector<double>& v);`
- Problem? Doesn't work for other *vector* types
  - To work for *vector* of any type use templates
  - 1. Template definition goes in the header file
  - 2. Must fully qualify all library types with `std::`
  -
```
template <typename AnyType>
void print(std::ostream& out,
           const std::vector<AnyType>& v)
```
- Exercise: *vprint.cpp*, *vprint.h*

Templates, Operators & Vector

- *vector* already defines =, ==, relational operators
  - Does not define an output operator <<
- We want it to work for any *vector* type
  -
```
template <typename AnyType>
ostream& operator<<(ostream& out,
                    const vector<AnyType>& v)
```
  - Remember, a template is not a function
  - Instead, it generates a function when called
  - Should be placed in a header file
- Exercise: complete *vecout.cpp*

Common Algorithms

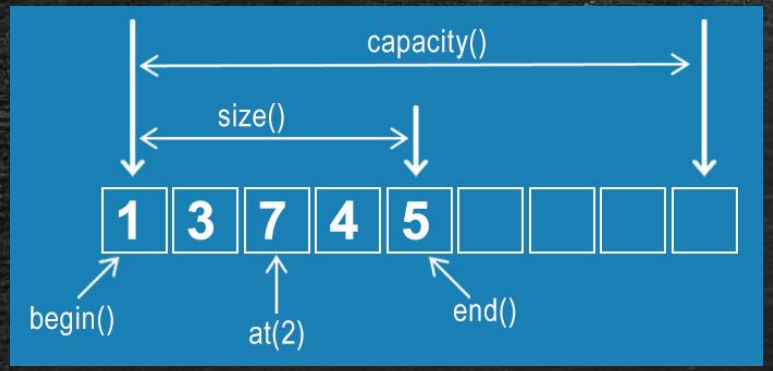
- There are several common algorithms you should memorize
  - Counting, accumulation, extremes, adjacent elements, separators
- Different loops are best for different algorithms
  - Need to visit every element? Use range *for*
  - Keep track of position? Use traditional *for*
  - Move, sort, shuffle? Use traditional *for*
  - Grow or shrink? Use iterator or *while* loop
- Alternatively, use standard library algorithms
  - Part of the STL – Standard Template Library

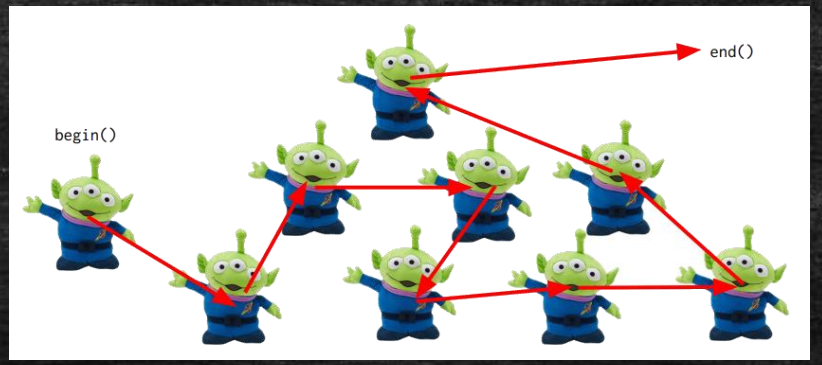
Counting Elements

- To count elements which match a condition:
  - 1. Create a counter for each type you want to count
  - 2. Write a loop that processes every element
  - 3. If the element matches the condition, count it
- Exercise: `divisibleBy(v, n)`
  - Use range `for` loop
- The standard library includes this algorithm
  - Part of the Standard Template Library (STL)
  - Containers: `vector`, `deque`, `list`, `map`, `set`, etc.
  - Iterators: allow you to access any container type

Introducing Iterators

- Different containers store data in different ways

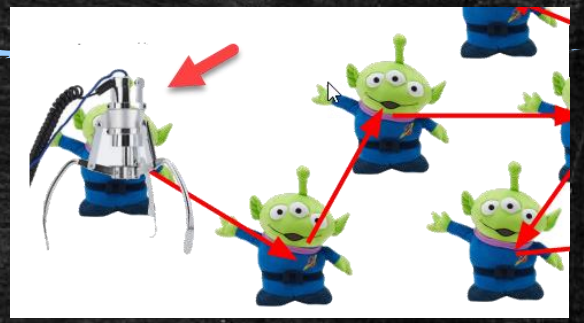


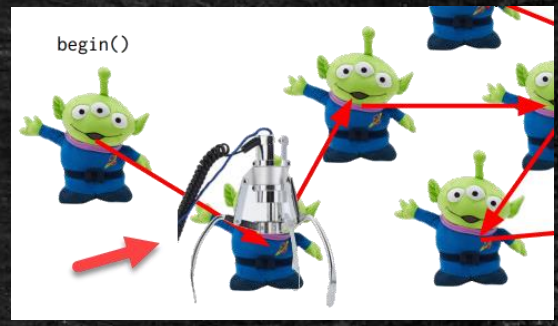


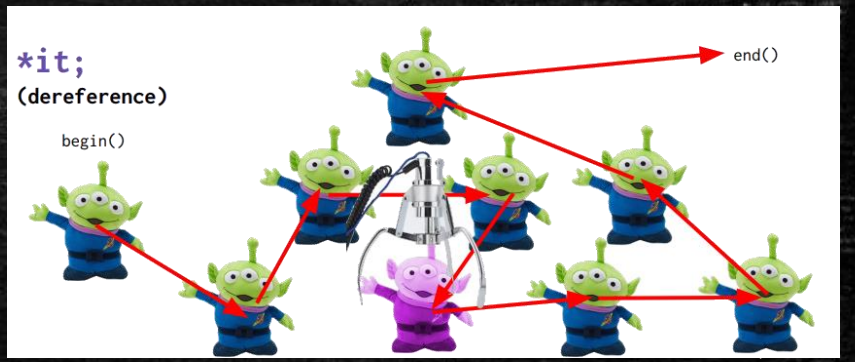
  - Both *vector* and *list* are sequential containers
  - List elements are not contiguous, thus they are not indexed
- Iterators are objects which ignore physical order
  - Like the "claw" in an arcade

Using begin() and end()

- Move the class to the first element: *begin(c)*
  - *c* is the container object
  - `auto itr = begin(list);`
- Move the claw to the next using *++itr* (prefix)
  - Stops when it reaches `end(c)`
  - "One-past" last element in collection
- "Pick up" element by dereferencing
  - `auto value = *it;`
- Exercise: *divisibleBy* with iterators







Constant Iterators & STL Algorithms

- Iterators permit you to change the items they refer to
  - A `const` iterator (ie. `vector::const_iterator`) does not
  - `begin()` returns a `const` iterator when the container is `const`
  - Starting in C++14 you can use `cbegin()` and `cend()`
  - Use when you don't want the container elements changed
- The STL has several collections of pre-built algorithms
  - Headers: `<algorithm>`, `<iterator>`, `<numeric>`
  - `vector<int> v{...};`
    `int threes = count(cbegin(v), cend(v), 3);`

STL Algorithms & Lambdas

- You may also want to count for a condition
  - Represent a condition by writing a predicate function
    - `bool isEven(int n){return n%2==0;}`
  - Then, pass the name of the function to the algorithm
    - `count_if(begin(c), end(c), isEven);`
- May also use an anonymous function (called a lambda)
  - `count_if(cbegin(c), cend(c), [](int e) { return e % 2 == 0; });`
  - Use a lambda capture to pass additional variables
- Exercise: write `divisibleBy()` using algorithm

Returning vector Objects

- Use *vector* to return a collection of items
  - Suppose you have a *vector* with duplicates
  - Write a function unique() that returns a new *vector* with all of the duplicates removed
- The unique() algorithm
  - Create an empty vector, result
  - Visit each element in input vector
  - If it is not in result, then add it
  - Return result
- Exercise: write *unique()* using loops

Insert and Delete

- *vector* has functions for inserting and erasing
  - The vector shrinks or expands as necessary
- Need to use iterator, not index
  - v.begin() → v[0], v.end()→v[v.size()]
  - v.begin()+1 → v[1], v.end() - 1 → last item
- Other complications?
  - Size of vector changes and iterators invalidated
  - Trying to erase all elements? Oops! http://cpp.sh/8hju2
- Exercise: write modifying version of *unique()*