

Rec13 - STL

Topics

- Collections
- Half-open range
- Iterators
- Algorithms
- Functors and Lambdas
- Function Templates
- Maps and sets

General notes

- Please note, do not *not* use the keyword `auto`, except where specified.
- For every task in your code, please put a comment to identify which task number it is. This will help both you and the TA who looks at your code.
[I am providing a main that has done much of this for you, but it would be good to do the same for any additional code, e.g. functions or classes, that you need to write.]

Tasks

1. Create a `vector` with various values. Print it out using the `ranged for`. (For later exercises, it may simplify your life if there are an even number of elements.)
2. Copy this into a `list`, using the list constructor that takes a half-open range, provided by `begin()` and `end()`. Print this list out.
3. Sort the original vector, then print both it and the list out.
 - Note that sorting the list cannot be done the same way. We will discuss why further down and how to work around that.
4. Write a loop to print every *other* element of the vector, starting with the first element. The `ranged for` won't work for this, instead use an "index-based" for-loop.
5. *Attempt* to do the same with the linked list.
 - It doesn't compile.
 - Why? No `op[]` for list!
 - Why didn't they provide `op[]` for list? Easy to write, right? (In fact you've done it.) They omitted it just so we wouldn't try to

use it in a loop like this. (How did you write it? Each call to `op[]` required walking from the front of the list. To loop over N items this way would take approximately $N^2 / 2$ steps. Ouch!)

Iterators

6. Use iterators to redo task 4, i.e display every *other* element of the vector.
 - Remember how to declare the *type* of an iterator.
 - Note that the iterators we are using here, i.e. the iterators for a list, support: copy constructor, `!=`, `++`, `--`, and dereference.
 - Also iterators for vectors support adding on arbitrary integer values, just as we could with pointer arithmetic. This sort of iterator is known as a **random access iterator**. For this task, use this random access ability of vector's iterators.
7. Repeat the previous task for the list.
 - In the previous task, you bumped the iterator by 2, but that won't work with the list iterator. List iterators do not support random access. Same reason that lists don't support the index operator.
 - So figure a work around in order to repeat the previous task for the list.
8. Also, we have now seen that there is more than one kind of iterator.
 - Lists have *bidirectional* iterators which support moving back and forth, but *only* by using the increment and decrement operators. Vectors have iterators that do all of that but also support an addition operator that takes as a second argument an integer, so that you can do the equivalent of pointer arithmetic with vector iterators. Again, these are known as random access iterators.
 - As you know, most sort algorithms require random access. Because of that, we cannot use the algorithm `sort` with the iterators for a list. Of course, lists can be sorted (hopefully you sorted lists in your data structures course) and the list class has its own sort *method*. The method does not take any arguments, its purpose being to sort the entire list.
 - Sort your list with the list class's sort method. And of course show that it worked.
9. Write a function that is passed in a list of ints and prints the list using iterators.
[Naturally, it should be passed a constant reference to the list.]

- First try using `list<int>::iterator` as before and observe that you get a compilation error.
 - Try again using `list<int>::const_iterator`.
 - It works!
 - But what a pain!
10. Repeat the previous task, using a ranged for.
Happily, this works just as before. No complication with `const`.

Auto

11. So, using ranged-for is great where we can get away with it. But when we can't, it can be annoying having to specify those messy types.
- In comes `auto`!!!
 - Let's write a function that will print every other item in the list. Instead of worrying about how to specify the type for the iterator, just use the keyword `auto`. The code is easier to write and actually easier to read, while still being statically typed. Yes, it is statically typed, but the compiler is doing the work using *type inference*
12. Write a function to find an item in the list, returning an iterator for where it is.
- The function will take a list, again passed by `const` reference, and an `int` to search for.
 - Do not use `auto`.
 - What did you have to use for the return type?
 - What should / did you return when the search failed?
 - Be sure to test the function with an item that is there and an item that is not there.
13. Ok, modify the for loop to use `auto`. That works, great.
- What about the return type. That still has the ugly `const_iterator`. Try to use an `auto` there. If your compiler supports not just `c++11`, but also `c++14`, then it will work.

Generic Algorithms

14. Use the algorithm `find` to do the same as the previous step.
- Note the need for the include `<algorithm>` (actually, we already included it to use `sort`).
 - Remember that the generic `find` is *not* passed a collection, instead it is passed a “half-open range”.

15. Use the algorithm `find_if` to locate the first even number in the vector and in the list and print them out.
- Note that you first have to define a *function* that takes an `int` and returns a `bool` indicating whether the integer was even or not.
 - When you call `find_if`, you will pass in the usual half-open range defining where to search, followed by the function. How do you pass a function? Just pass its name. No, don't imagine that the *code* for the function is being passed in, the name just specifies what function we are calling.
 - Of course, the vector / list that you are searching may not have any even numbers, so the code that checks the return value of `find_if` must be prepared for such a condition.
16. Let's use a **functor** for the problem of finding the first even number. A functor is an instance of a class that has implemented the "parenthesis operator", aka the *function call operator*.
- Define a class, call it whatever you want.
 - Define a member function for the function call operator. What will its name be? Naturally the word `operator` followed by the characters used for the operator. What are they in this a case? Yes, a pair of parentheses. So, despite how odd it may look, the name is `operator()`.
 - The parameters? Whatever parameters your function takes. Yes, you can define versions of the operator that take whatever and however many parameters you like. Not only that but the function call operator can be overloaded within your class.
17. Let's use a **lambda expression** for the problem of finding the first even number.
- A lambda expression is an "unnamed function". It has a bit of an odd syntax. The simplest example would be:

```
[ ] { cout << "Hello Lambda!"; }
```
 - You could actually run that. Just put it on a line in main. And put a pair of parentheses after it, because after all if that's a function, and we still have to *call* it. So main could look like:

```
int main() {  
    [ ] { cout << "Hello Lambda!"; }();  
}
```
 - So, what have we learned? The syntax for defining a lambda expression requires a pair of square brackets. Inside the

square brackets there may be specified a list of variables that would be "captured". That does not affect us right now, so I will not explain any more about it.

- But what we haven't seen that we do need is how to provide a parameter list. That's because the the function we wrote didn't need one. And the point of lambda expressions is to minimize what you need to write. So if you don't need any parameters, then you didn't even need a parameter list! If you do need parameters, then the parameter list would go right after the square brackets. I'll show you the use of a lambda expression taking two ints and printing the results. Try putting the following line in your program. It should print 9.

- ```
 [] (int a, int b) { cout << a + b << endl; }
 (4, 5);
```

- What if we want to return the sum instead of printing it?

- ```
    int result = [] (int a, int b) { return a + b;  
    } (4, 5);
```
- ```
 cout << "the result is: " << result << endl;
```

- Note that something else seems to be missing? The return type was never stated! Again, C++ is trying hard to learn how to reduce how much you have to type. It is clear to the compiler that this lambda expression is returning an int so it doesn't require that you say so.

- If you did want to say what the return type for that lambda expression was supposed to be, then the above code would instead look like: 

```
 int result2 = [] (int a, int b)
 -> int { return a + b; } (4, 5);
```

- ```
    cout << "the result is: " << result2 << endl;
```

- Yes, the return type shows up just before the body, after the optional parameter list. With the arrow which has nothing to do with a pointer. Wierd, but you don't usually need to mention the return type. Lambda expressions are usually short and clear.

18. Define a local array, the same size as the list and the vector.

- You can do this with the line:

```
int arr[8]; // if there are eight items in the  
vector and the list.
```

- Use the **copy** algorithm to fill it either from the list or the vector.

- Copy is passed the half-open range that it is copying *from* and the location of the first item it is copying *to*.
 - Again, print its contents.
 - Use find on the array to do the same tests we did before.
- What do we have to pass in to specify the half-open range?

Templated functions

19. Implement your own version of the find algorithm (call it ourFind) for lists of ints. This will be a *function*, **not** a function *template*.
 - Put a print statement in your function so that we know it is yours
 - show that it works, using the same tests you used with the library version.
20. Rewrite ourFind as a function template. You do not have to comment out the function you wrote in the previous task.
 - In the template, change the print line to say you are in the template
 - Test your template using the vector.
 - Run your test again on the list. Notice that C++ used your function, not the template.

That's a good thing!

Collections

21. Read a file, listing all of items, but list each distinct item only once. A text file is attached to this recitation for this purpose.
 - Solve this using a vector of strings to track what you have seen.
 - For each word that you read from the file check if it is in the vector and if it isn't then add it to the vector.
 - After filling the vector, display its size and contents.
 - Running this on the attached file, you will likely notice your program hesitate a moment while it fills the vector.
22. Repeat the previous task but instead of using a **vector** to hold the strings, use a **set** to do so.
 - You will have to include the set header file, i.e. `#include <set>`
 - You define a set the same way as you define a vector, simple type set and then place the kind of things the set should hold in angle brackets.



- You *could* test if an item is in a set, use the set method `find`. If the item is not found, `find` will return the same value as `set's end()` method. (Note this is not the generic `find` algorithm. Why not?)
- However, to add an item to a set, just use the set method called `insert`. If the item is already in the set, then nothing will change, so you don't actually need to first test if it is there.
- You will see that this version finishes much faster. *And* the elements will be displayed in sorted order without your having to write any code to do handle the sorting! (How do you think that was done?)

23. Finally, we will repeat a problem from early in the semester where we tracked word positions from a document. (By position, we mean whether it is the first token or the second or...) Remember how we created a struct to hold the token and a vector of positions. Then maintained a vector of these struct instances? This time we will use a `map`, much like a dictionary in Python, to map from the tokens to their vectors. The goal is to print out each word in the document, one per line, together with the token positions for that word.

- First we need to define our map variable. (Of course, we have to have included the map header file, just as we did with the `set`.)
 - The map requires two types, not just one. We have to say what we are mapping from, here a string, and what we are mapping to, here a vector of ints. So we could define a variable `wordMap`, as follows:
`map<string, vector<int>> wordMap;`
- We can access each element of the map, using the square bracket operator. `wordMap["fred"]` would return the value, a vector of ints, for fred's entry in the map. And if fred hasn't been entered yet, it automatically creates an entry with a value that is "default initialized". In the case of our map, since the value is a vector, it will be initialized to an empty vector.
- All we need to do is `push_back` onto that vector the new position. The line: `wordMap["fred"].push_back(17)` would add the position 17 to fred's vector. And if fred wasn't there before, it will now be there with a vector holding just the single value 17. Almost like magic!

- Finally we need to be able to loop over the map. Using the `ranged for`, it is very easy. Each item in the map is something called a `pair`. A pair has two fields `first` and `second`. In a map, the field `first` for each entry is the "key" (the word) and the field `second` is the "value" (the vector of positions). To keep your life simple, use the keyword `auto` to specify the type for the `ranged-for`.
- For each item in the vector, first print out the word and then print out all of the positions. It would look best to do this all on one line for each word.
- Again, notice how quickly this runs and that the words appear in sorted order. And it was a lot easier to write than what we did in class!

Additional resources for assignment

-  [pooh-nopunc.txt](#) (121 KB; Jan 2, 2018 2:39 pm)
-  [rec13_test.cpp](#) (3 KB; Jan 2, 2018 2:39 pm)