Heterogenous Lookup in the STL:
We Don't Need Probe Objects!

David Kieras, EECS Department, University of Michigan
February 18, 2017

Introduction

A common use of containers is to store objects that are then looked up, or searched for, either with algorithms or member functions of the container class. This tutorial is concerned with searching the `vector<>` and `set<>` containers, where the usual descriptions and examples of how to search them assume that you specify the object you are looking for with a *probe* object that has the same type as the objects in the container. This same-type approach is *homogenous* lookup.

If the container holds simple objects such as ints or strings, constructing a probe object is no problem; but for more complex objects, we are often in a situation where we could in principle identify an object by one of its properties, such as a member variable value, and it would be more efficient and convenient if we could search for an object with this value rather than construct a probe object just to hold this value.[1] The popular presentations of the STL do not deal with this problem.

It turns out[2] that there is, in fact, an alternative in the fine print of even the C++98 Standard for the `lower_bound` algorithm, and a pair of remarkably unpublicized alternatives in the C++14 Standard for the `set<>::find()` member function. These alternatives allow you to search a container using a probe that has a *different type* than the objects in the container; this is called *heterogenous* lookup.

The following `Thing` class is used in the examples in this tutorial. It has a `string` name, an `int` id number automatically assigned, an accessor for the name, and an output operator. Note that `operator<` is defined in terms of `Thing`'s `std::string` name, allowing us to easily order `Thing`s by name:

```
class Thing {
public:
   Thing(const string& name_) : name(name_), id(++count) {}
   const string& get_name() const
        {return name;}
   bool operator< (const Thing& rhs) const
        {return name < rhs.name;}
   friend ostream& operator<< (ostream& os, const Thing& t);
private:
   string name;
   int id;
   static int count; // used to give each Thing a unique ID number
};

ostream& operator<< (ostream& os, const Thing& t)
{
   os << "Thing " << t.id << ' ' << t.name;
   return os;
}

int Thing::count = 0;
```

Preview: In the examples shown below, we will have a container of `Thing` objects which we will search. In the traditional homogenous probe object approach, the search will be done for a matching `Thing` object; in the heterogenous lookup approach, the search will be done using only a `string` variable containing the name of the desired `Thing` object.

---

[1] The Project 1 `Ordered_container` supported both homogenous and heterogenous lookup with its different `find` functions.

[2] Thanks are due to an enterprising student who noticed the new C++14 `set<>` feature in some online discussions, and followed up on it and then `lower_bound`, and then told me about it. I had never seen these alternatives mentioned in the popular presentations.

Heterogenous lookup with `lower_bound()` and `vector<>`

The `lower_bound` algorithm does binary search on a `vector` container whose contents are already sorted, and returns an iterator pointing to the object furthest in the container that does not compare less than the probe. For example, using a `vector<Thing>` as the container, sorted in order, the traditional probe object approach would use `lower_bound` as follows:

```
vector<Thing> things = {Thing("Dick"), Thing("Harry"), Thing("Tom")}; // in order

while(true) {
    cout << "Enter name: ";
    string str;
    cin >> str;
    auto it = lower_bound(things.begin(), things.end(), Thing(str));
    if(it != things.end() && it->get_name() == str)
        cout << "found: " << *it << endl;
    else
        cout << "not found" << endl;
    }
```

Now there is a second form of `lower_bound` in which you supply a comparison function, declared as:

```
template<typename IT, typename T, typename Comparison>
IT lower_bound(IT first, IT last, const T& value, Comparison comp);
```

Type `IT` is the iterator type, `T` is normally the type of the objects in the vector; the probe is the supplied `value` argument which has type `T`. The `comp` argument is a function (or function object) that takes two parameters and returns a `bool` true value if the first argument comes before the second. Usually, `comp` takes arguments that have type `const T&`, so the function compares two objects whose types are the type of objects in the container. The usual story about this version of `lower_bound` is that it allows you to provide an ordering function that is different from the default use of the object's `operator<`.[3]

Here is where the fine print in the Standard comes in. The requirements for the `Comparison` function are:

- The first argument must match the type produced by dereferencing an iterator, which is the type of objects in the container.
- The second argument must match the type `T` shown in the above template declaration; this can be either the same type as the container objects, or it can be a *different type.*
- The function returns `true` if the first argument comes before the second argument in a way consistent with the container ordering.

In the traditional use of `lower_bound`, the comparison function has the same types for the first and second argument; in our homogenous lookup example above, this is a `Thing-Thing` comparison. Because the second argument can be a different type than the first, as long as the implied ordering is consistent, we can use `lower_bound` for heterogenous lookup.

Suppose we want to look up objects using just a `string` containing the `Thing`'s name, without creating a probe `Thing` object. Let's fill the container using the default < ordering. Then we could search the container with a `string` probe by using the following comparison function:

```
bool compare_Thing_to_string(const Thing& t, const string& s)
    {return t.get_name() < s;}
```

This compares an object in the container with an object of a different type, namely a `std::string` object, but the ordering implied by this comparison is the same as comparing two `Thing`s by `Thing::operator<`.

To do heterogenous lookup, we use the above custom `Thing-string` comparison function, and we don't need the probe object:

[3] This version and description of `lower_bound` is part of the C++98 Standard and the C++14 (draft) Standard section 25.4.3.1.

```
while(true) {
    cout << "Enter name: ";
    string str;
    cin >> str;
    auto it = lower_bound(things.begin(), things.end(), str, compare_Thing_to_string);
    if(it != things.end() && it->get_name() == str)
        cout << "found: " << *it << endl;
    else
        cout << "not found" << endl;
    }
```

No probe object needed! Isn't that amazing?[4]

Heterogenous lookup with the `set<>` container

Heterogenous lookup with the set container is somewhat more complex. To get started, here is the version using the traditional homogenous probe object approach along with the default < ordering for the container:

```
set<Thing> things; // use default operator< ordering
things.insert(Thing("Tom"));[5]
things.insert(Thing("Dick"));
things.insert(Thing("Harry"));

while(true) {
    cout << "Enter name: ";
    string str;
    cin >> str;
    auto it = things.find(Thing(str)); // construct probe Thing object
    // auto it = things.find(str);[6] // implicitly construct probe Thing object
    if(it != things.end())
        cout << "found: " << *it << endl;
    else
        cout << "not found" << endl;
    }
```

To insert objects into the container in order, the default `Thing::operator<` comparison function has to take two `const Thing&` arguments, and return `true` if the first comes before the second. However, to implement heterogenous lookup using the `std::string` variable `str` instead of a `Thing` probe object, the `set::find()` function has to use a comparison function that is different from the one used to order the container during insertion. Instead, we need a comparison function that knows to compare a `Thing`'s name with a `std:string`; moreover, because of the way the `set<>` container is implemented, the comparison needs to work in both directions: compare `Thing` name to `string` and compare `string` to `Thing` name.

The C++14 Standard committee came up with a simple, but somewhat cryptic, solution for how to supply multiple comparison functions for `set<>::insert()` and `set<>::find()` and still maintain backward compatibility with code using the pre-14 definition of `set<>`.[7] There are two ways to use the C++14 solution: using additional overloads of `operator<`, and using a custom comparison function object. In both cases, the traditional probe object approach for homogenous lookup still works.

---

[4] Inspecting the Standard template declarations shows that unfortunately, the same approach does not work for `binary_search`.

[5] The `emplace()` member function could have been used to fill the container; `insert()` is used here to be a bit more clear.

[6] The compiler can use `Thing`'s constructor to implicitly convert the `string` argument to a `Thing` object; marking the constructor `explicit` will prevent this.

[7] See C++14 (draft) Standard, section 23.2.4. If heterogenous lookup is used, the implicit conversion in the example is not performed.

Using additional overloads of `operator<`

In the first approach, we use a special form of the `std::less` function object template, which is the default ordering function in declaring a `set` container. For some type `T`, `less<T>` simply creates a function object whose `operator()` takes two arguments of the *same* type `T` and returns true if the first argument is less than the second argument, using `T`'s `operator<`. So by default, declaring a `set` holding type `T` objects uses `less<T>` as the comparison function object.

In C++14, a specialization of `less` was added[8], namely in which the type parameter is `void`, meaning *no type* is specified, written as simply `less<>`. In this case, the generated function object `operator()` takes two arguments of *different* type and returns `true` if the first is less than the second, using the `operator<` defined for those two different types. In addition, the `set<>` template was modified so that if this no-type version of `less` is involved, a different version of the `find()` function will be instantiated that accepts an argument of a different type than the objects in the container.

So to use this in our example, we need to provide `operator<` for `Thing` and `string`, and `string` and `Thing`, and then simply create the container using the no-type `less<>` comparison function object. Here is the simple code needed to get heterogenous lookup:

```
// additional overloads of less-than operator
bool operator< (const Thing& lhs, const string& rhs)
   {return lhs.get_name() < rhs;}
bool operator< (const string& lhs, const Thing& rhs)
   {return lhs < rhs.get_name();}

set<Thing, less<>> things;  // use the no-type specialization of less<T>
things.insert(Thing("Tom"));
things.insert(Thing("Dick"));
things.insert(Thing("Harry"));

while(true) {
   cout << "Enter name: ";
   string str;
   cin >> str;
   auto it = things.find(str); // no Thing probe object needed!
   if(it != things.end())
      cout << "found: " << *it << endl;
   else
      cout << "not found" << endl;
   }
```

Notice that you can't use this approach if your additional overloads involve only built-in types - remember you can't overload comparisons between two pointers, two integers, or between an integer and a pointer.

Using a custom comparison function object

Instead of using the `less<>` comparison function object, and the overloads of `operator<`, we can supply our own, as long as it includes all the required comparisons. This would be an advantage if for some reason we don't want to define the `operator<` overloads. For our example, we define a comparison function object class that has three overloaded `operator()` functions: one to compare two `Things`, one to compare `Things` to `string`, and one to compare `string` to `Things`. Each function returns `true` if the first object being compared comes before the second object in the order used to fill the container. For our example, this looks like:

```
struct Comp {
using is_transparent = std::true_type;  // see below for explanation of this line
bool operator() (const Thing& lhs, const Thing& rhs) const  // compare Things
   {return lhs < rhs;}
bool operator() (const Thing& lhs, const string& rhs) const // compare Things to strings
   {return lhs.get_name() < rhs;}
bool operator() (const string& lhs, const Thing& rhs) const // compare strings to Things
   {return lhs < rhs.get_name();}
};
```

[8] See C++14 (draft) Standard, section 20.9.6.

Then we declare our set<> container to use this comparison function object class; when we fill the container, the insert function uses the Thing-Thing operator() for the comparisons:

```
set<Thing, Comp> things;
things.insert(Thing("Tom"));
things.insert(Thing("Dick"));
things.insert(Thing("Harry"));
while(true) {
    cout << "Enter name: ";
    string str;
    cin >> str;
    auto it = things.find(str);  // no Thing object needed for probe!
    if(it != things.end())
        cout << "found: " << *it << endl;
    else
        cout << "not found" << endl;
    }
```

As with the less<> comparator, the set::find() function gets instantiated differently if this comparison function object is involved. If the argument to find() is the same as the container object type, the find() function is instantiated to use the Thing-Thing comparison operator(). However, if a string argument is supplied, the find() function is instantiated to use the two other overloads, the Thing-string and string-Thing comparisons.

Bonus: we can define additional operator()overloads in Comp if we want other types to work as probes, such as c-strings! These additional overloads can involve built-in types, just like the ordinary comparison function objects used with set<> can compare pointers.

Now what about that cryptic is_transparent line shown in the declaration of Comp? The C++14 Standards Committee needed a way to add this capability to set<> without breaking previous code using set<>; it was necessary to tell the set<> template whether it had a comparison function object that supported heterogenous lookup. The template magic solution was that if the comparison function object defined a type named is_transparent, the set<> template would instantiate the multiple find functions to use the other comparisons; if not, the old homogenous lookup version of find would be instantiated instead. Why is_transparent in particular? A bunch of Standard Library function objects templates that use perfect forwarding define is_transparent; in fact, the less<> template specialization is one of them. Our Comp function object does not need to be "transparent" - but in true C/C++ tradition, why not repurpose the key word?

What about heterogenous lookup with other containers and algorithms?

First, this is almost always a non-issue with the map<> container; you normally search using the key type rather than the objects stored as second in the container pairs. This leaves the other sequence containers such as list<>, and the find family of linear search algorithms. Note that the simple find algorithm does a homogenous lookup - it searches for a match to the supplied value using operator==() to do the comparison. However, the find_if family uses a predicate to define the match, and this predicate can compare anything you want. Thus it is easy to heterogenous lookup; here is our example using a lambda expression for the predicate:

```
list<Thing> things = { Thing("Tom"), Thing("Dick"), Thing("Harry"));

while(true) {
    cout << "Enter name: ";
    string str;
    cin >> str;
    auto it = find_if(things.begin(), things.end(),
        [&str] (const Thing& t) {return t.get_name() == str});
    if(it != things.end())
        cout << "found: " << *it << endl;
    else
        cout << "not found" << endl;
    }
```

No tricks needed for heterogenous linear search in the STL!