Generic Programming

Part 1: Allowing Generic Comparisons

You can define a comparison operator for a class/struct like this:

```
class Dog
{
    public:
        int getWeight() const
        {
            return m_weight;
        }
        ...
    private:
        int m_weight;
};
bool operator>=(const Dog &a, const Dog &b)
{
    if(a.getWeight() >= b.getWeight())
        return true;
    return false;
}
```

You can define >, <, >=, <=, ==, !=

All comparison operators must return a Boolean value.

Your comparison function should compare object a against object b using whatever approach makes sense. Here we say dog a is greater than dog b if its weight is bigger.

All comparison operators must accept constant reference parameters.

You can define your comparison operator inside or outside the class. Since the operator function above is defined outside the class, it cannot access private member variables.

The Solution

In C++, we use C++'s "template" function to solve this problem.
To turn any function into a "generic function"

1. Add the line `template <typename xxx>` above your function
2. Then use xxx as your data taype throught the function: `swap(xxx a, xxx b)`

```
template <typename Item>

void swap(Item &a, Item &b)

{

    Item temp;

    temp = a;

    a = b;

    b = temp;

}
```

- Always place your templated functions in a header file.
- You must put the entire template function in the header file, not just the prototype.

```
Swap.H

template<typename Data>

void swap(Data &x, Data &y)

{

    Data temp;

    temp = x;

    x = y;

    y = temp;

}

MyColProgram.CPP

#include "Swap.h"

int main()

{

    int a = 5, b = 6;

    swap(a, b); //GOOD!

}
```

- Every time you use a template function with a different type of variable, the compler generates a new version of the function in your program.

- You must use the template data type to define the type of at least one formal parameter, or you'll get an error!

```
BAD:

template <typename Data>

Data getRandomItem(int x)

{

    Data temp[10];

    return temp[x];

}

GOOD:

template <typename Data>

void swap(Data &x, Data &y)

{

    Data temp;

    temp = x;

    x = y;

    y = temp;

}
```

- If a function has two or more "templated parameters" with the same type, you must pass in the same type of variable/value for both.

```
MAX.H

template <typename Data>

Data max(Data x, Data y)

{

    if(x > y)

        return x;

    else

        return y;

}

==============================

#include "max.h"
```

```
int main()
{
    int i = 5;
    float f = 6.0;
    cout << max(i, f); //ERROR!
    Dog c;
    Cat d, e;
    cout << max(c, d); //ERROR!
}
```

- You may override a templated function with a specialized (non-templated) version if you like.
  - If C++ sees a specialized version of a function, it will always choose it over the templated version.

```
Dog bigger(Dog &x, Dog &y)
{
    if(x.bark() > y.bark())
        return x;
    else if (x.bark() < y.bark())
        return y;
    if(x.bite() > y.bite())
        return x;
    else
        return y;
}
template <typename Data>
Data bigger(Data &x, Data &y)
{
    if(x > y)
        return x;
    else
        return y;
}
int main()
```

```
{
    Circle a, b, c;
    c = bigger(a, b); //uses the templated version of bigger
    Dog fido, rex, winner;
    winner = bigger(fido, rex); //uses the specialized version of bigger just for Dogs
}
```

If your templated function uses a comparison operator on templated variables, then C++ expects that all variables passed in will have that operator defined.
So if you use such a function with a user-defined class, you must define a comparison operator for that class!

```
template <typename Data>
void winner(Data &x, Data &y)
{
    if(x > y)
        cout << "first one wins!\n";
    else
        cout << "second one wins!\n";
}
bool operator>(const Dog &a, const Dog &b)
{
    if(a.weight() > b.weight()
        return true;
    else
        return false;
}
bool operator>(const Circ &a, const Circ &b)
{
    if(a.radius() > b.radius())
        return true;
    else
        return false;
}
```

```
int main()
{
    int i1 = 3;
    int i2 = 4;
    winner(i1, i2);

    Dog a(5), b(6);
    winner(a, b); //works!

    Circ c(3), d(4);
    winner(c, d); //works!
}
```

Multi-type Templates

```
template <typename Type1, typename Type2>
void foo(Type1 a, Type2 b)
{
    Type1 temp;
    Type2 array[20];
    temp = a;
    array[3] = b;
    //etc....
}
int main()
{
    foo(5, "barf");
    foo("argh", 6);
    foo(42, 52);
}
```

We can use templates to make entire classes generic too:

```
template <typename Item> //you must use this prefix before the class definition itself
class HoldOneValue
{
    public:
        void setVal(Item a) //update the appropriate types in your class
        {
            m_a = a;
        }
        void printTenTimes()
        {
            for(int i = 0; i < 10; i++)
                cout << m_a;
        }
    private:
        Item m_a; //Now you class can hold any type of data you like
};
```

```
int main()
{
    HoldOneValue<int> v1;
    v1.setVal(10);
    v1.printTenTimes();

    HoldOneValue<string> v2;
    v2.setVal("ouch");
    v2.printTenTimes();
}
```

In classes with externally-defined member functions, things get ugly!
You add the prefix `template <typename xxx>` **before the class definition itself** AND **before each function definition** outside the class. Then update the types to use the templated type. Finally, place the postfix `<xxx>` between the class name and the `::` in all function definitions.

```
template <typename Item>
class Foo
{
    public:
        void setVal(Item a);
        void printVal();
    private:
        Item m_a;
};

template <typename Item>
void Foo<Item>::setVal(Item a)
{
    m_a = a;
}

template <typename Item>
void Foo<Item>::printVal()
{
    cout << m_a << "\n";
}
```

Template classes are very useful when we're building container objects like linked lists.

```
template <class HoldMe>
class LinkedList
{
    public:
        LinkedList();
        bool insert(HoldMe &value);
        bool delete(HoldMe &value);
        bool retrieve(int i, HoldMe &value);
```

```
        int size();
        ~LinkedList();
    private:
        ...
};
```

```
int main()
{
    Dog fido(10);
    LinkedList<Dog> dogLst;
    dogLst.insert(fido);

    LinkedList<string> names;
    names.insert("Seymore");
    names.insert("Butts");
}
```

Carey's Template Cheat Sheet

> To templatize a non-class function:

- Update the function header
- Replace appropriate types in the function to the new ItemType

```
//before
int bar(int a)
{
    int b;
    ...
}
//after
template <typename ItemType>
```

```
ItemType bar(ItemType a)

{

    ItemType b;

    ...

}
```

To templatize a class:

- Put `template <typename xxx>` in front of the class declaration

```
//before

class foo {...};

//after

template <typename xxx>

class foo {...};
```

- Update appropriate types in the class to the new ItemType
- Update internally defined methods
  - For normal methods, just update all types to ItemType
  - Assignment operator

```
//before

foo& operator=(const foo &other);

//after

foo<ItemType>& operator=(const foo<ItemType> &other);
```

  - Copy constructor

```
//before

foo(const foo &other);

//after

foo(const foo<ItemType> &other);
```

- Update externally defined methods
  - For non inline methods

```
//before
```

```
int foo::bar(int a);

//after

template <typename ItemType>

ItemType foo<ItemType>::bar(ItemType a);
```

- For inline methods

```
//before

inline

int foo::bar(int a)

//after

template <typename ItemType>

inline

ItemType foo<ItemType>::bar(ItemType a);
```

- Copy constructor

```
//before

foo& foo::operator=(const foo &other);

//after

foo<ItemType>& foo<ItemType>::operator=(const foo<ItemType>& other);
```

- Assignment operator

```
//before

foo::foo(const foo &other);

//after

foo<ItemType>::foo(const foo<ItemType>& other);
```

- If you have an internally defined struct, simply remplace the appropriate internal variables in your struct

```
template <typename ItemType>

class foo

{

    struct blah
```

```
    {
        ItemType val;
    }
}
```

- If an internal method in a class is trying to return an internal struct (or a pointer to an internal struct), you don't need to change the function's declaration at all inside the class declaration, but just update variables to your `ItemType`
- If an externally-defined method in a class is trying to return an internal struct (or a pointer to an internal struct)...

Assuming your internal structure is called "blah"

```
//before

blah foo::bar();

blah* foo::bar();

//after

template <typename ItemType>

typename foo<ItemType>::blah foo<ItemType>::bar();

template <typename ItemType>

typename foo<ItemType>::blah* foo<ItemType>::bar();
```

- Try to pass templated items by const reference if you can (to improve performance)
  - Bad: `template <typename ItemType> void foo(ItemType x)`
  - Good: `template <typename ItemType> void foo(const ItemType &x)`

The Standard Template Library

Vector

The STL vector is a template class that works just like an array, only it doesn't have a fixed size. Vectors grow/shrink automatically when you add/remove items.
To use a vector, make sure to `#include <vector>`

```
#include <vector>

using namespace std;

int main()

{

    //all of a vector's initial elements are automatically initialized/constructed

    vector<string> strs; //empty vector
```

```
    vector<int> nums;

    vector<Robot> robots;


    vector<int> geeks(950); //starts with 950 elements

}
```

Once you've created a vector, you can add items, change items, or remove items.

To add a new item to the very end of the vector, use the `push_back` command.
To remove an item from the back of a vector, use `pop_back`.
To get the current number of elements in a vector, use the `size()` method.
And to determine if the vector is empty, use the `empty()` method.

```
#include <vector>

using namespace std;

int main()

{

    vector<int> vals(3);

    vals.push_back(123);

    vals[0] = 42;

    cout << vals[3]; //returns 123

    vals[4] = 1971; //crash, there is no #4 in the vector

    cout << vals[7]; //crash

    cout << vals.back(); //returns 123

    vals.pop_back();

    vals.pop_back();


    vector<int> vals2(2, 444); //vector now includes 444 444

    vals.push_back(999); //444 444 999

    cout << vals.size(); //returns 3

    if(vals.empty() == false)

        cout << "I have items!";

}
```

List

The STL list is a class that works just like a linked list. Like vector, the list class has `push_back`, `pop_back`, `front`, `back`, `size`, and `empty` methods. The list class also has `push_front` and `pop_front` methods. These methods allow you to add/remove items from the front of the list!
Unlike vectors, you can't access list elements using brackets.

```
#include <list>

using namespace std;

int main()

{

    list<float> lf;

    lf.push_back(1.1);

    lf.push_back(2.2);

    lf.push_back(3.3);

    cout << lf[0] << endl; //error!

}
```

Vectors vs. Lists
Since vectors are based on dynamic arrays, they allow fast access to any element, but adding new items is often slower.
The list is based on a linked list, so it offers fast insertion/deletion, but slow access to middle elements.

Iterating Through the Items

Other than the vector class, none of the other STL containers have an easy-to-use "retrieve" method to quickly go through items.
To enumerate the contents of a container, you typically use an iterator variable.
An iterator variable is just like a pointer variable, but it's used just with STL containers.
Typically, you start by pointing an iterator to some item in your container. Just like a pointer, you can increment and decrement an iterator to move it up/down through a container's items. You can also use the iterator to read/write each value it points to.
To define an iterator variable, write the container type followed by two colons, followed by the word iterator and then a variable name.
`it = myVec.begin()` points the iterator at the first item.
You can move the iterator up or down by using `--` and `++` respectively.
Once the iterator points at a value, you can use the `*` operator with it and then the dot operator to access the value.
Or you can use the `->` operator.

```
int main()

{
```

```
    vector<int> myVec;


    myVec.push_back(1234);
    myVec.push_back(5);
    myVec.push_back(7);


    vector<int>::iterator it;
    it = myVec.begin();
    cout << (*it); //dereferenced iterator
    it++; //you can move your iterator down one item by using the ++ operator
    cout << (*it);
    it--; //you can use the -- operator to move the iterator backward
    //Output: 1234 5
    //iterator now points to 1234
    list<Nerd> nerds;
    Nerd d;
    nerds.push_back(d);
    list<Nerd>::iterator it;
    it = nerds.begin();
    (*it).beNerdy();
    it->beNerdy();
}
```

What if you want to point your iterator to the last item in the container?
Each container has an end() method, but it doesn't point to the last item! It points just past the last item in the container. So if you want to get to the last item, you've got to decrement your iterator first!

```
int main()
{
    vector<int> myVec;
    myVec.push_back(1234);
    myVec.push_back(5);
    myVec.push_back(7);
```

```
    vector<int>::iterator it;

    it = myVec.end();

    it--;

    cout << (*it); //prints out the last item

    //output: 7

}
```

Const Iterators
Sometimes you'll pass a container as a const reference parameter. To iterate through such a container, you can't use the regular iterator. It's easy to fix...just use a const iterator!

```
void tickleNerds(const list<string> &nerds)

{

    list<string>::const_iterator it;

    for(it = nerds.begin(); it != nerds.end(); it++)

    {

        cout << *it << " says teehee!\n";

    }

}

int main()

{

    list<string> nerds;

    nerds.push_back("Carey");

    nerds.push_back("David");

    tickleNerds(nerds);

}
```

Map

Maps allow us to associate two related values. Let's say I want to associate a bunch of people with each person's phone number, so names are stored in string variables and phone numbers are stored in integers.

```
#include <map>

#include <string>
```

```
using namespace std;

int main()

{

    map<string, int> name2Fone;

    name2Fone["Carey"] = 818551212;

    name2Fone["Joe"] = 3109991212; //maps automatically sort alphabetically

    //if int->string, map sorts by ascending

}
```

The map class basically stores each association in a struct variable!

```
struct pair

{

    string first;

    int second;

};
```

A given map can only associate in a single direction. For example, our name2Fone map can associate a string to an int, but not the other way around!
If you want to efficiently search in both directions, you have to use two maps.

```
#include <map>

#include <string>

using namespace std;

int main()

{

    map<int, string> fones2Names;

    fones2Names[4059913344] = "Ed";

    fones2Names[8183451212] = "Al"; //maps automatically sort alphabetically

    //if int->string, map sorts by ascending

}
```

How to search the map class:
To find a previously added association, first define an iterator to your map. Then, you can call the map's `find()` command in order to locate an association. You can only search based on the left-hand type! Then you can look at the pair of values pointed to by the iterator!

```
#include <map>

#include <string>

using namespace std;

int main()

{

    map<string, int> name2Age;

    map<string, int>::iterator it;

    it = name2Age.find("Dan");

}
```

If the find method can't locate your item, then it tells you this by returning an iterator that points past the end of the map! `it = name2Age.end();`

To iterate through a map, simply use a for/while loop as we did for vectors/lists:

```
#include <map>

#include <string>

using namespace std;

int main()

{

    map<string, int> name2Age;

    map<string, int>::iterator it;

    for(it = name2Age.begin(); it != name2Age.end(); it++)

    {

        cout << it->first;

        cout << it->second;

    }

}
```

On final: define your own operator< method for the left-hand class/struct.
You only need to define the operator< method if you're mapping **from** your own struct/class.

```
struct stud

{

    string name;
```

```
    int idNum;
};
int main()
{
    map<strud, float> stud2GPA; //this map associates a given Student with their GPA
    stud d;
    d.name = "David Smallberg";
    d.idNum = 916351351;
    stud2GPA[d] = 1.3;
}
//for the above to work, you have to define an operator< method for stud
//if it had been map<float, stud>, there would have been no need for an operator< method
bool operator<(const stud &a, const stud &b)
{
    return (a.name < b.name);
}
```

Set

A set is a container that keeps track of unique items. If you insert a duplicate item into the set, it is ignored (since it's already in the set!).

```
#include <set>
using namespace std;

int main()
{
    set<int> a;
    a.insert(2);
    a.insert(3);
    a.insert(4);
    a.insert(2); //this is ignored
    cout << a.size();
```

```
    a.erase(2);
}
```

You can have sets of other data types as well! But as with our map, you need to define the operator< method for your own classes. Otherwise you'll get a compile error!

```
struct Course
{
    string name;
    int units;
};
int main()
{
    set<Course> myClasses;
    Course lec1;
    lec1.name = "CS32";
    lec1.units = 16;
    myClasses.insert(lec1);
}
bool operator<(const Course &a, const Course &b)
{
    return (a.name < b.name);
}
```

Searching/Iterating Through a Set:
We can search the STL set using the `find()` function and an iterator, just like we did for the map!

```
#include <set>
using namespace std;
int main()
{
    set<int> a;
    a.insert(2);
    a.insert(3);
    a.insert(4);
```

```
    set<int>::iterator it;
    it = a.find(2);
    if(it == a.end())
    {
        cout << "2 was not found";
        return 0;
    }
    cout << "I found " << (*it);
    //OR
    it = a.begin();
    while(it != a.end())
    {
        cout << *it;
        it++;
    }
}
```

Deleting an Item from an STL Container

Most containers have an `erase()` method you can use to delete an item. First you search for the item you want to delete an get an iterator to it. Then, if you found an item, use the `erase()` method to remove the item pointed to by the iterator.

```
int main()
{
    set<string> geeks;
    geeks.insert("Carey");
    geeks.insert("Rick");
    geeks.insert("Alex");
    set<string>::iterator it;
    it = geeks.find("Carey");
    if(it != geeks.end())
    {
```

```
        cout << "bye bye" << *it;
        geeks.erase(it);
    }
}
```

However....
If you point an iterator to an item in a vector and then you either add or erase an item from the same vector, all the old iterators that were assigned before the add/erase are *invalidated*.

```
int main()
{
    vector<string> x;
    x.push_back("Carey");
    x.push_back("Rick");
    x.push_back("Alex");
    vector<string>::iterator it;
    it = x.begin();
    x.push_back("Yong")l
    x.erase(it);
    cout << *it; //I'm no longer valid!!
}
```

Fortunately, this same problem doesn't occur with sets, lists, or map.
With one exception...if you erase the item the iterator points to, then you've got troubles!
Erasing odd integers from a list:

```
void removeOdds(list<int>& li)
 {
     list<int>::iterator it;
     it = li.begin();
     while(it != li.end())
     {
         if(*it % 2 != 0)
         {
             it = li.erase(it);
```

```
        }
        it++;
    }
 }
```

Erasing odd integers from a vector:

```
void removeOdds(vector<int>& v)
{
     vector<int>::iterator it;
     it = v.begin();
     while(it != v.end())
     {
         if(*it % 2 != 0)
         {
             it = v.erase(it);
             it--;
         }
         it++;
     }
}
```

```
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
#include <cassert>
using namespace std;

vector<int> destroyedOnes;

class Movie
```

```
{
public:
    Movie(int r) : m_rating(r) {}
    ~Movie() { destroyedOnes.push_back(m_rating); }
    int rating() const { return m_rating; }
private:
    int m_rating;
};

// Remove the movies in li with a rating below 50 and destroy them.
// It is acceptable if the order of the remaining movies is not
// the same as in the original list.
void removeBad(list<Movie*>& li)
{
    list<Movie*>::iterator it;
    it = li.begin();
    while(it != li.end())
    {
        Movie* mp = *it;
        if(mp->rating() < 50)
        {
            li.erase(it);
            it--;
            delete mp;
        }
        it++;
    }
}

void test()
{
    int a[8] = { 85, 80, 30, 70, 20, 15, 90, 10 };
```

```
list<Movie*> x;
for (int k = 0; k < 8; k++)
    x.push_back(new Movie(a[k]));
assert(x.size() == 8 && x.front()->rating() == 85 && x.back()->rating() == 10);
removeBad(x);
assert(x.size() == 4 && destroyedOnes.size() == 4);
vector<int> v;
for (list<Movie*>::iterator p = x.begin(); p != x.end(); p++)
{
    Movie* mp = *p;
    v.push_back(mp->rating());
}
// Aside:  In C++11, the above loop could be
//     for (auto p = x.begin(); p != x.end(); p++)
//     {
//         Movie* mp = *p;
//         v.push_back(mp->rating());
//     }
// or
//     for (auto p = x.begin(); p != x.end(); p++)
//     {
//         auto mp = *p;
//         v.push_back(mp->rating());
//     }
// or
//     for (Movie* mp : x)
//         v.push_back(mp->rating());
// or
//     for (auto mp : x)
//         v.push_back(mp->rating());
sort(v.begin(), v.end());
int expect[4] = { 70, 80, 85, 90 };
```

```
    for (int k = 0; k < 4; k++)
        assert(v[k] == expect[k]);
    sort(destroyedOnes.begin(), destroyedOnes.end());
    int expectGone[4] = { 10, 15, 20, 30 };
    for (int k = 0; k < 4; k++)
        assert(destroyedOnes[k] == expectGone[k]);
    for (list<Movie*>::iterator p = x.begin(); p != x.end(); p++)
        delete *p;
}

int main()
{
    test();
    cout << "Passed" << endl;
}
```

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <cassert>
using namespace std;

vector<int> destroyedOnes;

class Movie
{
public:
    Movie(int r) : m_rating(r) {}
    ~Movie() { destroyedOnes.push_back(m_rating); }
    int rating() const { return m_rating; }
private:
```

```
    int m_rating;
};

// Remove the movies in v with a rating below 50 and destroy them.
// It is acceptable if the order of the remaining movies is not
// the same as in the original vector.
void removeBad(vector<Movie*>& v)
{
    for (vector<Movie*>::iterator p = v.begin(); p != v.end(); p++)
    {
        Movie *mp = *p;
        if(mp->rating() < 50)
        {
            v.erase(p);
            p--;
            delete mp;
        }
    }
}

void test()
{
    int a[8] = { 85, 80, 30, 70, 20, 15, 90, 10 };
    vector<Movie*> x;
    for (int k = 0; k < 8; k++)
        x.push_back(new Movie(a[k]));
    assert(x.size() == 8 && x.front()->rating() == 85 && x.back()->rating() == 10);
    removeBad(x);
    assert(x.size() == 4 && destroyedOnes.size() == 4);

    vector<int> v;
    for (int k = 0; k < 4; k++)
```

```
        v.push_back(x[k]->rating());
    sort(v.begin(), v.end());
    int expect[4] = { 70, 80, 85, 90 };
    for (int k = 0; k < 4; k++)
        assert(v[k] == expect[k]);
    sort(destroyedOnes.begin(), destroyedOnes.end());
    int expectGone[4] = { 10, 15, 20, 30 };
    for (int k = 0; k < 4; k++)
        assert(destroyedOnes[k] == expectGone[k]);
    for (vector<Movie*>::iterator p = x.begin(); p != x.end(); p++)
        delete *p;

}

int main()
{
    test();
    cout << "Passed" << endl;
}
```

STL Algorithms

The standard template library also provides some additional functions that work with many different types of data. The `find()` function can search most STL containers and arrays for a value. The `set_intersection()` function can compute the intersection of two sorted sets/lists/arrays of data. The `sort()` function can sort arrays/vectors/lists.

The sort function:

To sort, you pass in two iterators: one to the first item and one that points just past the last item you want to sort.

```
#include <vector>
#include <algorithm>
int main()
{
    vector<string> n;
```

```
    n.push_back("Carey");
    n.push_back("Bart");
    n.push_back("Alex");
    sort(n.begin(), b.end());
    sort(n.begin(), n.begin() + 2);
    int arr[4] = {2, 5, 1, -7};
    sort(&arr[0], &arr[4]);
}
```

Summary of STL Methods

Vector

- `push_back()`
- `pop_back()`
- `front()`
- `back()`
- `size()`
- `empty()`
- `erase()`
- can access methods using brackets

List

- `push_back()`
- `pop_back()`
- `front()`
- `back()`
- `size()`
- `empty()`
- `push_front()`
- `pop_front()`

Set

- `insert()`
- `size()`
- `erase()`
- `find()`

Inline Functions

When you define a function as being `inline`, you ask the compiler to directly embed the function's logic into the calling function (for speed). By default, all methods with their body defined directly in the class are inline. To make an externally-defined method inline, add the word `inline` right before the function return type.

```
class Foo
{
```

```
    public:
        void setVal(int a);
        ...
}
inline
void Foo::setVal(int a);
```