Generic Programming

Introduction

Generic Programming: when you write a function or class in a manner so that it can process many different types of data

E.g. generic sorting function that can sort an array holding ANY type of value

E.g. generic linked list class that can hold ANY variable

```
int main() {
    list<int> list_of_integers;
    list<string> list of strings;
}
```

Generic Comparisons

Consider the following `main` function that compares various objects to each other

```
int main()
{
    int i1 = 3, i2 = 5;
    if (i1 > i2)
        cout << "i1 is bigger";

    //Compare circle
    Circ a(5), b(6);
    if (a.radius() > b.radius())
        cout << "a was bigger";

    //Compare dog
    Dog fido(10), spot(20);

    if (fido.weight() > spot.weight())
        cout << "fido is bigger";
}
```

- The way we compare two `dogs` (by weight) is different than the way we compare two `circles` (by radius)
- We COULD just do this instead if we defined *custom comparison operators*

```
fido > spot
```

Define Custom Comparions Operators Syntax

- You can define `==`, `<`, `>`, `<=`, `>=`, and `!=`
- All comparison operators must return a Boolean value
- All comparison operators accept **const reference** parameters
    - Leave `const` out that cause compiler errors

Comparison operators defined **outside** the class

```
bool operator>=(const Dog& a, const Dog& b)
{
    if (a.getWeight() >= b.getWeight())
        return true;
    return false; //otherwise
}
```

- Two parameters, one for each of the two operands
- May only use public methods from the class
- Since parameter is `const` reference, our function can only call `const` functions in `Dog`
    - `getWeight()` needs to be `const` as well
    - Since we can't change the parameter, so calling any `nonconst` function on the object would be illegal
    - WILL CAUSE COMPILER ERROR

Comparison operators defined **inside** the class

```
bool operator<(const Dog& other) const
{
    if (m_weight < other.m_weight)
        return true;
    return false;
}
```

- Just have a single `other` parameter
    - like a copy constructor
    - `other` refers to the value to the **right** of the operator

```
if (a < other)
```

- Must be a `const` function

Trace through

- Use the operator in code -> C++ calls comparison function
- Enter body of comparison function and process code
- Returns boolean and exits function

Generic Functions: Templates

Suppose you need to write a swap function for both `Dog` and `Circle`, we have to write two different functions currently...

> This is a pattern for manufacturing functions...

Convention

```
template<typename T>

T functionName(T a, T b)
...
```

Define generic function

1. Add the following line above function

   ```
   template <typename xxx>
   ```

   - Can use literally any random typename for this

2. Then use `xxx` as your data type through the function

   ```
   swap(xxx a, xxx b)
   ```

Function Code

```
template<typename Item>

//or template<class Item> works the same way

void swap(Item &a, Item &b)
{
```

```
    Item temp;
    temp = a;
    a = b;
    b = temp;
}
```

Use the function

```
int main()
{
    Dog d1(10), d2(20);
    Circle c1(1), c2(5);

    swap(d1, d2);
    swap(c1, c2);
}
```

Function Template Details

1. Always place your templated functions in a header file
    - Include header file in your CPP files to use function
    - Must put ENTIRE template function in the header file

```
Swap.h

template <typename Data>

void swap(Data& x, Data& y)
{
    Data temp;

    temp = x;
    x = y;
    y = temp;
}
```

2. Each time you use a template function with a different type of variable, the compiler generates a new version of the function in your program

```
#include "Swap.h"
int main()
{
    Dog a(13), b(41);
    swap(a, b);
```

```
    int p = -1, q = -2;
    swap(p, q);

    string x("a"), y("b");
    swap(x, y);

    int r = 10, s = 20;
    swap(r, s);
}
```

- Compiler generates 3 versions of `swap` here
- That is what is actually executed, compiler sees template and generates a function for you
- **Template argument deduction**: compiler deducts the types of the argument and helps you generate

3. MUST use template data type to define the type of at least **one formal parameter**, or ERROR!

   Good:

```
template <typename Data>
void swap(Data& x, Data& y)
{
    Data temp;
    temp = x;
    x = y;
    y = temp;
}
```

   - Data use to specify the types of `x` and `y`

   BAD:

```
template <typename Data>
Data getRandomItem(int x)
{
    Data temp[10];
    return(temp[x]);
}
```

   - Data was not use to specify the type of any parameters

4. If a function has 2 or more templated parameters with the same type, MUST pass in the same type of variable/value for both

```
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
}
```

5. Templated function and custom comparison operators
    - If your templated function uses a comparison operator on templated variables -> then C++ expects that all variables passed in will have that operator defined
    - MUST define a comparison operator
6. The call matches some template

```
...minimum(3, 3,5);
```

- This function passes an **int** and a **double**
- There are NO templates for int and double function
    - Even though we can compare an int to a double, or conversion of int to a double
- The type has a match **exactly**
- Compilation error

7. The instantiated template must compile

```
Chicken c3 = minimum(c1, c2);
```

- Types match, so function is generated
- But this is a custom type, we have to define a < operator for our custom type -> so the template will NOT compile

8. The instantiated template must do what you want

```
cin.getline(ca1, 100);
cin.getline(ca2, 100);
char* ca3 = minimum(ca1, ca2);
```

- These are C-strings
- It is not comparing the two strings, because these are *array* of characters, will compare two pointers...
- Will compile, just not compare what you want

We have to...

```
char* minimum(char* a, char* b)
{
    if (strcmp(a, b) < 0)
        return a;
    else
        return b;
}
```

9. For matching, the only conversions considered for any type `A` are:
    - `A` => `A&`
    - `A` => `const A`
    - array of `A` => `A*`
    - e.g. int matches `const T&`, `T` will be int

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);
}
```

Caution, Example:

```
template<typename T1, typename T2>
??? minimum(T1 a, T2 b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

```
int main()
{
    int k;
    cin >> k;
    minimum(k, 3.5);
}
```

- What should the return type be?
    - In this case, it is not clear what the return type should be
    - If we put as int:

      `minimum(3.5, k)` and `k = 10`

      Then `a < b` and we return `a`, but `a` is converted to an int, return wrong value

  etc.

Passing By **Const Reference** in Template Functions

```
int main()
{
    ExpensiveToCopyThing x, y;
    minimum(x, y);
}
```

- Do I really want to pass by value if I just want to look at them?

```
T minimum(const T& a, const T& b)
```

- Cheap
- Still work correctly

Writing Template Functions for Different Types (Initialization)

```
double sum(const double a[], int n)
{
    double total = 0;
    ...
}

string sum(const string a[], int n)
{
    string total = 0; //string cannot be initialized this way
    string total; //if we do this, initialization for double will NOT work
    string total = string(); //double built-in types have no default constructors
}
```

Temporary Object

```
double total = double();
string total = string();
```

- NOW THIS IS LEGAL
- Extended the language to allow for built-in types

Writing Generic Classes

Class Code

```
template <typename Item>

class HoldOneValue
{
    public:
        void setVal(Item a)
        {
            m_a = a;
        }

        void printTenTimes()
        {
            for (int i = 0; i < 10; i++)
                cout << m_a;
        }
    private:
```

```
        Item m_a;
}
```

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

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

Classes with externally-defined member functions

1. Add the prefix: `template <typename xxx>` before the class definition itself AND before each function definition outside the class
2. Update the types of use templated type
3. Place the postfix `<xxx>` between the class name and the `::` in all function defs

```
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 Cheat Sheet (Yet to be completed)

The Standard Library (STL)

The Standard Template Library (STL) is a collection of pre-written, tested classes provieded by authors of C++

Classes all built using templates -> can be used with many different data types

- **Container classes**: classes that hold groups of items

STL Container 1: Vector Class

STL vector is a template class that works just like an array but with no fixed size

Create vector

```
#include <vector>

using namespace std;

int main()
{
    //Empty vector (0 initial elements)
    vector<string> strs;
    vector<int> nums;
    vector<Robot> robots;

    //Vector that starts with N elements
    vector<int> geeks(950);
        //Automatically initialized at 0
}
```

Add item

```
//add
strs.push_back("Carey");
strs.push_back("Scott");
```

```
vector<int> vals(3);
vals.push_back(123); //adds the fourth element
```

Insert

```
q = v.insert(p, 40);
//returns an iterator pointing to new item
```

- New memory allocation

Change item

```
vals[0] = 42;
```

- Only use brackets to access existing items

Read or write first/last elements

```
vals.back();
vals.front();
```

Remove an item

```
vals.pop_back(); //removes item from the back
```

- Shrinks the vector

Get current number of elements `size()`

```
vals.size();
```

Determine if vector is empty

```
if (vals.empty() == false)
    cout << "I have items!";
```

- Vectors are based on dynamic arrays -> allow fast element access but slow add

STL Container 2: List Class

The STL `list` is a class that works just like a linked list

Methods:

`push_back`

`pop_back`

`front`

`back`

`size`

`empty`

`push_front` & `pop_front`

`insert`

- `q = li.insert(p, 40);`
    - where in the list, and what item
    - `p` is the iterator
    - returns another list iterator that points to 40
- Can't access list elements using brackets
- STL list is based on a linked list -> fast insertion/deletion but slow access to middle elements

Iterate through STL Container Classes

To enumerate the contents of a container -> typically use an iterator variable

- Like a pointer variable, but it's used just with STL containers

1. Start by pointing an iterator to some item in container (e.g. first item)
2. Just like a pointer, increment and decrement an iterator to move it up/down through a container's items
3. Use iterator to read/write each value it points to

- We do not have general pointer arithmetic as with normal pointers
    - Only increment decrement
    - A vector iterator, we can do pointer arithmetic

```
p = v.end() - 2;
```

Define an Iterator:

write the container type + two colons + word iterator + variable name

```
vector<int>:: iterator it;
vector<string>:: iterator it2;
list<float>::iterator it3;
```

Use an iterator

1. Point iterator at the first item

```
it = myVec.begin();
```

2. Once the iterator points at a value, can use the `*` operator with it to access the value

```
cout << (*it);
```

   - Operator overloading when you use `*` to dereference an iterator, since it usually dereferences pointers

3. Increment/Decrement iterator

```
it++;
```

   - Moves iterator down one item, now the iterator points to the secon ditem

```
it--;
```

   - Moves the iterator backwards

4. Point iterator to the last item in container?

   - Each container has an `end()` method, but this points JUST PAST the last item the container
     - `it = myVec.end()` will make it point to null
   - Decrement it after this so it points to the last item

```
it = myVec.end();
it--;
```

5. Loop through container

```
vector<int>::iterator it;
it = myVec.begin();

while (it != myVec.end())
{
    cout << (*it);
    it++;
}
```

Deleting an Item from an STL Container

Most STL containers have an `erase()` method you can use to delete an item

1. Search for the item you want to delete and get an iterator to it

2. If you fond an item, use `erase() method to remove the item pointer to by the iterator

```
set<string>::iterator it;
it = geeks.find("carey");
if (it != geeks.end())
{
    //found my item!!
    cout << "bye bye" << *it;
    geeks.erase(it);
}
```

STL and Classes/Structs

```
struct Thing
{
    string first;
    int second;
    float third;
};
```

```
int main()
{
    list<Thing> things;

    Thing d;
    d.first = "IluvC++";
    d.second = 300;
    d.third = 3.1415;

    things.push_back(d);

    list<Thing>::iterator it;
    it = things.beging();
    cout << (*it).first; //is equivalent to
    cout << it -> first;
    it -> third = 2.718;

    //This is a class
    list <Nerd> nerds;
    Nerd d;
    nerds.push_back(d);

    list<Nerd>::iterator it;
    it = nerds.begin();
```

```
    (*it).beNerdy();
    it -> beNerdy();
}
```

Const Iterators

> When you pass a container as a const reference parameter into some function

- Cannot use regular iterator
- Use `const_iterator`

```
void tickleNerd(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);
}
```

STL Iterators Internal Workings

An iterator is an object (a class variable) that knows three things:

1. What element it points to
2. How to find the previous element in the container
3. How to find the next element in the container

```
struct Node
{
    int value;
    Node* next;
    Node* prev;
}

class MyIterator
{
```

```
    public:
        int getVal() { return cur->value };
        void down() { cur = cur->next; }
        void up() { cur = cur->prev; }

        Node *cur;
};
```

The `begin()` method in container classes

```
class LinkedLIst
{
    public:
    ...
    MyIterator begin()
    {
        MyIterator temp;
        temp.cur = m_head;
        return(temp);
    }
    private:
        Node* m_head;
};
```

Using the code

```
int main()
{
    LinkedList GPAs; //list of GPAs
    ...
    MyIterator itr = GPAs.begin();
    cout << itr.getVal(); //like *it
    itr.down(); //like it++;
    cout << itr.getVal();
}
```

Iterator Notes

For a vector...

- When you add an item anywhere in a vector you must assume the iterator previously put is invalidated
- If you erase that item or an item that comes before the iterator, your iterator is also invliadted
- Add/erase items in a vector shuffles memory around -> iterators my not point to the right place anymore
- Same problem does not occur with `sets`, `lists`, or `maps`

```
int main()
{
    vector<string> x;
    x.push_back("Carey");
    x.push_back("Rick");
    x.push_back("Alex");

    vector<string>::iterator it;
    it = x.end();
    it--; //it points at Alex

    x.push_back("Yong"); //add OR
    x.erase(x.begin()); //kill earlier item

    cout << *it; //ERROR!!!!
}
```

However for all STL containers: deletion notes

- Do not use iterator after erase the item that the iterator points to

```
it = s.find("carey");
s.erase("carey");
```

- We have to assign `it` a new value

```
it = s.erase("carey");
```

Other STL Containers

STL Class 3: Map

`Map` allow us to associated two related values

`<key, value>`

- A key can associate to value, but not the other way around
- Map always maintains its items in alphabetical order
    - When you iterate through them, they are automatically ordered for you

Example:

```
#include <map>
#include <string>

using namespace std;
int main()
{
    map <string, int> name2Fone;
    name2Fone["Carey"] = 8185551212;
    name2Fone["Joe"] = 3109991212;

    name2Fone[4059913344] = "Ed"; //ERROR

    map<int, string> fone2Names;
    fones2Names[4059913344] = "Ed";
}
```

How the Map Class Works:

The map class bascially stores each association in a `struct` variable

```
struct pair
{
    string first;
    int second;
};
```

```
int main()
{
    map<string, int> name2Age;
    name2Age["Carey"] = 40;
    name2Age["Dan"] = 22;

    //change value
    name2Age["Carey"] = 39;
}
```

Find a previously added association

```
int main()
{
    map<string, int> name2Age;
    map<string, int>::iterator it;

    it = name2Age.find("Dan");

    cout << (*it).first;
```

```
    cout << (*it).second;
}
```

- Points iterator to the key "Dan" using `find()`
    - Can only find the keys, not values
    - Then you can look at the pair of values pointer to by the iterator

Find a key that does not exist

```
it = name2Age.find("Ziggy");
if (it == name2Age.end())
{
    cout << "Not found!\n";
    return;
}
```

- If the `find` method cannot locate the item, then it returns an iterator that points past the end of the map
- Check for this case

Iterate through a map using a for/while loop

```
map<string, int>::iterator it;

for (it = name2Age.begin(); it != name2Age.end(); it++)
{
    cout << it->first;
    cout << it->second;
}
```

Associate more complex data types

> MUST define own operator< method for the key class/struct for map to know how to order

Associate student with GPA

```
struct stud
{
    string name;
    int idNum;
};

bool operator<(const stud& a, const stud& b)
{
    return (a.name < b.name); //can also compare by ids
```

```
}

int main()
{
    map<stud, float> stud2GPA;
    stud d;
    d.name = "David Smallberg";
    d.idNum = 916451243;

    stud2GPA[d] = 1.3;
}
```

STL Class 4: Set

A set is a container that keeps track of unique items

- All the items are alphabetically ordered

Define a set of integers

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

int main()
{
    set<int> a;
}
```

Insert items into set

```
a.insert(2);
a.insert(3);
a.insert(2) //dup
```

- If insert a duplicate item into set, it is ignored

Get the size of set

```
cout << a.size();
```

Erase a member of the set

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

Set of Custom Data Types

> Need to define operator< for own classes, otherwise compile error

```
struct Course
{
    string name;
    int units;
};

bool operator<(const Course& a, const Course& b)
{
    return (a.name < b.name);
}

int main()
{
    set<Course> myClasses;

    Course lec1;
    lec1.name = "CS32";
    lec1.units = 16;

    myClasses.insert(lec1);
}
```

Search/Iterate Through A Set

Search the STL set using the `find` function and an iterator

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

STL Algorithms

The STL also provides some additional functions that work with many different types of data

`#include <algorithms>`

1. `find()` function can search most STL containers and arrays for a value
2. `set_intersection()` function: compute the intersection of two sorted sets/lists/arrays of data

3. `sort()` function can sort arrays/vectors/list

The `sort` function

- Works on arrays and vectors
- Sort all items in ascending(increasing) order
- To sort: pass in two iterators
    - One to the first item
    - One that points just past the last item you want to sort

Example: Sort Vector

```
int main()
{
    vector<string> n;
    n.push_back("carey");
    n.push_back("bart");
    n.push_back("alex");

    //sort the whole vector
    sort(n.begin(), n.end());

    //sorts just the first 2 items of n
    sort(n.begin(), n.begin() + 2);
}
```

Pass in addresses to sort arrays

```
int arr[4] = {2, 5, 1, -7};

//sorts the first 4 array items
sort(&arr[0], &arr[4]);
```

> Can also use `sort` to order objects based on arbitary criteria

Example: Compare `Dogs`

Sort `Dogs` based on how nasty their bit is first, and how loud their bark is, second

```
class Dog
{
    public:
        int getBark() { return m_barkVolume; }
        int getBite() { return m_bitePain; }
};
```

1. Define a new function that can compare two Dogs, A and B

```
//returns true if dog A should go before dog B
bool customCompare(const Dog& a, const Dog& b)
{
    if (a.getBite() > b.getBite())
        return true; //Dog a has a nastier bite
    if (a.getBite() < b.getBite())
        return false; //Dog b has a nastier bite

    return a.getBark() > b.getBark();
}
```

   - This function must return true if A belongs before B, return false if A belongs after B
   - This function will palce dogs with bigger bite BEFORE dogs with smaller bite

2. Pass in function's address as a parameter to `sort()`

```
int main()
{
    Dog arr[4] = {...}
    sort(arr, arr+4, &customCompare);
}
```

The `find()` function

STL provides a `find` function that works with **vectors/lists**

- They don't have built-in find methods like map & set
- First argument: an iterator that points to where you want to start searching
- Second argument: an iterator that points JUST AFTER where you want to stop searching
- Third argument: searched value
- Return type: an iterator to the item that it found
   - If cannot locate, will return whatever you pass in for the **second parameter**

Find in a list

```
#include <algorithm>

int main()
{
```

```
    list<string> names;
    list<string>:: iterator a, b, itr;

    a = names.begin(); //start here
    b = names.end(); //end here

    itr = find(a, b, "Judy");

    if (itr == b)
        cout << "I failed!";
    else
        cout << "Hello: " << *itr;
}
```

Find in an array

```
int a[4] = { 1, 5, 10, 25 };
int* ptr;

ptr = find(&a[0], &a[4], 19);

if (ptr == &a[4])
    cout << "Item not found!\n";
else
    cout << "Found " << *ptr;
```

- First argument: pass the address of start point
- Second argument: pass the address of the element AFTER the last item in your search
- Will return a pointer to the found item, or to the second parameter if the item can't be found

The `find_if()` function

The `find_if` function loops through a container/array and passes each item to a "predicate function" that you specify

```
bool is_even(int n) //predicate function
{
    if (n % 2 == 0)
        return true;
    else return false;
}

int main()
{
    int a[4] = {1, 5, 10, 25};
    int *ptr;
    ptr = find_if(&a[0], &a[4], is_even);
```

```
    if (ptr == &a[4])
        cout << "No even numbers!\n";
    else
        cout << "Found even num: " <<*ptr;
}
```

- `find_if` processes each item in the container until the predicate function returns true or it runs out of items
- returns an iterator/pointer to the first item that triggers the predicate function

Predicate function

- **must return a boolean value**
- must accept values that are of the same type as the ones in the container/array

> `find_if` provides a convenient way to locate an item in a set/map/list/vector that meets specific requirements

How does `find_if` work?

Use pointers to functions

```
int main()
{
    int (*ptr) (int);

    ptr = sqaured;
    cout << ptr(5); //prints 25

    ptr = cubed;
    cout << ptr(5); //print 124
}
```

- `int (*ptr) (int)`: ptr is a pointer variable that can point to any function that returns an int, and takes a single int as a parameter
- `ptr = squared` points ptr to the squared function
- `ptr(5)` just uses the function like normal

Use function pointers as arguments to functions

```
void applyToArray(int (*ptr)(int), int x[], int size)
{
    for (int i = 0; i < size; i++)
        x[i] = ptr(x[i]);
}
```

```
int arr[3] = { 10, 20, 30 };
applyToArray(cubed, arr, 3);
```

- This is how `find_if` works

Compound STL Data Structures

Example: maintain a list of courses for each UCLA student

Create a map between a student's name and their list of courses

```
map<string, list<Course>> crsmap;
Course c1("cs", "32");
Course c2("math", "3b");
Course c3("english", "1");

crsmap["carey"].push_back(c1);
crsmap["carey"].push_back(c2);

crsmap["david"].push_back(c1);
crsmap["david"].push_back(c3);
```

More examples

Design a compount STL data structure that allows us to associate people (a Person object) and each person's set of friends (also Person objects)

```
class Person
{
    public:
        string getName();
        string getPhone();
};

bool operator<(const Person& a, const Person& b)
{
    return (a.getName() < b.getName());
}

map<Person, set<Person>> facebook;
```

Design a compount STL data structure to associate people with the group of courses they've taken, further associate each course with the grade (a string) they got

```
map<Person, map<Course, string>> x;
```

Aside: Inline Methods

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
- If code defined outside the class declaration -> not inline *unless programmer explicitly says so*
- They make your EXE file bigger

Compiler steps for inline function compilation:

```
class Foo
{
    public:
        void setVal(Item a);
        void printVal()
        {
            cout << "The value is: ";
            cout << m_a << "\n";
        }
    private:
        Item m_a;
};
```

```
int main()
{
    Foo<int> nerd;
    nerd.setVal(5);
    //nerd.printVal(); replaced directly with following
    cout << "The value is: ";
    cout << m_a << "\n";
    nerd.setVal(10);
}
```

- Reduces the amount of jumping around your program must do -> speeds up

Make externally-defined inline method

Add `inline` right before the function return type

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

Now the main function:

```
int main()
{
    Foo<int> nerd;
    nerd.m_a = 5;
    cout << "The value is: ";
    cout << m_a << "\n";
    nerd.m_a = 10;
}
```