

Lecture #9

Generic Programming

Generic Programming!

In this lecture, we're going to learn about
"Generic Programming"

The goal of GP is to build algorithms that are able to **operate** on many different types of **data** (not just a single type).

- Custom Comparison Operators
- Templates
- The Standard Template Library (STL)
- STL Iterators
- STL Algorithms (sort, etc.)

On-your-own Study:

- Inline Functions, Template Exercise, More STL Algorithms

For example, a **sort** function that doesn't just sort **ints** but can sort **strings**, **ints**, **Student objects**, etc.

Or a linked list **class** that doesn't just hold **Students**, but can hold **Students**, **ints**, **Robots**, etc.

Once you define such a generic function or class, you can quickly reuse it to solve many different problems.

Part 1: Allowing Generic Comparisons

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

```
main()
{
    int i1 = 3, i2 = 5;
    if (i1 > i2)
        cout << "i1 is bigger";
    Circ a(5), b(6);
    if (a.radius() > b.radius())
        cout << "a was bigger";
    Dog fido(10), spot(20);
    if (fido.weight() > spot.weight())
        cout << "fido is bigger";
```

Wouldn't it be nice if we could compare objects like circles and dogs just like we compare two integers?

Notice that the way we compare two **dogs** (by **weight**) is different than the way we compare two **circles** (by **radius**).

```
class Dog
{
public:
    bool operator<(const Dog &other) const
    {
        if (m_weight < other.m_weight)
            return true;
        return false;
    }
    int getWeight() const
    {
        return m_weight;
    }
private:
    int m_weight;
}
```

All comparison operators accept **const reference** parameters.
(Leaving const out can cause compile errors!)

Don't forget to make it a **const** function - otherwise it won't work when you compare const objects!

Since I'm defined **inside** the class, I can access private data too, like **m_weight**!

You can define a **comparison operator** for a class/struct like this...
If you like, you can also define your comparison operator **inside** your class...

NOTE: If you define the operator **outside** of the class, it may only use **assignment operator**, only it compares two objects instead of assigning one to another.

You can define **==**, **<**, **>**, **<=**, **>=** and **!=**.

Comparison operators defined **outside** the class have two parameters, one for each of the two operands.
If (a >= b);
cout << "a is >= b\n";

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 return a Boolean value: **true** or **false**. In this example, our function should return **true** if **a > b**, and **false** otherwise.

We can! Let's see how!

Custom Comparison

If you forgot the
const keyword...

```
// the old way
void SwapCircle(Circ &a, Circ &b)
{
    Circle temp;
    temp = a;
    a = b;
    b = temp;
}

void SwapDog(Dog &d1, Dog &d2)
{
    Dog temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    Circle a(5), b(6);
    Dog c(100), d(750);
    SwapCircle(a,b);
    SwapDog(c,d);
}
```

private:
int m_weight;
};

public:
int getWeight() const
{
 if (a.getWeight() >= b.getWeight())
 return(true);
 else return(false);
}

Oh, and by the way... since a and b are
const, our function can only call const
functions in Dog. So you'd better make
getWeight() const too!

You'll see this kind of
cryptic error...
Description: 10 Errors
① error C2661: 'int Dog::getWeight(void)': cannot convert this pointer from 'const Dog' to 'Dog &'.

5
fido weight
if (fido (5) , spot (3) ;
if (fido >= spot)
cout << "Fido wins";
...
Simply using the operator in
your code causes C++ to call
your comparison function!

3
spot weight

The Solution

In C++, we use C++'s "template" feature to solve this problem.

```
template <typename Item>
void swap(Item &a, Item &b)
{
    Item temp;
    temp = a;
    a = b;
    b = temp;
}

// use our templated func
main()
{
    Dog d1(10), d2(20);
    Circle c1(1), c2(5);

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

Now you can use your generic
function with any data type!

Function Template Details

Always place your templated functions in a header file.

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

You must put the ENTIRE
template function in the header
file, not just the prototype!

```
MyCoolProgram.CPP
#include "Swap.h"
void main(void)
{
    int a=5, b=6;
    swap(a,b); // GOOD!
}
```

11

Part 2: Writing Generic Functions

In this code, we've written several different **swap** functions that swap the two values passed into the function.
Wouldn't it be nice if we could write one **swap** function and have it work for **any** data type?

```
// the new way
...
... (we'll learn how in a sec)

main()
{
    Circle a(5), b(6);
    Dog c(100), d(750);
    SwapCircle(a,b);
    SwapDog(c,d);
}

OurGenericSwap(a,b);
OurGenericSwap(c,d);
OurGenericSwap(e,f);
}
```

We can!! Let's see how!

10

Function Template Details

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!

Question: How many versions of our function would be defined in this example?

```
void swap(Dog &x, Dog &y)
{
    Dog temp;
    temp = x;
    x = y;
    y = temp;
}

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

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

Dog a(13), b(41);
swap(a,b);
int p=-1, q=-2;
swap(p,q);
string x("a"), y("b");
swap(x,y);
```

So you can think of templates as a time-saving/bug-reducing/source-simplifying technique rather than one that reduces the size of your compiled program.

Swap.H

Function Template Details

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

MAX.H

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

Function Template Details

And here's a version of **bigger** that is just used for comparing dogs!

```
Dog bigger(Dog &x, Dog &y)
{
    if (x.bark() > y.bark())
        return x;
    else if (x.bark() < y.bark())
        return y;
    // barks are equal, check bite
    if (x.bite() > y.bite())
        return x;
    else
        return y;
}
```

This function call will use the specialized version of bigger just for Dogs. Why? If C++ sees a specialized version of a function, it will always choose it over the templated version.

You may override a templated function with a specialized (non-templated) version if you like.

```
int main()
{
    Circle a, b, c;
    c = bigger(a,b);
    Dog fido, rex, winner;
    winner = bigger(fido,rex);
}
```

Heres a templated function to return the **bigger** of two items.

Function Template Details

You MUST use the template data type (e.g. **Data**) 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]);
}
```

Data was not used to specify the type of any parameters.

GOOD:

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

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

Data used to specify the types of **x and **y**!**

And here's a version of **bigger** that is just used for comparing dogs!

This function call will use the specialized version of bigger just for Dogs. Why? If C++ sees a specialized version of a function, it will always choose it over the templated version.

You may override a templated function with a specialized (non-templated) version if you like.

```
int main()
{
    Circle a, b, c;
    c = bigger(a,b);
    Dog fido, rex, winner;
    winner = bigger(fido,rex);
}
```

Heres a templated function to return the **bigger** of two items.

Function Template Details

And here's a version of **bigger** that is just used for comparing dogs!

This function call will use the specialized version of bigger just for Dogs. Why? If C++ sees a specialized version of a function, it will always choose it over the templated version.

You may override a templated function with a specialized (non-templated) version if you like.

```
int main()
{
    Circle a, b, c;
    c = bigger(a,b);
    Dog fido, rex, winner;
    winner = bigger(fido,rex);
}
```

Heres a templated function to return the **bigger** of two items.

A Hairy Template Example

If your templated function uses a **comparison operator** on templated variables...

```
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);
}

template <typename Data>
void winner(Data &x, Data &y)
{
    if (x > y)
        cout << "first one wins!\n";
    else
        cout << "second one wins!\n";
}
```

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!

Don't forget or you'll **suffer!**

```
main()
{
    int i1=3, 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...
}
```

And **yes**, just in case you were guessing...

You can do this type of thing too...

```
int main(void)
{
    foo(5,"barf"); // OK!
    foo("argh",6); // OK!
    foo(42,52); // OK!
}
```

19 In classes with **externally-defined member functions**, things get **ugly**!

You add the prefix:

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

before the **class definition** itself...

AND before each function definition, **outside** the class.

THEN update the types to use your templated type...

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

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

Finally, place the postfix:

```
>>>
```

Between the **class name** and the **::** in all function defs.

Part 3: Writing Generic Classes

We can use templates to make entire classes generic too:

```
template <typename Item>
class HoldOneValue
{
public:
    void setVal(Item a)
    {
        m_a = a;
    }
    void printTenTimes(void)
    {
        for (int i=0;i<10;i++)
            cout << m_a;
    }
private:
    Item m_a;
};
```

You must use the prefix:

template <typename xxx> before the **class definition** itself...

Then update the appropriate types in your class...

Now your class can hold any type of data you like - just like the C++ **stack** or **queue** classes!

```
main()
{
    HoldOneValue<int> v1;
    v1.setVal(10);
    v1.printTenTimes();
    HoldOneValue<string> v2;
    v2.setVal("ouch");
    v2.printTenTimes();
}
```

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

Template Classes

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

Before

```
template <class HoldMe>
class LinkedList
{
public:
    LinkedList() ;
    bool insert(HoldMe &value) ;
    bool delete(HoldMe &value) ;
    bool retrieve(int i, HoldMe &value) ;
    int size(void) ;
    ~LinkedList() ;
private:
    ...
};

class LinkedListofDogs
{
public:
    LinkedListofDogs() ;
    bool insert(Dog &value) ;
    bool delete(Dog &value) ;
    bool retrieve(int i, Dog &value) ;
    int size(void) ;
    ~LinkedListofDogs() ;
private:
    ...
};
```

After

```
template <class HoldMe>
class LinkedList
{
public:
    LinkedList() ;
    bool insert(HoldMe &value) ;
    bool delete(HoldMe &value) ;
    bool retrieve(int i, HoldMe &value) ;
    int size(void) ;
    ~LinkedList() ;
private:
    ...
};

class LinkedListofStrings
{
public:
    LinkedListofStrings() ;
    bool insert(string &value) ;
    bool delete(string &value) ;
    ...
};

class LinkedListofDogs
{
public:
    LinkedListofDogs() ;
    bool insert(Dog &value) ;
    bool delete(Dog &value) ;
    bool retrieve(int i, Dog &value) ;
    int size(void) ;
    ~LinkedListofDogs() ;
private:
    ...
};
```

Carey's Template Cheat Sheet

- To template a non-class function called bar:
 - Update the function header: int bar(int a) → `template <typename ItemType> ItemType bar(ItemType a);`
 - Replace appropriate types in the function to the new ItemType: { int a; float b; ... } → `[ItemType a; float b; ...]`
- To template a class called foo:
 - Put this in front of the class declaration: class foo { ... } → `template <typename ItemType> class foo { ... }`
 - Update appropriate types in the class to the new ItemType
 - How to update internally-defined methods:
 - For normal methods, just update all types to ItemType: int bar(int a, ...) → `foo::ItemType bar(ItemType a, ...)`
 - Assignment operator: foo operator=(const foo &other) → `foo::operator=(const foo::ItemType &other)`
 - Copy constructor: foo(const foo &other) → `foo::copyConstructor(const foo::ItemType &other)`
 - For each externally-defined method:
 - For non inline methods: int foo:bar(int a) → `template <typename ItemType> ItemType foo::bar(ItemType a)`
 - For inline methods: inline int foo:bar(int a) → `template <typename ItemType> ItemType inline foo::bar(ItemType a)`
 - For copy constructors and assignment operators:
 - For & operator (const foo &other) → `foo::operator=(const foo::ItemType &other)`
 - foo::operator=(const foo &other) → `foo::operator=(const foo::ItemType &other)`
 - Simply replace appropriate internal variables in your struct (e.g., int val) with your ItemType (e.g., `ItemType val;`)
 - If you have an internally defined struct blah in a class: class foo { ... struct blah { int val; }; ... }
 - Simply replace appropriate internal variables in your struct (e.g., int val) with your ItemType (e.g., `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. Just update variables to your ItemType.
 - If an externally-defined method in a class is trying to return an external function bar definitions as follows:
 - Assuming your internal structure is called "blah" update your external function bar definitions as follows:
 - blah::foo:bar(. . .) → `template <typename ItemType> blah foo::ItemTypes bar(. . .)`
 - blah::foo:bar(. . .) → `template <typename ItemType> blah foo::ItemTypes *bar(. . .)`
 - Assuming your internal structure is called "blah" update your external function bar definitions as follows:
 - blah::foo:bar(. . .) → `template <typename ItemType> blah foo::ItemTypes bar(. . .)`
 - blah::foo:bar(. . .) → `template <typename ItemType> blah foo::ItemTypes *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`)

21

Part 4: The Standard Template Library (aka "STL")

The Standard Template Library or **STL** is a collection of **pre-written, tested** classes provided by the authors of C++.

These classes were all **built using templates**, meaning they can be used with many different data types.

You can use these classes in your programs and it'll **save you hours of programming! Really!**

As it turns out, we've already seen two of these STL classes!

The "STL"

We've already seen several STL classes (which are all implemented using templates)

```
#include <stack>
#include <queue>
using namespace std;

main()
{
    stack<int>           is;
    queue<string>        sq;
    ...
    is.push(5);
    is.push(10);
    ...
    sq.push("goober");
    ...
}
```

- The **Stack** and **Queue** classes are both part of the **STL**.
- These classes are called "**container**" classes because they hold groups of items.
- The STL has many more container classes for your use as well!

Let's learn about them...

22

Cool STL Class #1: Vector

25

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 vectors in your program, make sure to `#include <vector>`:

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

main()
{
    vector<string> strs;
    vector<int>   nums;
    vector<Robot> robots;
    vector<int>   geeks(950);
}
```

Cool STL Class #1: Vector

Remember: If you don't include a "using namespace std" command, then you'll need to use the `std::` prefix for all of your STL containers, e.g.:

`std::vector<std::string> strs;`

To create an empty vector (with 0 initial elements) do this...

Or create a vector that starts with N elements like this...

All of a vector's initial elements are automatically initialized/constructed (e.g., `geeks` 950 values start at zero)!

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

main()
{
    vector<string> strs;
    strs.push_back("Carey");
    strs.push_back("Scott");

    vector<int> vals(3);
    vals.push_back(123);
}
```

Cool STL Class #1: Vector

26

To read or change an **existing item** use brackets to access it.

But be careful! You may only use brackets to access existing items!

Finally, you can use the `front` or `back` methods to read/write the first/last elements (if they exist).

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

main()
{
    vector<int> vals(3);
    vals.push_back(123);

    vals[0] = 42;
    cout << vals[3];
    vals[4] = 1971;
    cout << vals[7];
    cout << vals.back();
}
```

There is no item #4 in the vector, so this is illegal!
CRASH!

vals [0] 42
[1] 0
[2] 0
[3] 123
123 123

Cool STL Class #1: Vector

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`.

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

main()
{
    vector<int> vals(3);
    ...
```

This actually **shrinks** the vector (afterward it has fewer items)

Be careful! Once you've removed an item from the vector, you can't access its slot with brackets!

We'll learn how to remove an item from the **middle/front** of a vector in just a bit...

vals [0] 42
[1] 0
[2] 0
[3] 123
123 123

CRASH!

Cool STL Class #1: Vector

For some candy - any guesses how vectors are implemented?

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

main()
{
    vector<int> vals(2, 444);
    vals.push_back(999);

    cout << vals.size(); // 3 I have items!

    if (vals.empty() == false)
        cout << "I have items!" // 444
    cout << arr[10]; // ERROR!
}
```

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!

Carey says: Remember - the `size()` function works for vectors but NOT arrays:
`int arr[10]; cout << arr.size(); // ERROR!`

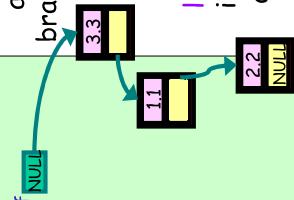
3 I have items!

Cool STL Class #2: List

The STL **list** is a class that works just like a linked list.
(So you can be lazy and not write your own)

So when should you use a **vector** and when should you use a **list**?

Since vectors are based on **dynamic arrays**, they allow fast access to any element (via brackets) but adding new items is often slower.



The STL **list** is based on a **linked list**, so it offers fast insertion/deletion, but slow access to middle elements.

```
#include <list> // ← don't forget!
using namespace std;

main()
{
    list<float> lf;
    lf.push_back(1.1);
    lf.push_back(2.2);
    lf.push_front(3.3);

    if (lf[0] == 1.1) // NULL
        cout << "I have items!" // 33
    for (int j=0;j<lf.size();j++)
        cout << lf[j];
}
```

Like **vector**, the **list** class has **push_back**, **pop_back**, **front**, **back**, **size** and **empty** methods!

But it 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**.

31 Iterating Through The Items

Question: Given an STL container class (like a **list**), how do you **iterate** through its elements?

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

main()
{
    list<int> poof;
    poof.push_back(5);
    poof.push_back(7);
    poof.push_back(1);

    // how do I enumerate elements?
    for (int j=0;j<poof.size();j++)
        cout << poof.retrieve(j);
}
```

Unfortunately, other than the **vector** class which allows you to use **brackets** [] to access elements...

None of the other STL containers have an easy-to-use "**retrieve**" method to quickly go thru the items.

Won't work...

Iterating Through The Items

To enumerate the contents of a container (e.g., a list or vector), you typically use an **iterator variable**.

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

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 (e.g., the first item).

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.

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

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

To define an iterator variable, write the **container type** followed by **two colors**, followed by the word **iterator** and then a **variable name**.

Here are a few more examples:

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

STL Iterators

35

STL Iterators

When you call the **begin()** method it returns the position of the very first item in the container.

How do you use your iterator?

Well, first you must point it at an item in your container... For example, to point your iterator at the first item, simply use the container's **begin()** method.

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

Carey says:

When we use the ***** operator with an iterator, this is called **operator overloading**. The C++ guys realized that you already use the ***** to **dereference pointers**, so why not use it to dereference iterators as well!

```
myVec
[0] 1234 → myVec.begin()
it [ ]
```

```
myVec
[0] 1234 → myVec.begin()
it [ ]
```

1234

1234 5

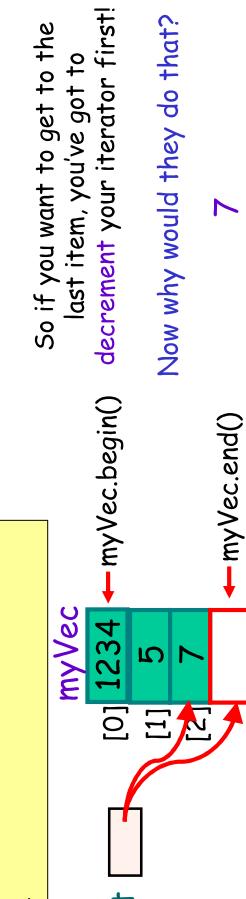
34

STL Iterators

36

```
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);
}
```

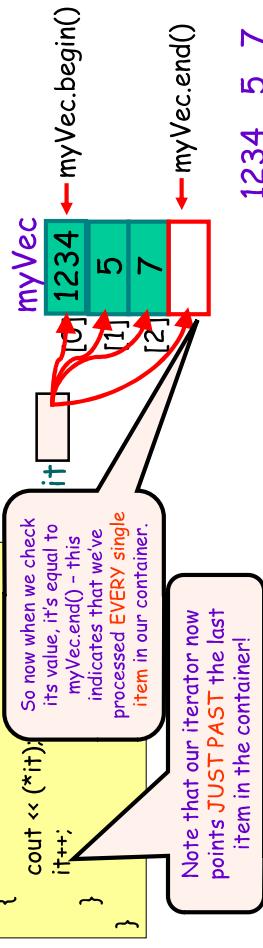
What if you want to point your iterator to the last item in the container?
Well, it's not quite so simple. ☺
Each container has an **end()** method, but it **doesn't** point to the last item!
It points **JUST PAST** the last item in the container...



```
main()
{
    vector<int> myVec;
```

```
    myVec.push_back(1234);
    myVec.push_back(5);
    myVec.push_back(7);
    vector<int>::iterator it;
    it = myVec.begin();
    while ( it != myVec.end() )
    {
        cout << (*it);
        it++;
    }
}
```

That's when you know you're done!



STL And Classes/Structs

38

```
class Nerd
{
public:
    void beNerdy();
    ...
};
```

Of course, you can also create STL containers of **classes** or **structs**!

And here's how you would access the items with an iterator.

You can use the ***** operator and then the dot operator ...

Or you can also use the **->** operator if you like!

```
main()
{
    list<Nerd> nerds;
    Nerd d;
    nerds.push_back(d);
    list<Nerd>::iterator it;
    it = nerds.begin();
    (*it).beNerdy();
    it->beNerdy();
}
```

Const Iterators and Headaches

39

You'll know you made this **mistake** if you see something like this:
error C2440: 'initializing' : **cannot convert from** 'std::_List_iterator<Mylist>' **to** 'std::_List_iterator<Mylist>'

```
void tickleNerds(const list<string> &nerds)
{
    list<string>::const_iterator it; // works!!
    for (it=nerds.begin(); it != nerds.end(); it++)
        cout << *it << " says teehee!\n";
}
```

Sometimes you'll pass a container as a **const reference** parameter...

To iterate through such a container, you **can't** use the **regular iterator**! ☺

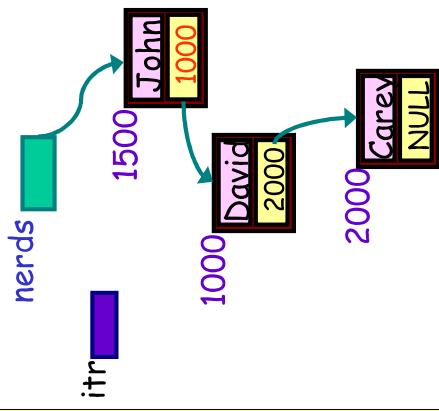
But it's easy to fix. You **just** use a **const iterator**, like this...

STL Iterator Challenge

40

What does it print out?

```
main()
{
    list<string> nerds;
    nerds.push_back("John");
    nerds.push_back("David");
    nerds.push_back("Carey");
    list<string>::iterator itr;
    itr = nerds.begin();
    cout << *itr << endl;
    itr++;
    cout << *itr << endl;
    itr = nerds.end();
    cout << *itr << endl;
}
```



So what is an iterator, anyway? It looks like a pointer, sort of works like a pointer, but it's *not* a pointer!

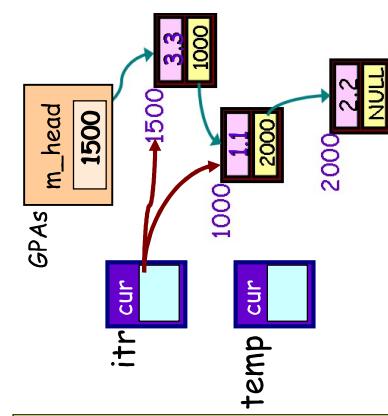
An iterator is an object (i.e. a class variable) that knows three things:

- What element it points to.
- How to find the previous element in the container.
- How to find the next element in the container.

Let's see what this looks like in C++ code!

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

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



43

Other STL Containers

So far we've learned how to use the STL to create **linked lists** and **dynamic arrays** (vectors).

What else can the STL do for us?



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

Cool STL Class #3: Map

```
#include <map>
#include <string>
using namespace std;

main()
{
    map<string, int> name2Phone;
    name2Phone["Carey"] = 8185551212;
    name2Phone["Joe"] = 3109991212;

    The string "Joe" is
    now associated with
    the integer value
    3109991212.
}

The string "Carey"
is now associated
with the integer
value 8185551212.
```

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...

Ok. Names are stored in **string** variables, and phone **#s** in **integers**.

Here's how we create a **map** to do this.

Here's how I **associate** a given string to an integer.
 "Carey" → 8185551212
 "Joe" → 3109991212

Cool STL Class #3: Map

```
#include <map>
#include <string>
using namespace std;

main()
{
    map<string, int> name2Phone;
    name2Phone["Carey"] = 8185551212;
    name2Phone["Joe"] = 3109991212;

    name2Phone[4059913344] = "Ed";
    fones2Names[4059913344] = "Ed";
    fones2Names[8183451212] = "Al";
}

Cool! So how does the
Map class work?
```

A given map can only associate in a single direction...

For example, our name2Phone map can associate a **string** to an **int**, but not the other way around!

So how would we create a map that lets us associate **integers** → **strings**?

If you want to **efficiently** search in both directions, you have to use two maps.

Cool! So how does the Map class work?

How the Map Class Works

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

map<string, int> name2Age;
```

The map class basically stores each association in a **struct** variable! Let's see how

```
#include <map>
#include <string>
using namespace std;

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

```
name2Age["Carey"] = 40;
name2Age["Dan"] = 22;
name2Age["David"] = 53;

name2Age["Carey"] = 39; // @
```

```
cout << (*it).first; // cout << it->first;
cout << (*it).second; // cout << it->second;
}
```

name2Age

name2Age

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

```
it = name2Age.find("Dan");
```

Then you can look at the pair of values pointed to by the iterator!

```
Note: You can only search based on the left-hand type!
```

```
cout << (*it).first; // cout << it->first;
cout << (*it).second; // cout << it->second;
}
```

Dan 22

How to Search the Map Class

```
#include <map>
#include <string>
using namespace std;

main()
```

first	Carey
second	39
first	Dan
second	22
first	David
second	53

name2Age

name2Age

map<string, int>; it;

it = name2Age.find("Dan");

Then you can look at the pair of values pointed to by the iterator!

Note: You can only search based on the left-hand type!

cout << (*it).first; // cout << it->first;
cout << (*it).second; // cout << it->second;
}

Of course, you can use the alternate → syntax if you like too!

How to Search the Map Class

48

```
#include <map>
#include <string>
using namespace std;

main()
{
    map<string, int> name2Age;
    ...
    map<string, int>::iterator it;
    it = name2Age.find("Ziggy");
    if ( it == name2Age.end() )
    {
        cout << "Not found! \n";
        return;
    }
    cout << it->first;
    cout << it->second;
}
```

But be **careful!**
What if the item you search for (e.g. "Dan") isn't in your map? You've got to check for this case!

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!
We can check for and handle this!

Not found()

How to Iterate Through a Map

49

```
#include <map>
#include <string>
using namespace std;

main()
{
    map<string, int> name2Age;
    ...
    map<string, int>::iterator it;
    for ( it = name2Age.begin() ;
          it != name2Age.end() ;
          it++)
    {
        cout << it->first;
        cout << it->second;
    }
}
```

Carey 39 Dan 22 And so on...
name2Age.end()

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

Carey says: As it turns out, the map **always** maintains its items in **alphabetical order!** This means that when you iterate thru them, they're automatically ordered for you! (i.e., no sorting required!)

```
struct stud // student class
{
    string name;
    int idNum;
};

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

stud d;
d.name = "David Smallberg";
d.idNum = 916451243;
stud2GPA[8183451234] = d;
```

In this case, the left-hand side is a `stud`. Therefore, for this to work we must define an `operator<` method for `stud`.

You can even associate more complex data types like **structs** and **classes**.

For example, this code allows us to associate a given `Student` with their `GPA`!
But for this to work, you **must** define your own `operator<` method for the left-hand class/struct!

```
map<stud, float> stud2GPA;
```

We define the `operator<` to allow our map to differentiate two different students from each other. (Right now, you might be asking:
"Why not use `operator==` instead?" We'll learn why in a few lectures)

Cool STL Class #3: Map

51

Note: You only need to define the `operator<` method if you're mapping **from** your own struct/class (its on the **left-hand-side** of the map).

In this case, our `student` struct is on the right-hand-side, so we don't need to define an `operator<` method for it.
(Unless you're feeling nerdy.)

```
struct stud // student class
{
    string name;
    int idNum;
};

main()
{
    map<int, stud> phone2Stud;
```

```
stud d;
d.name = "David Smallberg";
d.idNum = 916451243;
stud2GPA[8183451234] = d;
```

Cool STL Class #4: Set

53

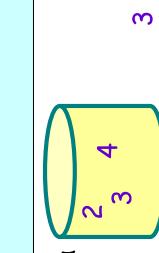
Cool STL Class #4: Set

A **set** is a container that keeps track of **unique items**.

Our set already contains the value of 2, so this is ignored.

Here's how you define a **set** of integers.

Here's how you **insert** items into a **set**.
If you insert a **duplicate** item into the **set**, it is ignored (since it's already in the **set**!).



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

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

int main()
{
    set<Course> myClasses;

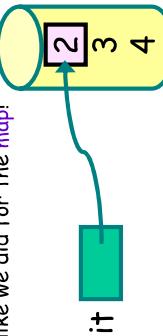
    Course lec1;
    lec1.name = "CS32";
    lec1.units = 16;

    myClasses.insert(lec1);
}
```

Here's how you get the **size** of a **set**.
Finally, here's how you **erase** a member of the **set**.

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**!



I found 2

BTW, you can iterate through a **set**'s items just like we did with a **map** - and the items will all be **alphabetically ordered**!

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

main()
{
    set<int> a;
    a.insert(2);
    a.insert(3);
    a.insert(4);

    set<int>::iterator it;
    it = a.begin();
    while (it != a.end())
    {
        cout << "I found " << (*it);
        cout << endl;
    }
}
```

Deleting an Item from an STL Container

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

First you search for the item you want to delete and 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())
    {
        // found my item!
        cout << "bye bye " << *it;
        geeks.erase(it); // kill
    }
}
```

And of course, you can have sets of other data types as well!

But as with our map, you **need** to define the **operator<** for your own classes (e.g., **Course**).

Otherwise you'll get a compile error! ☹

Iterator Gotchas!

* For more details, see:
http://en.cppreference.com/w/cpp/container/Sequence_containers

57

Deletion Gotchas

```
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"); // add
    x.erase(it); // kill 1st item
    cout << *it; // ERROR!
}
```

Leaving the old iterator pointing to a **random spot** in your PC's memory.

Let's say you point an iterator to an item in a **vector**...

And then you either **add** or **erase** an item from the same vector...

You should assume that your **iterator** will become **invalid!***

I'm no longer valid!! ☹

Why? When you **add/erase** items in a **vector** it may **shuffle** its memory around (without telling you) and then your iterators **won't point to the right place** any more!

Fortunately, this same problem doesn't occur with **sets, lists or maps**.

With one exception...

If you erase the item the iterator points to, then you've got troubles!

```
main()
{
    set<string> s;
    s.insert("carey");
    s.insert("rick");
    s.insert("alex");
    set<string>::iterator it;
    it = s.find("carey");
    s.insert("yong"); // add
    s.erase("rick"); // removes rick
    cout << *it; // still works!
}
```

Part 5: STL Algorithms

See: <http://en.cppreference.com/w/cpp/algorithm>



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

For instance, the **find()** function can **search most STL containers** and **arrays** for a value.

And the **set_intersection** function can compute the **intersection** of two **sorted sets/lists/arrays** of data.

And the **sort()** function can **sort arrays/vectors/lists** for you!

Let's learn about the **sort()** function!

The sort() function

First, to use the STL **sort()** function, or any of its other algorithms, you need to include **this** header file.

```
#include <vector>
#include <algorithm>
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 );
}
```

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.

You can similarly pass in **addresses** to sort **arrays**!

Finally, you can use **sort()** to **order objects** based on your own **arbitrary criteria!**

```
int arr[4] = {2, 5, 1, -7};
// sorts the first 4 array items
sort( &arr[0], &arr[4] );
}
```

58

The find() function

The STL provides you with a **fast sorting function** which works on **arrays** and **vectors**! It will sort all of the items in **ascending (increasing) order**.

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.

You can similarly pass in **addresses** to sort **arrays**!

Finally, you can use **sort()** to **order objects** based on your own **arbitrary criteria!**

59

