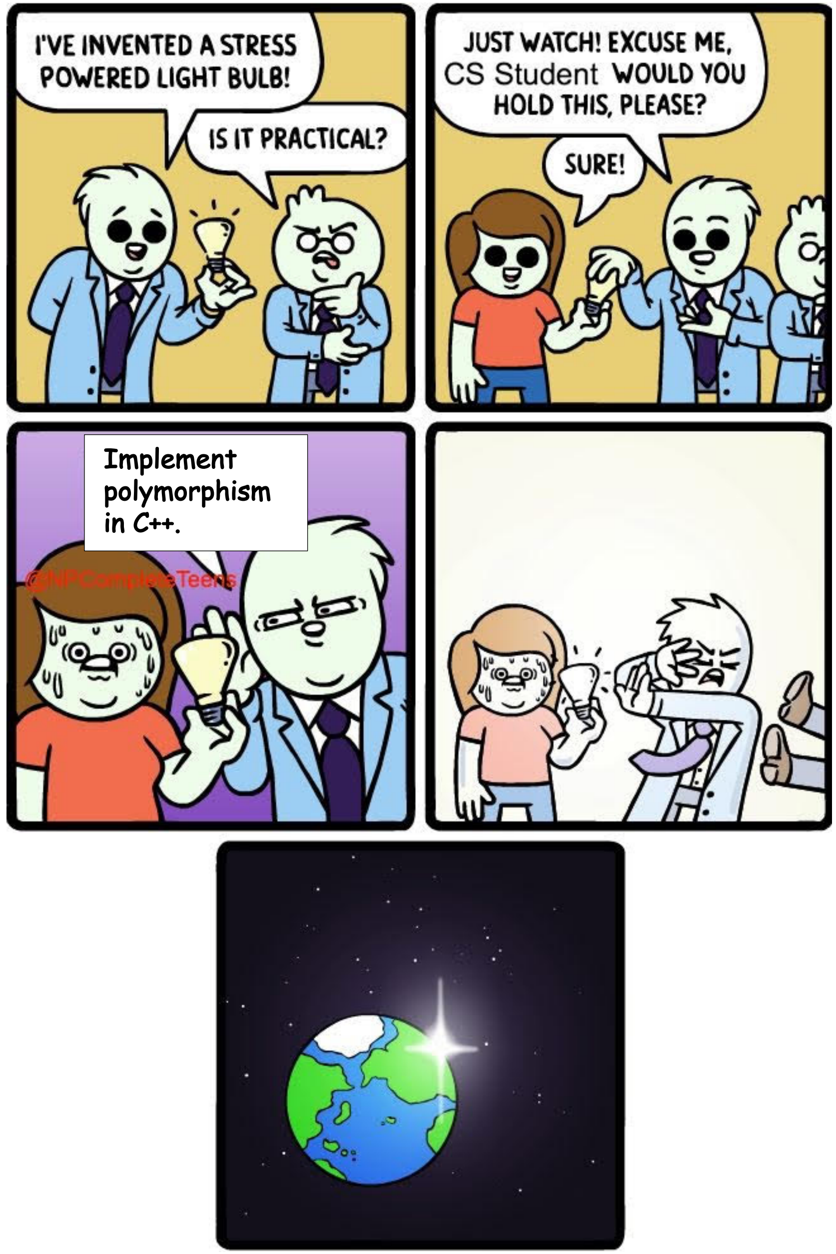
# Lecture #9

## Generic Programming!


- 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


I'VE INVENTED A STRESS POWERED LIGHT BULB!
IS IT PRACTICAL?
JUST WATCH! EXCUSE ME, CS Student WOULD YOU HOLD THIS, PLEASE?
SURE!
Implement polymorphism in C++.
@NPCompleteTeens

# Generic Programming
## What's the big picture?

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

Like a generic sorting function that can sort an array holding ANY type of value

Or a generic linked list class that can hold ANY type of variable

```
int main() {
   list<int> list_of_integers;
   list<string> list_of_strings;

   list_of_integers.push_back(42);
   list_of_strings.push_back("It's LIT!");
}
```

Create a linked list holding any type of data in ONE LINE!

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

**Uses:**

Used in virtually every C++ program to speed up development – search engines, video games, etc.

# Part 1: Allowing 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";

   Circ   a(5), b(6);

   if (a > b)
     cout << "a was bigger";

   Dog fido(10), spot(20);

   if (fido > spot)
    cout << "fido is bigger";

}
```

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

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

We can! Let's see how!

```cpp
class Dog
{
public:
  bool operator<(const Dog &other) const
  {
    if (m_weight < other.m_weight)
       return true;
    return false; // otherwise
  }

  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; // otherwise
}
```

## Custom Comparison Operators

- You can define comparison operators for a class/struct!
- Here are two examples, one defined inside your class declaration and one outside, so you can see both versions.
- Do these look familiar? They're just like an assignment operator, only they compare two objects instead of assigning one to another.
  You can define ==, <, >, <=, >= and !=
- 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, but you could have compared bite or bark too
- 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.
- Comparison operators defined outside the class have two parameters, one for each of the two operands (see operator >= left)

  `if (a >= b)  cout << "a is >= b\n";`
- Comparison operators defined inside the class have a single "other" parameter, just like a copy constructor does. See operator < above-left. The "other" parameter refers to the value to the right of the < operator:

  *if (a < other) cout << "a is less than the item right of < ";*
- If a comparison operator is defined inside your class declaration, it may access private members (see how operator< accesses m_weight)
- But if the operator is defined outside your class, then it may only use public member functions to compare objects. See how operator>= uses getWeight() rather than accessing m_weight, which is private.
- All comparison operators must take ***const reference*** parameters. And don't forget to make your member comparison operators const (see the const after operator<(…) const;

# Custom Comparison Operators

## And here's how they work!

Oh, and by the way… since a and b are const, our comparison function can only call const functions in Dog! So you'll need to make sure functions like getWeight() are const in your class, e.g.:

```
class Dog {
public:
   int getWeight() const;
};
```

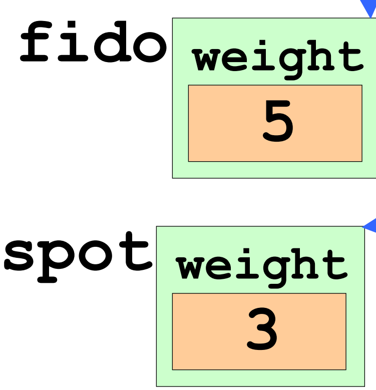
```
bool operator>=(const Dog &a, const Dog &b)
{
  if (a.getWeight() >= b.getWeight())
     return(true);
  else return(false);
}
```

10 Errors | 0 Warnings | 0 Messages

Description

1 error C2662: 'int Dog::getWeight(void)' : cannot convert 'this' pointer from 'const Dog' to 'Dog &'

If you forget to make functions like getWeight() const, you'll see this kind of cryptic error…




Simply using the operator in your code causes C++ to call your comparison function!

```
int main()
{
  Dog fido(5), spot(3);
  if (fido >= spot)
     cout << "fido wins";

  …

}
```

# Part 2: Writing Generic Functions

```
// 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);
}
```

- In the code to the left, 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, like the code below?
- Let's see how!

```
// the new way

... (we'll learn how in a sec)

int main()
{
  Circ a(5), b(6);
  Dog c(10), d(75);
  int e = 5, f = 10;

  OurGenericSwap(a,b);
  OurGenericSwap(c,d);

  OurGenericSwap(e,f);
}
```

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

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

To turn any function into a "generic function," do this:

1. Add the following line above your function:

   template <typename poop>

2. Then use xxx as your data type throughout the function:

   swap(xxx a, xxx b)

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

# Function Template Details

*Always* place your templated functions in a header file, as shown below and right.

Then include your header file in your CPP file(s) to use your function!

You must put the ENTIRE template function body in the header file, not just the prototype, or you'll get an error.

See below for what not to do!

**Swap.H**

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

ERROR!

**Swap.H**

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

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

**MyCoolProgram.CPP**

```
#include "Swap.h"
int main()
{
  int a=5, b=6;

  swap(a,b); // GOOD!
}
```

# Function Template Details

- Each time you use a template function with a different type of variable (e.g., int, string, Dog, …), the compiler generates a new version of the function in your program to handle that type of variable!
- So if we define the swap function below, and used it with the main function shown, C++ will actually generate all of the code on the right and then compile it!
- 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**

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

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

```
#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);
```

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

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

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

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

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

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); // ????
```

# 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!

**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!

**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.

# 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**

```
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;
    e = max(d,c);     // ERROR!
}
```

# A Hairy Template Example

```
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";
}
```

- If your templated function uses a comparison operator on templated variables then C++ expects that all variables passed in will have that operator defined.
- In the example below, we compare Dogs, Circles and ints using the > operator, so we must have operator> defined for Dogs and Circles for this code to work.
- Since C++ automatically provides > for ints, you don't need to (nor are you allowed to) define your own operator> for ints.
- If you forget to define a required operator, you'll get an error when your templated function tries to compare two items.

```
int 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...
}
```

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

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

You can do this type of thing too...

# 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()
   {
     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!

```
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!

```
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";
}
```

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 your templated type…

Finally, place the postfix:

`<xxx>`

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

# Template Classes

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

```
#include "linkedlist.h"
int main( )
{
  Dog fido(10);

  LinkedList<Dog> dogLst;
  dogLst.insert(fido);

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

```
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:
   …
};
```

# Carey's Template Cheat Sheet

- To templatize 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 templatize 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) { … } → ItemType bar(ItemType a) { … }
    - Assignment operator: foo &operator=(const foo &other) → foo<ItemType>& operator=(const foo<ItemType>& other)
    - Copy constructor: foo(const foo &other) → foo(const foo<ItemType> &other)
  - For each externally defined method:
    - For non inline methods: int foo::bar(int a) → template <typename ItemType> ItemType foo<ItemType>::bar(ItemType a)
    - For inline methods: inline int foo::bar(int a) → template <typename ItemType> inline ItemType foo<ItemType>::bar(ItemType a)
    - For copy constructors and assignment operators
    - foo &foo::operator=(const foo &other) → foo<ItemType>& foo<ItemType>::operator=(const foo<ItemType>& other)
    - foo::foo(const foo &other) → foo<ItemType>::foo(const foo<ItemType> &other)
  - 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 internal struct (or a pointer to an internal struct):
    - Assuming your internal structure is called "blah", update your external function bar definitions as follows:
    - blah foo::bar(…) { … } → template<typename ItemType>typename foo<ItemType>::blah foo<ItemType>::bar(…) { … }
    - blah *foo::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)

# 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;

int 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...

# Cool STL Class #1: 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 automagically when you add/remove items.

```
#include <vector>


int main()
{
  std::vector<string>   strs;
  std::vector<int>      nums;
  std::vector<Robot>  robots;

  std::vector<int>  geeks(950);




}
```

To use vectors in your program, make sure to #include <vector>!

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

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

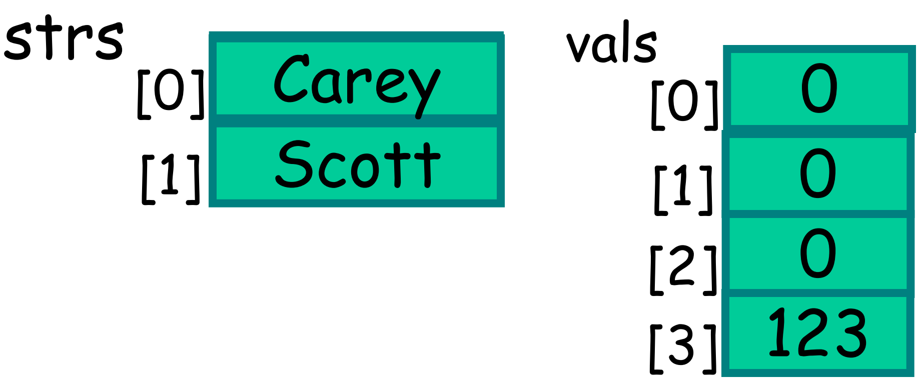
All of a vector's initial elements are automatically initialized/constructed (e.g., geeks 950 values start at zero)!

# Cool STL Class #1: Vector

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

int main()
{
  vector<string>    strs;

  strs.push_back(“Carey”);
  strs.push_back(“Scott”);


  vector<int>  vals(3);

  vals.push_back(123);



}
```

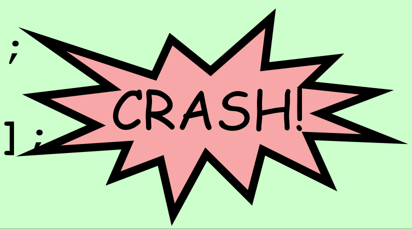
Once you've created a vector, you can add items, change items, or remove items…

To add a new item to the very end of the vector, use the push_back command.

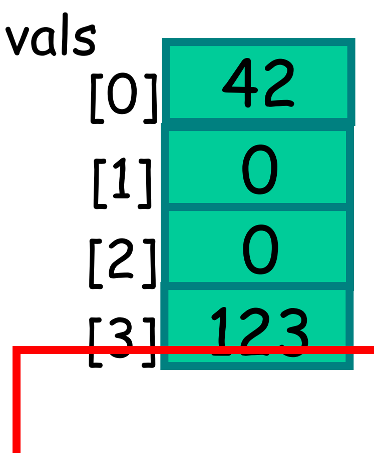


# Cool STL Class #1: Vector

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

int main()
{

  vector<int>  vals(3);

  vals.push_back(123);
  cout << vals.back(); // prints 123

  vals[0] = 42;

  cout << vals[3];

  vals[4] = 1971;

  cout << vals[7];
}
```

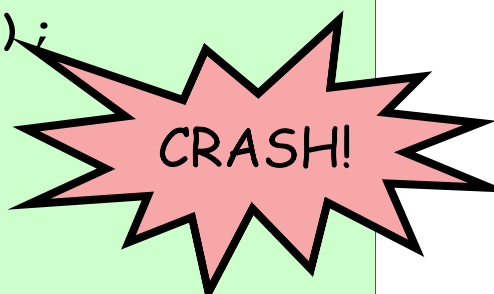



- To read or change an existing item from a vector, use brackets to access it.
- You can use the front or back methods to read/write the first/last elements (if it's not empty).
- But be careful! You may only use brackets to access existing items!
- In this example, our vector has 4 items (the initial three values and 123 which was pushed on the end). So you can only access entries 0 - 3.
- Nor can you add new items past the end of the vector by just setting them, e.g: vals[1000] = 10;
- This will cause a crash, since our vector only has 4 items.



# Cool STL Class #1: Vector

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

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

  ...

  vals.pop_back();
  vals.pop_back();

  vals[3] = 456;


}
```

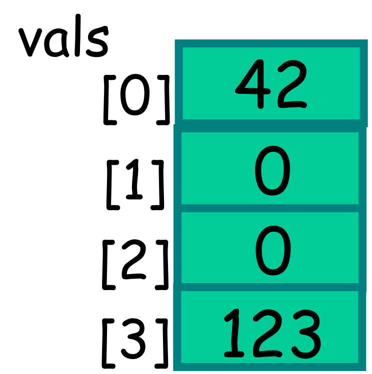



To remove an item from the back of a vector, use pop_back.

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…



# Cool STL Class #1: Vector

- Vectors are implemented using dynamically-allocated arrays with new/delete.
- The vector will initially have just a few allocated slots, with an official size of 0 items.
- Once you fill up those slots by adding values, the vector object will allocate a bigger array, copy all of its current items over to the new array, and then delete the old, smaller array.
- So once in a while, adding an item could take a looooong time. since it will not just add the one item, but move everything over to a new array.
- But on average, adding an item to a vector is pretty fast, since this copying operation is amortized across many additions, most of which just take one step.

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

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

  vals.push_back(999);

  cout << vals.size();

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

}
```

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!
```

# 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)

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

int main()
{
   list<float>     lf;

   lf.push_back(1.1);
   lf.push_back(2.2);
   lf.push_front(3.3);

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

}
```

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!

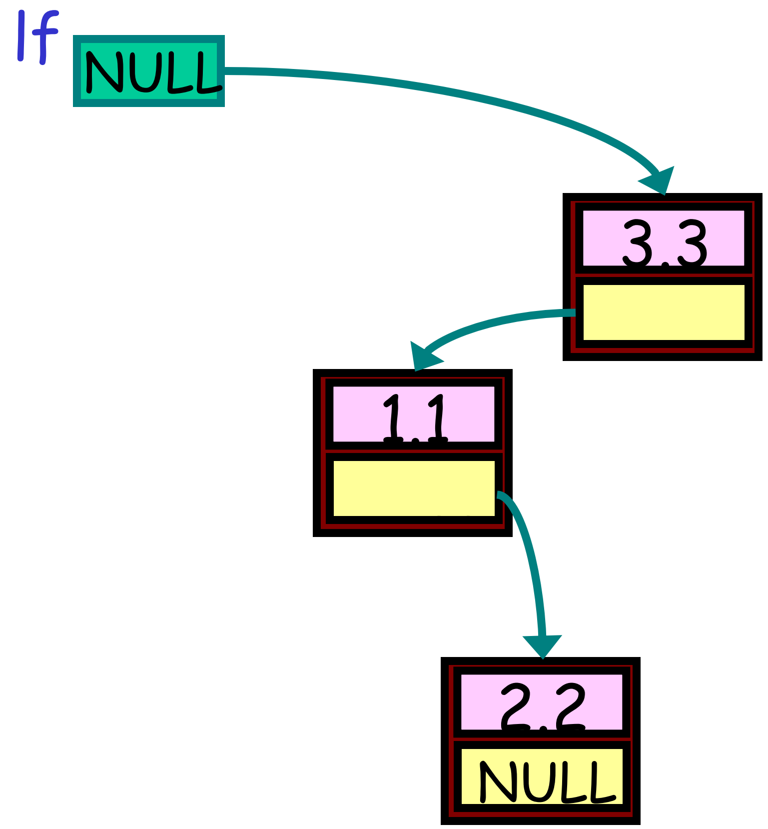
These methods allow you to add/remove items from the front of the list!

Unlike vectors, you can't access list elements using brackets.

# Cool STL Class #2: List

- 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.
- Don't forget to put std:: in front of your list definition, e.g., std::list<float> unless you first add a "using namespace std;" command, as shown below.

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

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



}
```



# 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;

int 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.

```
int 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.

# Defining an Iterator

```
int 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 colons, followed by the word iterator and then a variable name.

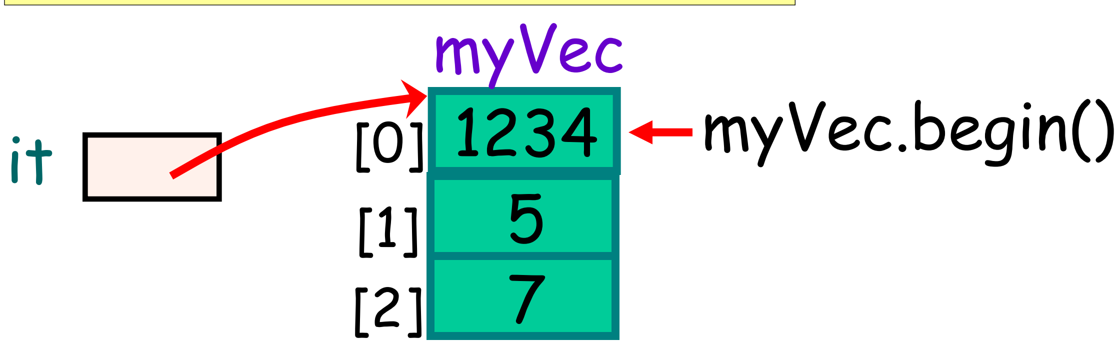
Here are a few more examples:

```
vector<string>::iterator it2;

list<float>::iterator it3;
```

# STL Iterators

```
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) ;



}
```

- How do you use your iterator?
- Iterators start out pointing at nothing, so before using one 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. When you call the begin() method it returns the position of the very first item in the container.
- Once the iterator points at a value, you can use the * operator with it to access the value.
- 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!



# STL Iterators

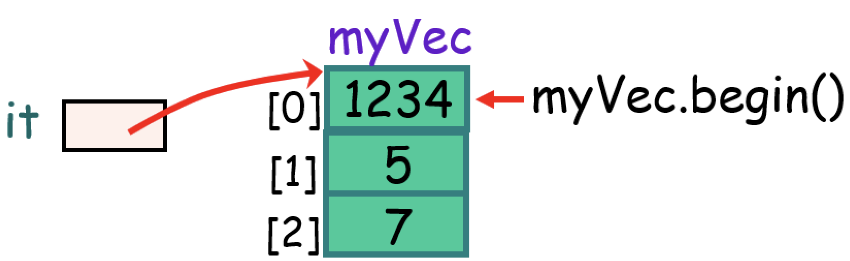
```
int main()
{
  vector<int>       myVec;

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

   vector<int>::iterator  it;

   it = myVec.begin();  // #1

   cout << (*it) ;          // prints 1234

   it++;                    // #2

   cout << (*it) ;          // prints 5
}
```

You can move your iterator down one item by using the ++ operator!

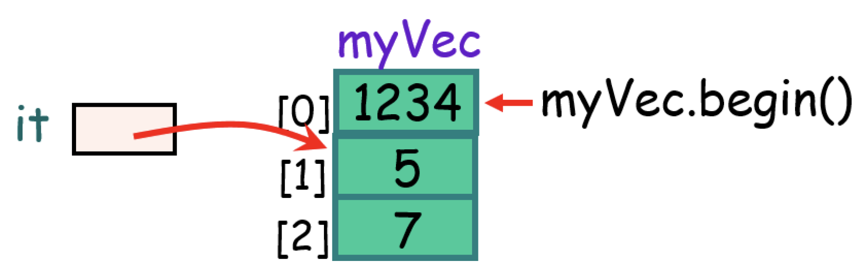
Now the iterator points to the second item!

In a similar way, you can use the -- operator to move the iterator backward!

After line #1




After line #2

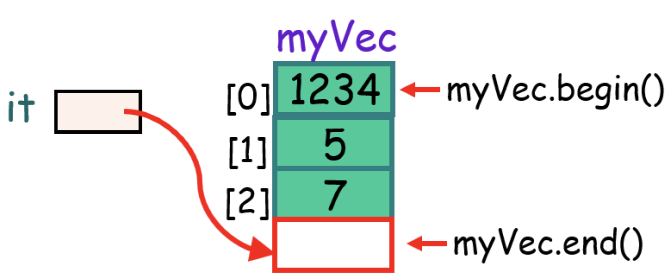


# STL Iterators

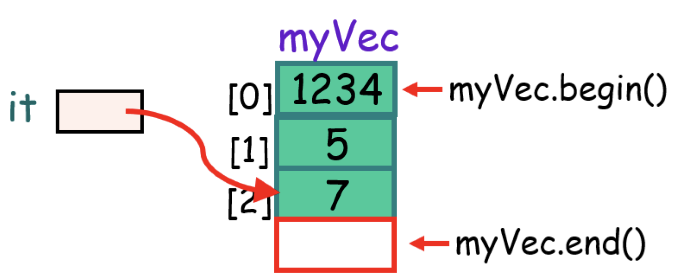
```
int main()
{
  vector<int>       myVec;

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

   vector<int>::iterator  it;

   it = myVec.end();   // #1

   it--;               // #2

   cout << (*it);      // Prints 7

}
```

- 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 first point it at end() and then decrement it!
- Now why would they do that – we'll see in the next slide!

After line #1




After line #2



# STL Iterators

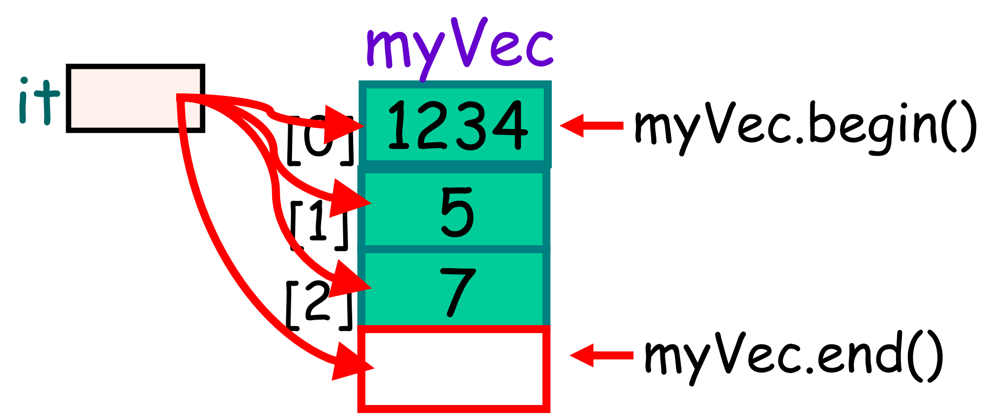
```
int main()
{
  vector<int>       myVec;

   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++;
   }
}
```

1234   5   7

So you can make loops, of course!

When you loop through a container, you don't want to stop at the last item, you want to stop once you've gone JUST PAST the last item!

That's when you know you're done!



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

# STL And Classes/Structs

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

```
int main()
{
  list<Nerd>        nerds;

  Nerd d;

  nerds.push_back(d);

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

  it->beNerdy();

}
```

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!

# Const Iterators and Headaches

You'll know you made this mistake if you see something like this:

```
error C2440: 'initializing' : cannot convert from
'std::_List_const_iterator<_Mylist>' to 'std::_List_iterator<_Mylist>'
```

```
void tickleNerds(const list<string> & nerds)
{
   list<string>::iterator it;   // won't work
   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);
}
```

Sometimes you'll pass a container as a const reference parameter…

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

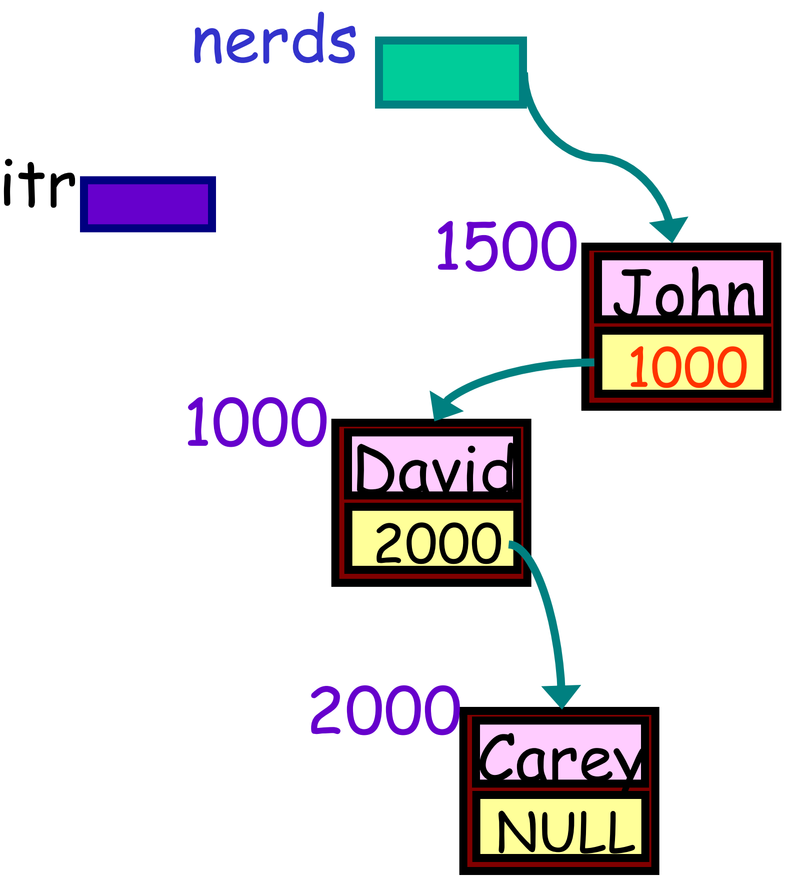
But it's easy to fix. You just use a const iterator, like this…

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

# STL Iterator Challenge

```
int 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();
  itr--;
  cout << *itr << endl;
}
```

What does it print out?



# STL Iterators

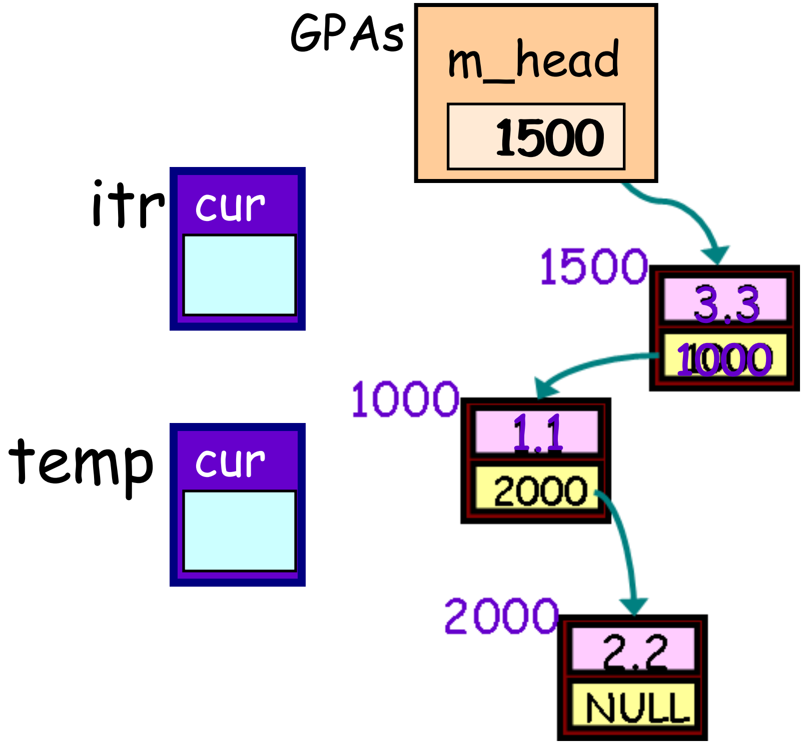
So what is an iterator, anyway? It looks like a pointer, sort of works like a pointer, but it's *not* a pointer!

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;
};
```




- This is obviously a simplification, but it gives you the general idea of how things work.

```
int main()
{
  LinkedList GPAs; // list of GPAs

  ...

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

# 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?

# Cool STL Class #3: Map

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

int main()
{
   map< string, int >    name2Fone;

   name2Fone["Carey"] = 8185551212;
   name2Fone["Joe"] = 3109991212;
}
```

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;

int main()
{
  map<string , int >   name2Fone;

  name2Fone["Carey"] = 8185551212;
  name2Fone["Joe"] = 3109991212;

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

  map<int,string> fones2Names

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

}
```

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!

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?

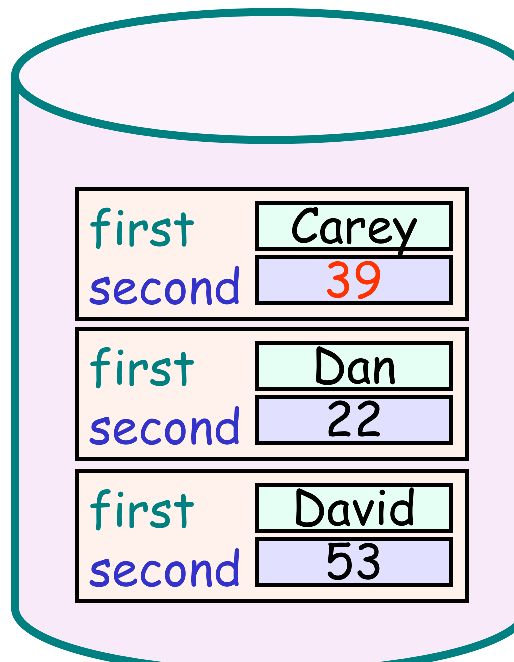
# How the Map Class Works

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

};
```

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

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

  name2Age["Carey"] = 49;
  name2Age["Dan"] = 22;
  name2Age["David"] = 53;

  name2Age["Carey"] = 39; // ☺




}
```

- The map class basically stores each association in a struct variable!
- It uses your two types (in this case string and int) to create a new struct type with two fields, named first and second.
- The "first" variable has the left type in your map, string in this example. The "second" variable has the second type, int in this example.
- Then when you associate a new item, as we do when we associate Dan with 22, it adds a new struct variable with these values to its data structure
- *As you can see, you can replace earlier associations with new ones. See how we reassigned Carey to 39 from its original value of 49.*

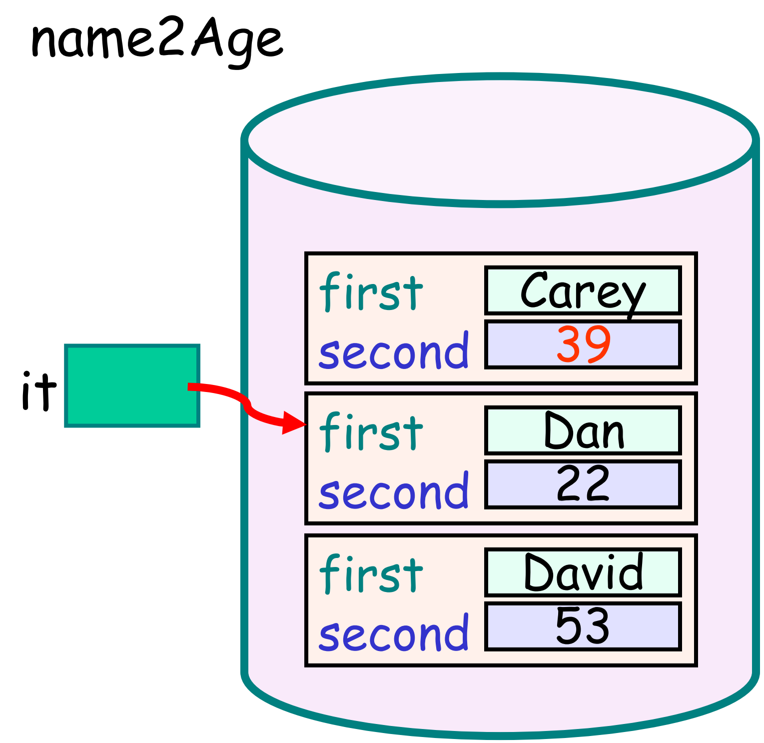
name2Age



# How to Search the Map Class

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

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

  it = name2Age.find("Dan");



 cout << (*it).first;
 cout << (*it).second;
 cout << it->first;  // same thing
 cout << it->second; // as above
}
```

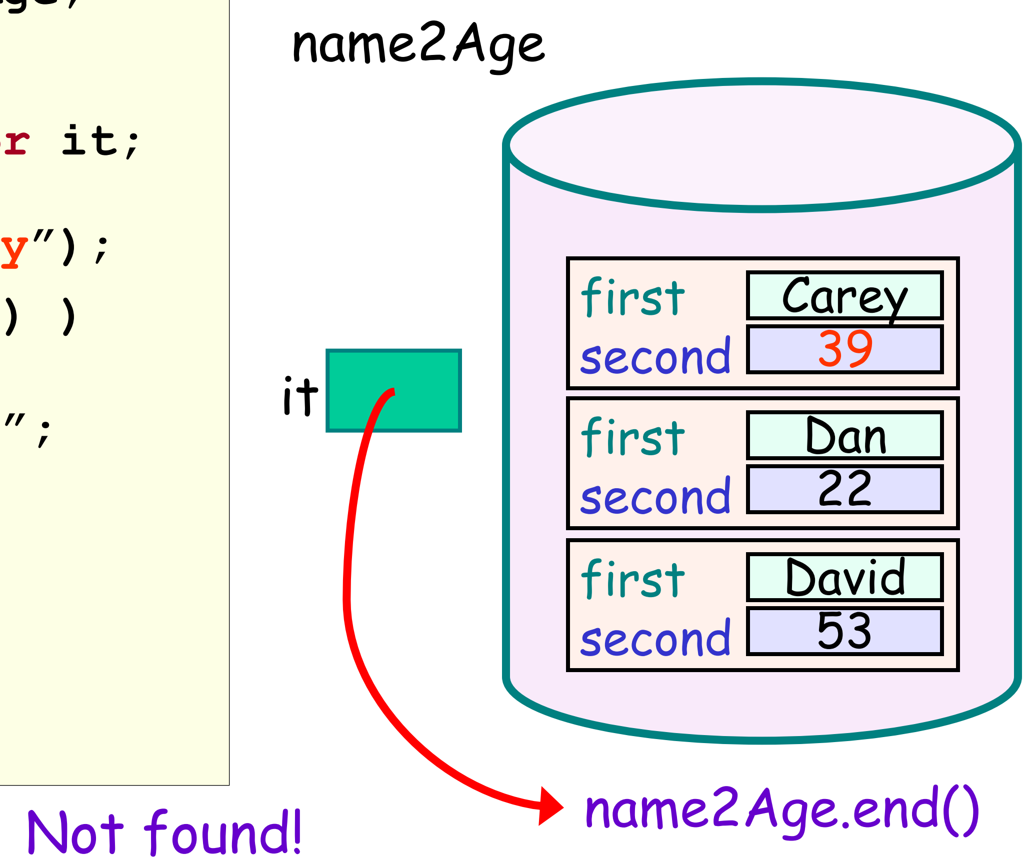
- To search a map for an association, you must first define an iterator to your map, as we do with the "it" variable to the left.
- Then you can call the map's find command in order to locate an association.
- Once we find our item, we can use the dot or -> operator to print out the mapped-from (first) and mapped-to (second) items.
- Note: You can only search efficiently based on the left-hand type (e.g., string)! So we can't do:
  it = name2Age.find(39);



# How to Search the Map Class

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

int 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;
}
```

- What if the item you search for (e.g. "Ziggy") 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!

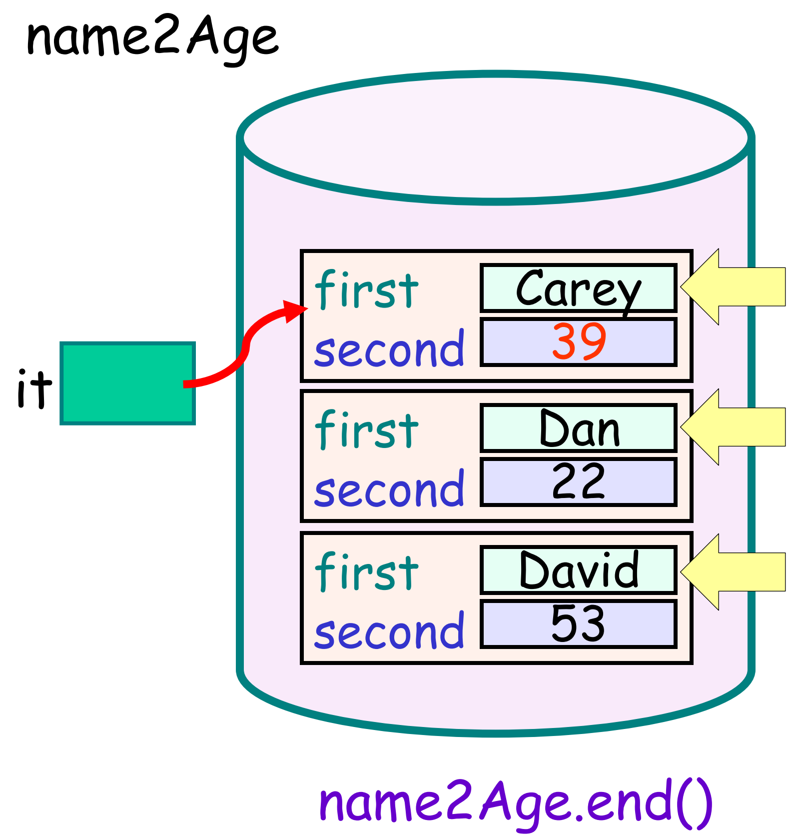


# How to Iterate Through a Map

```
#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;
  }
}
```

Carey 39
Dan 22
David 53

- To iterate through a map, simply use a for/while loop as we did for vectors/lists!
- As it turns out, the map always maintains its items in alphabetical order!
- Entries are ordered by the "first" item, so in this case, by the strings: Carey, Dan and David.
- This means that when you iterate thru them with an iterator, they're automatically ordered for you! (i.e., no sorting required!)



# Cool STL Class #3: Map

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


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


int main()
{
   map<stud,float>  stud2GPA;

   stud d;
   d.name = "David Smallberg";
   d.idNum = 916451243;

   stud2GPA[d] = 1.3;
}
```

- 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! Why the < operator?!?!
- In this case, the left-hand side type is a stud (Student), therefore, for this to work we must define an operator< method for stud (shown to the left)
- We define the operator< to allow our map to differentiate different items
- Right now, you might be asking: "Why not use operator== instead?"
- Well, if you think about it, defining the < operator is sufficient to do everything that defining the == operator can do and more.
- In this example, we differentiated students by their names in operator<, but we could just have easily used their student ID, phone number, etc. to identify unique students.

# Cool STL Class #3: Map

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


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


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

  stud d;
  d.name = "David Smallberg";
  d.idNum = 916451243;

  stud2GPA[8183451234] = d;
}
```

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

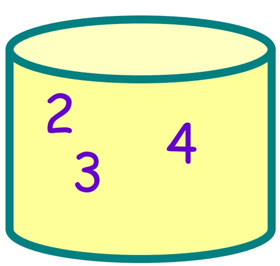
In this case, our student struct is on the right-hand-side, so we don't need to define an operator< method for it.

Since there's already a less-than operator built in to C++ to compare integers, this phone2Stud definition will work as-is.

# Cool STL Class #4: Set

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

int main()
{
  set<int>      a;
  a.insert(2);
  a.insert(3);
  a.insert(4);
  a.insert(2); // dup
  cout << a.size();

  a.erase(2);

}
```

a




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

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!).

Use .size() to get the size of a set.

Use .erase() to erase a member of the set.

# Cool STL Class #4: 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);
}
```

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

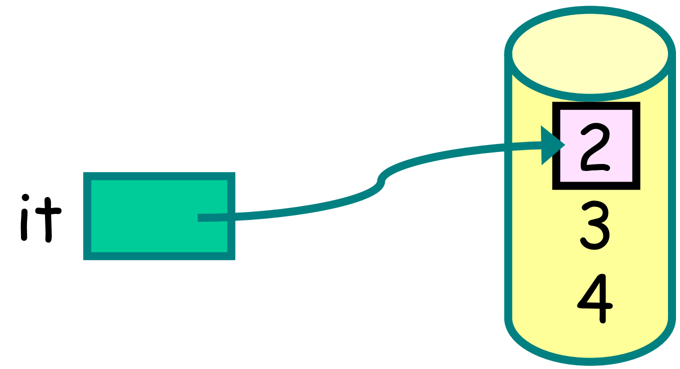
But as with our map, you need to define the operator< for your own classes (e.g., Course)!

Otherwise you'll get a compile error! ☹

# Searching/Iterating Through a Set

```
#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);
}
```

We can search the STL set using the find function and an iterator, just like we did for the map!




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

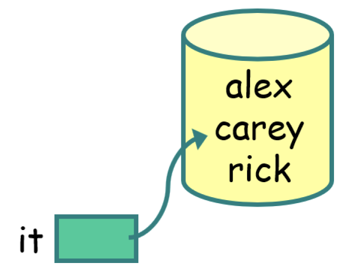
```
it = a.begin();
while (it != a.end())
{
  cout << *it; // alpha order
  it++;
}
```

# Deleting an Item from an STL Container

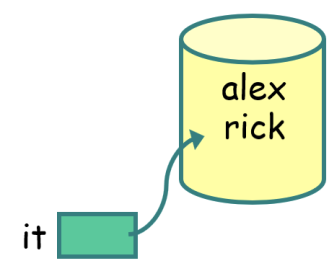
```
int main()
{
   set<string>  geeks;

   geeks.insert("carey");
   geeks.insert("rick");
   geeks.insert("alex");

   set<string>::iterator it;

   it = geeks.find("carey"); // #1

   if (it != geeks.end())
   {
     // found my item!!
     cout << “bye bye “ << *it;
     geeks.erase(it);  // #2
   }
}
```

- *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.

**After line #1**




**After line #2**




bye bye carey

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

# Iterator Gotchas!

```
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

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

I'm no longer valid!!! ☹

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

If you add an item anywhere to the vector you must assume your iterator is invalidated!

And if you erase that item or an item that comes before it, your iterator is also invalidated!

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

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

# Deletion Gotchas

```
int 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 yong
  s.erase("rick"); // removes rick

  cout << *it; // prints "carey"!

  s.erase("carey"); // removes carey

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

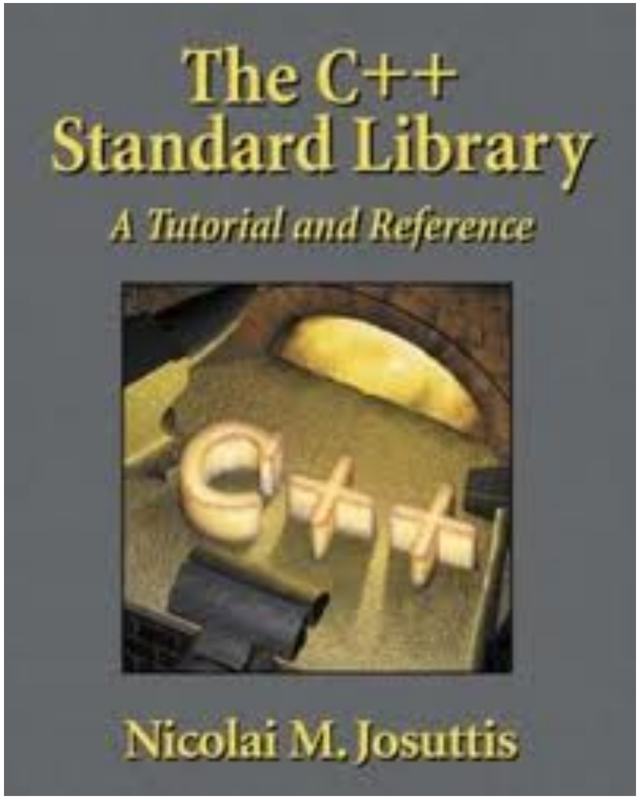
Fortunately, this same problem doesn't occur with sets, lists or maps.

With one exception...

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

# 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>

int main()
{
  vector<string>  n;

  n.push_back("carey");
  n.push_back("bart");
  n.push_back("alex");

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

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

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

| carey | bart | alex |
|---|---|---|

→

| alex | bart | carey |
|---|---|---|

| 2 | 5 | 1 | -7 |
|---|---|---|---|

→

| -7 | 1 | 2 | 5 |
|---|---|---|---|

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!

```cpp
#include <algorithm>

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

// 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();
}

int main()
{
    Dog arr[4] = {...};

   sort ( arr, arr+4, &customCompare);
}
```

# The "sort" function

- You can use the sort function to sort a bunch of items in many different arbitrary.
- For example, lets say we want to sort Dogs based on how nasty their bite is first, and how loud their bark is, second…
- Here's how do do that:
- First, you define a new function that can compare two Dogs, A and B. The function must:

return true if A belongs before B
return false if A belongs after B.

- For instance, our customCompare function will place dogs with a bigger bite before dogs with a smaller bite and break ties by the loudest bark…
- We then pass in a pointer to our function to the sort() function and it will use the passed-in comparison function to figure out how to order the items!

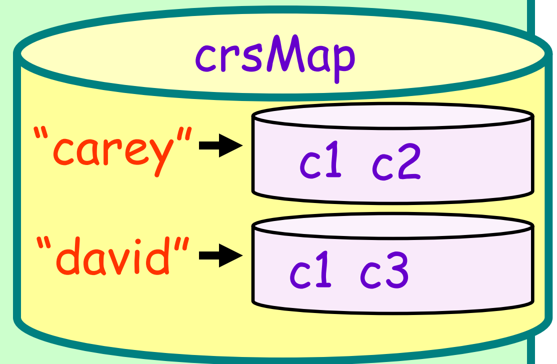
# Part 6: Compound STL Data Structures

Let's say you want to maintain a list of courses for each UCLA student.

How could you do it with the STL?

Well, how about creating a map between a student's name and their list of courses?

In many cases, you'll want to combine multiple STL containers to represent more complex associations like this!




```
#include <map>
#include <list>

class Course
{
public:
     ...
};

int main()
{
   map<string, list<Course>>  crsmap;

   Course c1("cs","32"),
          c2("math","3b"),
          c3("english","1");

   crsmap["carey"].push_back(c1);
   crsmap["carey"].push_back(c2);
   crsmap["david"].push_back(c1);
   crsmap["david"].push_back(c3);
```

# STL Challenges

Design a compound 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 compound STL data structure to associate people with the group of courses (e.g., Course objects) they've taken, and further associate each course with the grade (e.g. a string like "A+") they got for that course.

```
// you could do this…

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

- Don't forget: If you're mapping your own class or struct to something else you'll need to define the operator< for it
- Also, if you have a set containing your own class you'll need to define the operator< for it

# Appendix - On Your Own Study

- Inline Functions
- Template Exercise
- More STL Algorithm Functions
  - find()
  - find_if()

# Inline Methods

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

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

Since my entire body is defined inside the class declaration, I'm inline by default in C++.

Since my code is defined outside the class declaration, I'm not an inline method unless the programmer explicitly says so.

```
int main()
{
 Foo<int> nerd;
 nerd.setVal(5);
 nerd.printVal();
 nerd.setVal(10);
}
```

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

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.
When the compiler compiles your inline function, it basically copies the body of your function directly to where the function was being called, eliminating the function call.

- By replacing the function call to printVal with its actual code, this reduces the amount of jumping around your program must do, speeding it up!
This is transparent during compilation, so you don't see it happening.
Technically, C++ is not required to honor inline keyword – this is just a request by the programmer to the compiler.
- Be careful, while inline functions can speed up your program, they also can make your EXE file bigger!

# Template Exercise

```
class Stack
{
public:
    Stack()
    {  m_top = 0; }

    void push( int v )
    {
      m_items[m_top++] = v;
    }

    int pop();

private:
    int m_items[100];
    int m_top;
};


int Stack::pop()
{
   return m_items[--m_top];
}
```

## Part #1

Convert this Stack class to one that can hold any type of data.

## Part #2

Show how you would create a stack of Dogs and push Fido on.

```
int main()
{


}
```

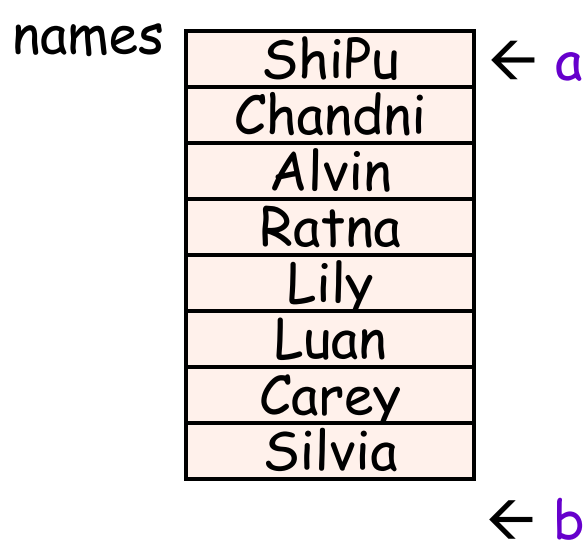
# The STL "find" Function

```
#include <list>
#include <algorithm>

int main()
{
  list<string>  names;
  ... // fill with a bunch of 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;
}
```

- The STL provides a find function that works with vectors/lists.
- (They don't have built-in find methods like map & set)
- Make sure to include the algorithm header file!
- The first argument is an iterator that points to where you want to start searching.
- The second argument is an iterator that points JUST AFTER where you want to stop searching!
- The final argument is what you're searching for.
- And just like set and map's find methods, this version returns an iterator to the item that it found.
- And if find couldn't locate the item, it will return whatever you passed in for the second parameter.
- So make sure to check for this value to see if the find function was successful!

# The STL “find” Function

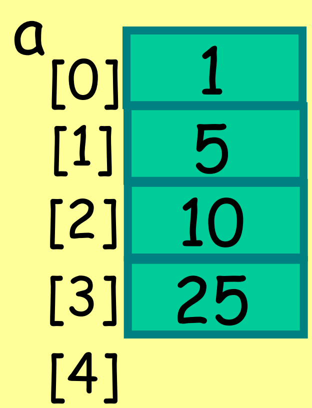
```
#include <list>
#include <algorithm>

int main()
{
  list<string>  names;
  ... // fill with a bunch of names


  list<string>::iterator a, b, iter;

  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;
}
```




- In this example, since Judy is not in our list, the find() function will return the value of b, or in this case, names.end()
- Note: The second parameter need not always point at the end of the list/vector.
- If you just wanted to search the first 500 items of a list of 10k items, you could just pass in an iterator to the 501st item as the second parameter.

# The STL “find” Function

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
  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;
}
```




This find function also works with arrays!

For the first argument, pass the address where you want to start searching in the array.

For the second argument, pass the address of the element AFTER the last item your want to search.

find will return a pointer to the found item, or to the second parameter if the item can't be found.

# The find_if Function

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

bool is_even(int n) // predicate func
{
  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;
}
```

ptr

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

Your predicate function must take in a single value as a parameter (the type must match the type of values in your container), and return a true if the value matches the predicate, false otherwise.

find_if processes each item in the container until the predicate function returns true or it runs out of items.

find_if returns an iterator/pointer to the first item that triggers the predicate function.

# The find_if Function

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

bool is_even(int n)
{
  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;
}
```

Your predicate function must return a boolean value.

The predicate function must accept values that are of the same type as the ones in the container/array.

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

(your predicate function's logic determines the requirements)