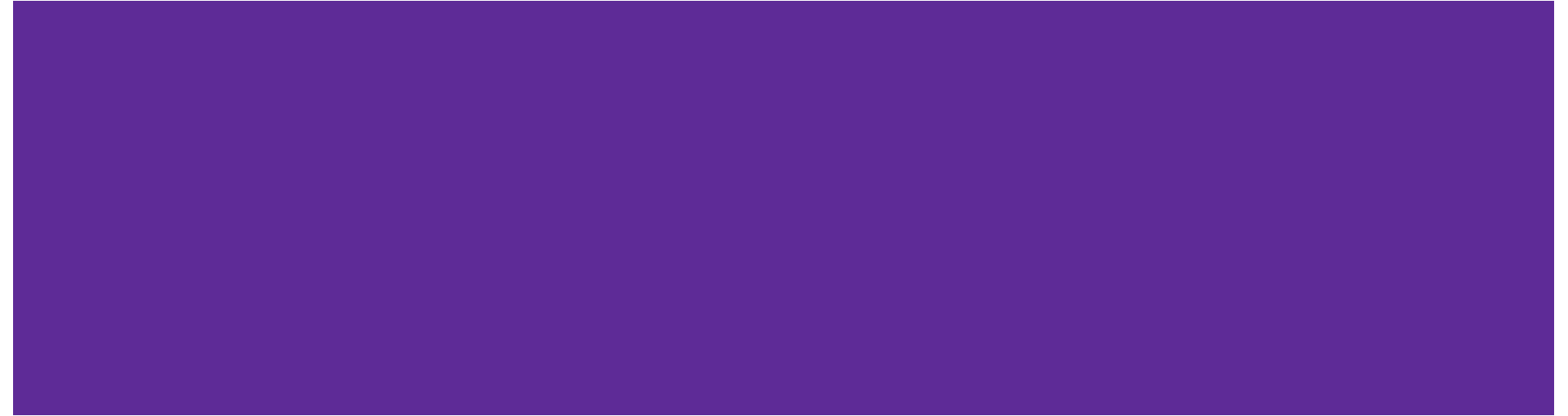


CSE 374 Programming Concepts and Tools

Lecture 20 - C++ Templates, STL



This week

Monday: C++ Class Details

Wednesday: C++ Templates, STL

Friday: C++ Smart Pointers

Today

Today: Templates, STL

- Review
- Template Types
- Template Non-Types
- STL
- Containers
- Iterators
- Algorithms
- C++ STL vs. Java Standard Library

Review: new/delete

To allocate on the heap using C++, you use the `new` keyword

- You can use `new` to allocate an object (e.g. `new Point`)
- You can use `new` to allocate a primitive type (e.g. `new int`)
- When allocating you can specify a constructor or initial value
 - (e.g. `new Point(1, 2)`) or (e.g. `new int(333)`)
- If no initialization specified, it will use default constructor for objects, garbage for primitives (integer, float, character, boolean, double)
 - `new` returns a *pointer* (of the appropriate type)
 - You don't need to check that `new` returns `nullptr`

To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of `free()`

- **Don't mix and match!**

Review: Dynamically Allocated Arrays

To dynamically allocate an array:

- Default initialize: `type* name = new type[size];`

To dynamically deallocate an array:

- Use `delete[] name;`
- It is an *incorrect* to use “`delete name;`” on an array
 - The compiler probably won't catch this, though (!) because it can't always tell if `name*` was allocated with `new type[size];` or `new type;`
 - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
 - Result of wrong `delete` is undefined behavior

Templates

Suppose that...

- You want to write a function to compare two `ints`
- You want to write a function to compare two `strings`
 - Function overloading!

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int& value1, const int& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string& value1, const string& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```

Hm...

The two implementations of `compare` are nearly identical!

- What if we wanted a version of `compare` for every comparable type?
- We could write (many) more functions, but that's obviously wasteful and redundant
 - Too much repeated code!

What we'd prefer to do is write “*generic code*”

- Code that is `type-independent`
- Code that is `compile-time polymorphic` across types

C++ Parametric Polymorphism

C++ has the notion of **templates** (similar to Java's *generics*)

- A function or class that can accept **type(s)** as parameter(s)
 - You define the function or class once in a type-agnostic way
 - When you invoke the function or instantiate the class, you specify the type(s) to fill in the “template”
- At compile-time, the compiler will generate the “specialized” code from your template using the types you provided
 - The template definition is NOT runnable code
 - Runnable code is *only* generated if you use your template
- Implemented using *macros*
 - So you can also use values in templates as well, which we'll see in an example soon...

Function Templates

Template to **compare** two “things” of the same type:

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>    // <...> can also be written <class T>
int compare(const T& value1, const T& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return EXIT_SUCCESS;
}
```

Only uses operator < to minimize requirements on T

Explicit type argument

Function Templates with Inference...

Writing all these type parameters can get tedious...

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>    // <...> can also be written <class T>
int compare(const T& value1, const T& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return EXIT_SUCCESS;
}
```

Function Templates with Inference...

Writing all these type parameters can get tedious...so we can remove them!

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>    // <...> can also be written <class T>
int compare(const T& value1, const T& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl;
    std::cout << compare(h, w) << std::endl;
    std::cout << compare(50.5, 50.6) << std::endl;
    return EXIT_SUCCESS;
}
```

Compiler Inference

The compiler can infer the types for us, but there are some things to keep in

```
m #include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(const T& value1, const T& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl; // ok, infers int
    std::cout << compare(h, w) << std::endl;    // ok, infers string
    std::cout << compare("Hello", "World") << std::endl; // hm...
    return EXIT_SUCCESS;
}
```

← Infers char* – does address integer
comparison

Template Non-types

You can use non-types (constant values) in a template:

```
#include <iostream>
#include <string>

// return pointer to new N-element heap array filled with val
// (not entirely realistic, but shows what's possible)
template <typename T, int N>
T* valarray(const T& val) {
    T* a = new T[N];
    for (int i = 0; i < N; ++i)
        a[i] = val;
    return a;
}

int main(int argc, char **argv) {
    int* ip = valarray<int, 10>(17);
    string* sp = valarray<string, 17>("hello");
    ...
}
```

Fixed type template
parameter

Use comma separated list to
specify template arguments

What's Going On?

The compiler doesn't generate any code when it sees the template function

- It doesn't know what code to generate yet, since it doesn't know what types are involved

When the compiler sees the function being used, then it understands what types are involved

- It generates the **instantiation** of the template and compiles it (kind of like macro expansion)
 - The compiler generates template instantiations for *each* type used as a template parameter

Polling Time!

Please answer in the LP20 assignment on Gradescope!

Suppose we have a template function called **Add3**.

What value is returned for each of the following lines of code?

- **Add3**<int>(3); $\rightarrow 6$
- **Add3**(5.5); $\rightarrow 8.5$
- **Add3**<char*>("a str");

```
template <typename T>
T Add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Polling Time!

Please answer in the LP20 assignment on Gradescope!

Suppose we have a template function called **Add3**.

What value is returned for each of the following lines of code?

- **Add3**<int>(3); -> 6
- **Add3**(5.5); -> 8.5
- **Add3**<char*>("a str"); -> "tr"

```
template <typename T>
T Add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Class Templates

Templates are very useful for classes

- This was one of the main motivations for adding templates

Imagine we want a class that holds a pair of things that we can:

- Set the value of the first thing
- Set the value of the second thing
- Get the value of the first thing
- Get the value of the second thing
- Swap the values of the things
- Print the pair of things

Pair Class Definition

Pair.h

```
#ifndef PAIR_H_
#define PAIR_H_
```

Template parameters for class definition



```
template <typename Thing> class Pair {
public:
    Pair() { };

    Thing get_first() const { return first_; }
    Thing get_second() const { return second_; }
    void set_first(Thing& copyme);
    void set_second(Thing& copyme);
    void Swap();
```

Could be objects, could be primitives



```
private:
    Thing first_, second_;
};
```

```
#include "Pair.cc"
#endif // PAIR_H_
```

Pair Function Definitions

Pair.cc

```
template <typename Thing>
void Pair<Thing>::set_first(Thing& copyme) {
    first_ = copyme;
}
template <typename Thing>
void Pair<Thing>::set_second(Thing& copyme) {
    second_ = copyme;
}
template <typename Thing>
void Pair<Thing>::Swap() {
    Thing tmp = first_;
    first_ = second_;
    second_ = tmp;
}
template <typename T>
std::ostream &operator<<(std::ostream& out, const Pair<T>& p) {
    return out << "Pair(" << p.get_first() << ", "
               << p.get_second() << ")";
}
```

Definition of member
function of template class

member of template
class

Non member function to print
out data in template class

Using Pair

usepair.cc

```
#include <iostream>
#include <string>

#include "Pair.h"

int main(int argc, char** argv) {
    Pair<std::string> ps;
    std::string x("foo"), y("bar");

    ps.set_first(x);           // ("foo", "")
    ps.set_second(y);          // ("foo", "bar")
    ps.Swap();                 // ("bar", "foo")
    std::cout << ps << std::endl;

    return EXIT_SUCCESS;
}
```

Invokes default ctor, which
default constructs members ("",
"")



Demo: Pair Template



STL

C++'s Standard Library

C++'s Standard Library consists of four major pieces:

1. The entire C standard library: `cstdio`, `cstdlib`, etc.
2. C++'s input/output stream library
 - `std::cin`, `std::cout`, `stringstream`, `fstream`, etc.
3. C++'s standard template library (STL) 🙌
 - Containers, iterators, algorithms (sort, find, etc.), numerics
4. C++'s miscellaneous library
 - Strings, exceptions, memory allocation, localization

STL Containers

A **container** is an object that stores (in memory) a collection of other objects (elements)

- Implemented as class *templates*, so hugely flexible

Several different classes of container

- Sequence containers (vector, deque, list, ...)
- Associative containers (set, map, multiset, multimap, bitset, ...)
- Differ in algorithmic cost and supported operations

STL Containers

STL containers store elements by **value**, not by reference

- When you insert an object, the container makes a **copy**
- If the container needs to rearrange objects, it makes copies
 - e.g. if you sort a `vector`, it will make many, many copies
 - e.g. if you insert into a `map`, that may trigger several copies
- What if you don't want this? (disabled copy constructor or copying is expensive)
 - You can insert a wrapper object with a pointer to the object
 - Called a “**smart pointer**” – we'll learn about these during the next lecture

Our Tracer Class

Wrapper class for an `int` `value_`.

Two fields:
`value`
`id` (unique to the
instance)

- Also holds unique `int` `id_` (increasing from 0)
- Default ctor (set unique `id_` for each instance), ctor, dtor, `op=`, `op<` defined
- `friend` function `operator<<` defined
- Private helper method `PrintID()` to return "`(id_, value_)`" as a string
- Class and member definitions can be found in `Tracer.h` and `Tracer.cc`

Useful for tracing behaviors of containers

- All methods print identifying messages
- Unique `id_` allows you to follow individual instances

Demo: Tracer Walkthrough



STL **vector**

A generic, dynamically resizable array

- <https://cplusplus.com/reference/vector/vector/>
- Elements are stored in **contiguous** memory locations
 - Like a normal C array, or the ArrayList in Java!
 - Elements can be accessed using pointer arithmetic if you'd like
 - Random access is $O(1)$ time
 - Pointer arithmetic, then access
- Adding/removing from the end is cheap (amortized constant time)
- Inserting/deleting from the middle or start is expensive (linear time)
 - Need to shift all of the elements in the array

vector/Tracer Example

vectorfun.cc

```
#include <iostream>
#include <vector>
#include "Tracer.h"
```

Most containers are declared in library of same name

```
using namespace std;
```

```
int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;
```

Construct three tracer instances and empty vector

```
    cout << "vec.push_back " << a << endl;
    vec.push_back(a);
    cout << "vec.push_back " << b << endl;
    vec.push_back(b);
    cout << "vec.push_back " << c << endl;
    vec.push_back(c);
```

Add tracers to end of vector

```
    cout << "vec[0]" << endl << vec[0] << endl;
    cout << "vec[2]" << endl << vec[2] << endl;
    return EXIT_SUCCESS;
```

Array syntax to access elements

```
}
```

Dynamic Resizing

What's going on here?

- Answer: a C++ vector (like Java's ArrayList) is initially small, but grows if needed as elements are added
 - Implemented by allocating a new, larger underlying array, copy existing elements to new array, and then replace previous array with new one
- And vector starts out *really* small by default, so it needs to grow almost immediately!
 - But you can specify an initial capacity if “really small” is an inefficient initial size (use `reserve()` member function)

Demo: Vectors



STL iterator

Each container class has an associated `iterator` class (e.g. `vector<int>::iterator`) used to iterate through elements of the container

- <https://cplusplus.com/reference/iterator/>
- `Iterator range` is from `begin` up to `end` i.e., [`begin`, `end`)
 - `end` is one past the last container element!
- Some container iterators support more operations than others
 - All can be incremented (`++`), copied, copy-constructed
 - Some can be dereferenced on RHS (e.g. `x = *it;`)
 - Some can be dereferenced on LHS (e.g. `*it = x;`)
 - Some can be decremented (`--`)
 - Some support random access (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

iterator Example

vectoriterator.cc

```
#include <vector>
#include "Tracer.h"
using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    vector<Tracer>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

(first element, one past the end, increment to next element)

Dereference to access element

Type Inference (C++11)

The `auto` keyword can be used to infer types

- Simplifies your life if, for example, functions return complicated types
- The expression using `auto` must contain explicit initialization for it to work

Compiler knows return value of
`Factors()`



???? No information to infer
type



```
// Calculate and return a vector  
// containing all factors of n  
std::vector<int> Factors(int n);  
  
void foo(void) {  
    // Manually identified type  
    std::vector<int> facts1 =  
        Factors(324234);  
  
    // Inferred type  
    auto facts2 = Factors(12321);  
  
    // Compiler error here  
    auto facts3;  
}
```

auto and Iterators

Life becomes much simpler!

```
for (vector<Tracer>::iterator it = vec.begin(); it < vec.end(); it++) {  
    cout << *it << endl;  
}
```



```
for (auto it = vec.begin(); it < vec.end(); it++) {  
    cout << *it << endl;  
}
```



Range for Statement (C++11)

Syntactic sugar basically the same as Java's `for-each` loops

```
for ( declaration : expression ) {  
    statements  
}
```

- `declaration` defines loop variable
- `expression` is an object representing a sequence
 - Strings, initializer lists, arrays with an explicit length defined, STL containers that support

`str` = sequence of
characters

```
// Prints out a string, one  
// character per line  
std::string str("hello");  
  
for (auto c : str) {  
    std::cout << c << std::endl;  
}
```

Updated iterator Example

vectoriterator_2011.cc

```
#include <vector>
#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    for (auto& p : vec) { // p is a reference (alias) of vec
        cout << p << endl; // element here; not a new copy
    }

    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

Look at how much more simplified this is!
No `begin()`, `end()`, or dereferencing! :O

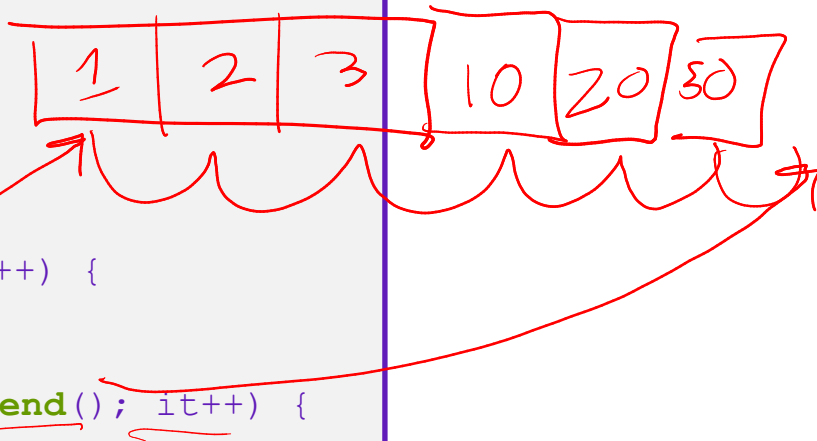
Polling Time!

Please answer in the LP20 assignment on Gradescope!

What is the output of this program?

poll.cc

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> nums = {1, 2, 3};
    size_t original_size = nums.size();
    for (size_t i = 1; i <= original_size; i++) {
        nums.push_back(i * 10);
    }
    for (auto it = nums.begin(); it != nums.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
    return EXIT_SUCCESS;
}
```



Polling Time!

Please answer in the [LP20 assignment on Gradescope!](#)

What is the output of this program?: **1 2 3 10 20 30**

poll.cc

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> nums = {1, 2, 3};
    size_t original_size = nums.size();
    for (size_t i = 1; i <= original_size; i++) {
        nums.push_back(i * 10);
    }
    for (auto it = nums.begin(); it != nums.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
    return EXIT_SUCCESS;
}
```

STL Algorithms

A set of functions to be used on ranges of elements

- **Range**: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
- General form `algorithm(begin, end, ...);`

Algorithms operate directly on range *elements* rather than the containers they live in

- Make use of elements' copy ctor, =, ==, !=, <
- Some do not modify elements
 - e.g. find, count, for_each, min_element, binary_search
- Some do modify elements
 - e.g. sort, transform, copy, swap

Algorithms Example

vectoralgos.cc

```
#include <vector>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer& p) {
    cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(c);
    vec.push_back(a);
    vec.push_back(b);
    cout << "sort:" << endl;
    sort(vec.begin(), vec.end());
    cout << "done sort!" << endl;
    for_each(vec.begin(), vec.end(), PrintOut);
    return EXIT_SUCCESS;
}
```

Sort elements from
[vec.begin(), vec.end())

Runs function on each
element. In this case,
prints out each element.

STL `list`

A generic doubly-linked list

- <https://cplusplus.com/reference/list/list/>
- Elements are **not** stored in contiguous memory locations
 - Does not support random access (e.g. cannot do `list[5]`)
- Some operations are much more efficient than vectors
 - Constant time insertion, deletion anywhere in list
 - Can iterate forward or backwards
 - Backward: `--`
 - Forward: `++`
- Has a built-in sort member function
 - Doesn't copy! Manipulates list structure instead of element values

list Example

listexample.cc

```
#include <list>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer& p) {
    cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
    Tracer a, b, c;
    list<Tracer> lst;

    lst.push_back(c);
    lst.push_back(a);
    lst.push_back(b);
    cout << "sort:" << endl;
    lst.sort();
    cout << "done sort!" << endl;
    for_each(lst.begin(), lst.end(), PrintOut);
    return EXIT_SUCCESS;
}
```

Use case is similar to vector, but
internal implementation is different.

Won't copy elements, just modifies the
next and prev pointers.

STL `map`

One of C++'s *associative* containers: a key/value table, implemented as a search tree

- <https://cplusplus.com/library/numeric/stringmap/> `map<key_type, value_type> name;`
- General form:
- Keys must be *unique*
 - `multimap` allows duplicate keys
- Efficient lookup ($O(\log n)$) and insertion ($O(\log n)$)
 - Access value via `name[key]`
 - If key doesn't exist in map, it is added to the map
- Elements are type `pair<key_type, value_type>` and are stored in *sorted* order (key is field `first`, value is field `second`)
 - Key type must support less-than operator (`<`)

map Example

mapexample.cc

```
void PrintOut(const pair<Tracer,Tracer>& p) {  
    cout << "printout: [" << p.first << "," << p.second << "]" << endl;  
}
```

```
int main(int argc, char** argv) {  
    Tracer a, b, c, d, e, f;  
    map<Tracer,Tracer> table;  
    map<Tracer,Tracer>::iterator it;
```

```
    table.insert(pair<Tracer,Tracer>(a, b));
```

```
    table[c] = d;
```

```
    table[e] = f;
```

```
    cout << "table[e]:" << table[e] << endl;
```

```
    it = table.find(c);
```

```
    cout << "PrintOut(*it), where it = table.find(c)" << endl;
```

```
    PrintOut(*it);
```

```
    cout << "iterating:" << endl;
```

```
    for_each(table.begin(), table.end(), PrintOut);
```

```
    return EXIT_SUCCESS;
```

```
}
```

Equivalent
behavior

Returns iterator (end if not
found). Can also use
map.count() to see if a key
exists.

Unordered Containers (C++11)

`unordered_map, unordered_set`

- Average case for key access is $O(1)$
 - But range iterators can be less efficient than ordered `map/set`
 - Elements are not stored in contiguous order (stored based on the hash).
- See *C++ Primer*, online references for details

Demo: Animals



C++ standard lib is built around templates

Containers store data using various underlying data structures

- The specifics of the data structures define properties and operations for the container

Iterators allow you to traverse container data

- Iterators form the common interface to containers
- Different flavors based on underlying data structure

Algorithms perform common, useful operations on containers

- Use the common interface of iterators, but different algorithms require different 'complexities' of iterators

Common C++ STL Containers (and Java equiv)

Sequence containers can be accessed sequentially

- `vector<Item>` uses a dynamically-sized contiguous array (like `ArrayList`)
- `list<Item>` uses a doubly-linked list (like `LinkedList`)

Associative containers use search trees and are sorted by keys

- `set<Key>` only stores keys (like `TreeSet`)
- `map<Key, Value>` stores key-value pair<>'s (like `TreeMap`)

Unordered associative containers are hashed

- `unordered_map<Key, Value>` (like `HashMap`)

Assignment Reminders