Today

- Quiz 2
- Templates in C++
- CMake and GTest

Assignments

- HW2 due
- HW3 out
C++ Template types

Many ADTs and data structures work over any type of value

- stack of ints, strings, doubles, ...
- linked list of ints, strings, doubles, ...

One approach is to use a `typedef` statement

```cpp
typedef int ListType;

... then ...

ListType my_array[10];

... or ...

struct Node {
    ListType val;
    Node* next;
};
```

- and then change the type in the `typedef` as needed

Q: What is the problem with this?

- changing type means changing the code (thus, recompiling the library)
- can’t easily have a list of `int` and a list of `string`
Class “templates” allow you to declare types as parameters

A simple templated class (with a type parameter)

```cpp
template <typename T>
class Box
{
public:
    Box(T val); // box up a value
    T unbox() const; // get box value
    void set(T new_val); // replace box value with new value
private:
    T boxed_value;
};
```

We pass in types (as parameters) to create boxed values:

```cpp
int main()
{
    Box<int> b1(42); // create a box of int
    cout << b1.unbox() << endl; // prints 42

    Box<string> b2("foo"); // create a box of string
    cout << b2.unbox() << endl; // prints foo
}
```
To implement the class methods (have to include type params)

```cpp
template <typename T>
Box<T>::Box(T val) : boxed_value(val)
{
}

template <typename T>
T Box<T>::unbox() const
{
    return boxed_value;
}

... etc ...

• Every function has to announce the template type

• Annoying but required

Also, because of issues with compilation ...

• need to include the function implementations in the class header file

• also annoying but required
Example with 2 Parameterized Types

```cpp
template <typename K, typename V>
class KVPair
{
  public:
    KVPair(K the_key, V the_val);
    K get_key() const;
    V get_val() const;
  private:
    K key;
    V val;
};

template <typename K, typename V> KVPair<K,V>::KVPair(K the_key, V the_val)
  : key(the_key), val(the_val)
{
}

template <typename K, typename V> K KVPair<K,V>::get_key() const
{
  return key;
}

template <typename K, typename V> K KVPair<K,V>::get_val() const
{
  return val;
}

... examples ....

int main()
{
  KVPair<string,int> p1("foo", 42);
  KVPair<int,int> p2(2,8);
  KVPair<KVPair<string,int>,double> p3(p1, 3.14);
}
```
The **Collection** class is both abstract and has parameterized types!

```cpp
#ifdef COLLECTION_H
#define COLLECTION_H

#include <vector>

template<typename K, typename V>
class Collection
{
public:

    // add a new key-value pair into the collection
    virtual void add(const K& a_key, const V& a_val) = 0;

    // remove a key-value pair from the collectiona
    virtual void remove(const K& a_key) = 0;

    // find and return the value associated with the key
    virtual bool find(const K& search_key, V& the_val) const = 0;

    // find and return the values with keys >= k1 and <= k2
    virtual void find(const K& k1, const K& k2, std::vector<V>& vals) const = 0;

    // return all of the keys in the collection
    virtual void keys(std::vector<K>& all_keys) const = 0;

    // return all of the keys in ascending (sorted) order
    virtual void sort(std::vector<K>& all_keys_sorted) const = 0;

    // return the number of key-value pairs in the collection
    virtual int size() const = 0;

};
#endif
```
In HW3, we create a vector-based implementation of Collection

- recall that a vector is a resizable array ...

The vector_collection.h file

```cpp
#ifndef VECTOR_COLLECTION_H
#define VECTOR_COLLECTION_H

#include <vector> // for vector
#include <algorithm> // for sorting
#include "collection.h" // the base class

template<typename K, typename V>
class VectorCollection : public Collection<K,V>
{
public:
  void insert(const K& a_key, const V& a_val);

  void remove(const K& a_key);

  bool find(const K& search_key, V& the_val) const;

  void find(const K& k1, const K& k2, std::vector<V>& vals) const;

  void keys(std::vector<K>& all_keys) const;

  void sort(std::vector<K>& all_keys_sorted) const;

  int size() const;

private:
  std::vector<std::pair<K,V>> kv_list;
};

#endif
```
Note that C++ provides a pair type similar to KVPair ...

We'll use this in HW3 for our "vector" based implementation

```cpp
template <typename K, typename V>
class VectorCollection : public Collection<K, V>
{
    public:
        ...

    private:
        std::vector<std::pair<K, V>> kv_list;
};
```

To create a pair object ...

```cpp
// assuming K, V, key, and val are known/defined
std::pair<K, V> p(key, val)
```

To access the elements of a pair

```cpp
key = p.first;
val = p.second;
```
**C++ for each style of looping**

In C++ often convenient to use “for each” style loops:

```cpp
// assuming kv_list defined as: vector<pair<K,V>> kv_list

for (std::pair<K,V> p : kv_list) {
    if (p.first == some_k_value) {
        // do something with p
    }
}
```

Can also loop through a vector using an iterator

```cpp
// assume xs defined as: vector<int> xs

// traditional approach
for (int i = 0; i < int(xs.size()); ++i)
    cout << xs[i] << endl;

// using an iterator
for (vector<int>::iterator it = xs.begin(); it != xs.end(); ++it)
    cout << *it << endl;
```
Can use the `std::sort` function to sort values (in `algorithms`)

```cpp
// assume xs is defined as: vector<int> xs
vector<int> sorted_xs;
for (int x : xs)
    sorted_xs.push_back(x);

// begin() and end() return iterator pointers
// to first and last element
std::sort(sorted_xs.begin(), sorted_xs.end());

// check if sorted (ascending order)
for (int i = 0; i < int(xs.size()) - 1; ++i)
    assert(xs[i] <= xs[i+1]);
```
Building and Compiling HW3 (and beyond)

We are using some new tools ...

- Google Test for basic unit tests (hw3_test.cpp)
- CMake to help set up compilation (to create a makefile)
- `make` to compile everything

Works out of the box on ada and the department VM

- but requires installation otherwise (make, CMake, and Google Test)

You will also be creating two executable test programs

- One for running your unit tests (hw3test)
- One for running some performance tests (hw3perf)
- I’m providing you with some test data to use for basic performance testing
- The goal:
  - make sure the data structure implementations are correct
  - see how the different data structure implementations compare

Short Demo ...