lecture07: Templates

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

19th June, 2013
Announcements

- Lab memory due tomorrow evening (6/20)
- mp2.1 due Friday evening (6/21)
- mp2 due Monday evening (6/24)

Source: https://developers.google.com/apps-script/images/mail_merge_image5.png
Announcements

- lab_memory due tomorrow evening (6/20)
- mp2.1 due Friday evening (6/21)
- mp2 due Monday evening (6/24)

Announcements

- lab_memory due tomorrow evening (6/20)
- mp2.1 due Friday evening (6/21)
- mp2 due Monday evening (6/24)

Source: https://s3.amazonaws.com/luuux-original-files/bookmarklet_uploaded/stamp3.jpg
Finishing up a concept from yesterday

- What if a function is declared `virtual`, but never implemented?
- The class becomes an *abstract base class*, and cannot be instantiated (think “interface”)
- Let’s turn our `Animal` base class into an abstract base class by making `speak` be *pure virtual*
- Instead of making the default `Animal` say “???” , we’ll make it so you can’t create one
An abstract base class

/** @file animal.h */

#include <string>
using std::string;

class Animal
{
  public:
    Animal(const string & name);
    void eat();
    void sleep() const;
    // note the = 0 below:
    virtual void speak() const = 0;
};

// an abstract base class has at
// least one pure virtual function

/** @file main.cpp */

// compiler error!
Animal ani("Fred");
Lion lion("George");
lion.sleep();
lion.speak(); // "roar!"

Animal* pen = new Penguin("Awkward");
pen->eat();
pen->speak(); // "ork ork!"
Derived classes must implement all pure virtual functions; otherwise the deriving class will also be abstract and can’t be instantiated.

An abstract base class contains a list of functions that must be implemented; this way, we can be sure that derived classes have some specific functionality.
void swap_int(int & a, int & b)
{
    int temp = a;
    a = b;
    b = temp;
}

void swap_sphere(Sphere & a, Sphere & b)
{
    Sphere temp = a;
    a = b;
    b = temp;
}

void swap_double(double & a, double & b)
{
    double temp = a;
    a = b;
    b = temp;
}

// do we really need this many
// swap functions??

int main()
{
    int x = 42;
    int y = 47;
    Sphere s1(4.0);
    Sphere s2(6.0);
    double a = 0.01;
    double b = 0.9;

    swap_int(x, y);
    swap_sphere(s1, s2);
    swap_double(a, b);

    return 0;
}
Clearly, all this duplication is bad; the only difference in the functions was the type of the variables.

Can we somehow parameterize the type as well as the variables?

Templates to the rescue!

(Could we do this with dynamic polymorphism? How?)
Templates to the rescue

```cpp
template <class T>
void swap(T & a, T & b)
{
    T temp = a;
    a = b;
    b = temp;
}

// this is certainly shorter,
// and it works for types we
// haven’t even seen yet!

int main()
{
    int x = 42;
    int y = 47;
    Sphere s1(4.0);
    Sphere s2(6.0);
    double a = 0.01;
    double b = 0.9;

    swap(x, y);
    swap(s1, s2);
    swap(a, b);

    return 0;
}
```
How does this actually work?

- At compile time, the compiler finds all calls to the templated function, and replaces the templated types (in our case, T), with the actual types.
- That means that a separate copy of the function is created for each type it is called with.
- Since all this happens at compile time, this is an example of static polymorphism.
- Can you write a generic \texttt{max} function?
Max function

```cpp
template <class T>
T max(const T & a, const T & b) 
{
    if(a > b)
        return a;
    return b;
}
```

// what is required of type T here?
Write your own "pow" function

We want the following functionality:

```cpp
double d = 2.0;
int i = 3;

pow(d, 3);    // 8.0
pow(d, 4);    // 16.0
pow(i, 0);    // 1
pow(i, 10);   // 59049
```

Can you write the templated pow function? To simplify the function, assume that the exponent is a non-negative integer.
Pow function

```cpp
template <class T>
T pow(T base, int exp)
{
    if(exp == 0)
        return 1;  // converted to T in return

    T ans = base;
    for(int i = 1; i < exp; ++i)
        ans *= base;

    return ans;
}
```
Classes can be templated, too!

A templated class can also have one or more placeholder types associated with it:

```cpp
/** @file pair.h */

template <class T>
class Pair
{
    public:
        Pair(const T & first, const T & second);
        T max() const;
    private:
        T _first;
        T _second;
};
```
Using a templated class

```cpp
/** @file pair.h */

#ifndef PAIR_H
#define PAIR_H

template <class T>
template class Pair
{
    public:
        Pair(const T & first,
             const T & second);
        T max() const;
    private:
        T _first;
        T _second;
};
#endif

/** @file main.cpp */

#include <iostream>
#include "pair.h"
#include "sphere.h"

using namespace std;

void main()
{
    Pair<int> mypair(4, 5);
    cout << mypair.max() << endl;

    Sphere a(6.7);
    Sphere b(3.2);
    Pair<Sphere> myspheres(a, b);
    cout << myspheres.max() << endl;
}
```
Implementing a templated class

```cpp
/** @file pair.h */

#ifndef PAIR_H
#define PAIR_H

template <class T>
class Pair
{
  public:
    Pair(const T & first, const T & second);
    T max() const;
  private:
    T _first;
    T _second;
};

#include "pair.cpp"
#endif

/** @file pair.cpp */

// no #include here

template <class T>
Pair<T>::Pair(const T & first, const T & second)
{
  _first = first;
  _second = second;
}

template <class T>
T Pair<T>::max() const
{
  if(_first > _second)
    return _first;
  return _second;
}
```

Sean Massung

Templates
What do you notice?

- We include pair.cpp inside pair.h!
  - The definition of a template function or class *must* be available to the calling code
  - Otherwise, it can't be instantiated

- When scoping to the class in the .cpp file, we include the template parameter: `Pair<T>::__`

- Also inside the .cpp file, we need a template `<class T>` before each function
Pop quiz!

- Write the function signature for the copy constructor of the templated Pair class

- Dynamically allocate an array of size $size$ of Pairs of Spheres

- Write the class definition of a template Pair class that takes two different types
std::swap and std::pair already exist
Don't rewrite them!
#include <algorithm> for std::swap
#include <utility> for std::pair
pow also already exists...
What are some other potentially useful template classes or functions?