CSCI 262
Data Structures

18 – Templates

CS@Mines

Generic Programming

- Multiple types can take on identical roles *in certain contexts*
- Define *generic* behaviors/containers to work with *unspecified* types
- One way we can achieve code reuse
- In C++, generic programming is achieved via *templates*

CS@Mines

FUNCTION TEMPLATES

CS@Mines

Example: swap()

- A common function:
  - Used heavily, for instance, in sort algorithms
  - You've probably written code like this before ☺
- A simple implementation for int values:

```
void swap(int &a, int &b) {
        int tmp = a;
        a = b;
        b = tmp;
}
```

CS@Mines

Overloading swap()

We could *overload* swap to also act on doubles, chars, or just about anything:

```
void swap(double &a, double &b) {
        double tmp = a;
        a = b;
        b = tmp;
}
void swap(char &a, char &b) {
        char tmp = a;
        a = b;
        b = tmp;
}
```

Note how only the types change – otherwise identical!

CS@Mines

Templatizing swap()

Declares the template and the template parameter, T. An alternate syntax is template <typename T>.

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

Defines the template function. Note how it is identical to a regular function definition, but now we have access to an *unspecified* type, T.

CS@Mines

Using swap()

Explicit parameterization:

```
double x = 4.0, y = 10.5;
swap<double>(x, y);
```

Implicit parameterization:

```
double x = 4.0, y = 10.5;
swap(x, y);
```


Using swap(), con't.

Note this won't work:

```
double x = 4.0;
int y = 10;
swap(x, y);
```

Why? It doesn't match the pattern!

Unfortunately, this kind of mistake results in **huge** compiler error messages, which can be daunting. They usually start like this, though:

```
error: no matching function for call to 'swap(int&, double&)'
 swap(x, y);
```


Where to Define swap()

- Note:
  - swap(double &x, double &y) doesn't exist until used
  - Compiler generates each version of swap *as needed*
  - As a result, template definitions have to be "visible" to compiler wherever they are used: they have to be #included!
- So, put them in a header file (more on this later)

Last words on swap()

- swap() is defined in standard library:
  - #include <algorithm> (pre-C++ 11)
  - #include <utility> (C++ 11 or later)
- swap() on complex types may be expensive:
  - Assignment operator/copy constructor used
  - May be overloaded by complex types (e.g. container classes) to be more efficient

Another Example: max()

- Also provided in standard library <algorithm>

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

- Note the use of T in the return value.
- Also, note operator< used – must be defined!

CLASS TEMPLATES

Class Templates

- What we've seen so far: *function templates*
- We can also define *class templates*

```
template <class K, class V>
class association {
public:
        K key;
        V value;
        association(K k, V v) {
                key = k; value = v;
        }
        association(association<K,V> &);
        void print();
};
```

association is basically just the pair template from the standard library, with different names for things.

CS@Mines

Defining Class Template Methods

Inside class declaration, type parameters are already known (see inline constructor on previous slide).

Outside class declaration, we have to declare our type parameterization all over again:

```
template <class K, class V>
void association<K,V>::print() {
        cout << key << "->" << value << endl;
}
```

CS@Mines

More on Defining Methods

```
// print method again
template <class K, class V>
void association<K,V>::print() {
      cout << key << "->" << value << endl;
}

template <class K, class V>
association<K,V>::association(association<K,V> &a)
{
      key = a.key;
      value = a.value;
}
```

CS@Mines

Using Class Template `association`

```
association<int, string> assoc1(4, "four");
association<int, string> assoc2(assoc1);

assoc1.print();
assoc2.print();

assoc1.key = 10;
assoc1.value = "ten";
assoc1.print();

// prints out:
4->four
4->four
10->ten
```

CS@Mines

Class Template Files

- Again, everything must be #included!
- Generally, everything needs to be in header file – class template declaration, method definitions, etc. There is no .cpp file!

CS@Mines

Array List Class

```
class array_list {
public:
        array_list();
        array_list(const array_list& lst);
        ~array_list();

        array_list& operator=(const array_list& lst);

        int size();
        int get(int index);
        void set(int index, int val);
        …
private:
        int*  _arr;
        int   _size;
        int   _capacity;
        void  _resize();
};
```

CS@Mines

Array List Template

```
template <class T>
class array_list {
public:
        array_list();
        array_list(const array_list<T>& lst);
        ~array_list();

        array_list& operator=(const array_list<T>& lst);

        int size();
        T get(int index);
        void set(int index, T val);
        …
private:
        T*    _arr;
        int   _size;
        int   _capacity;
        void  _resize();
};
```

CS@Mines

Array List Template Methods

```
template <class T>
array_list<T>::array_list() {
      _size = 0;
      _capacity = 1;
      _arr = new T[_capacity];
}

template <class T>
void array_list<T>::array_list(const array_list<T>& lst)
{
      deep_copy(lst);
}
```

CS@Mines

Array List Template Methods

```
template <class T>
void array_list<T>::_resize() {
        if (_size == _capacity) {
                _capacity = _capacity * 2;
                T* newarr = new T[_capacity];
                for (int j = 0; j < _size; j++)
                        newarr[j] = _arr[j];
                delete[] _arr;
                _arr = newarr;
        }
}

template <class T>
void array_list<T>::add(T val) {
        _resize();
        _arr[_size] = val;
        _size++;
}
```

CS@Mines

Up Next

- Reading for next week: Chapter 16 (16.5 is optional)
- Friday, April 5
  - Lab 11 – Queue, part 2
- Monday, April 8
  - Binary Trees
  - Lab 11 due

CS@Mines