CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 22
November 26, 2018

Templates

Templates

Template overview

Templates are instructions for generating code.

Are type-safe replacement for C macros.

Can be applied to functions or classes.

Allow for type variability.

Example:

```
template <class T>
class FlexArray { ...  };
```

Later, can instantiate

```
class RandString :  FlexArray<const char*> { ...  };
```

and use

```
FlexArray<const char*>::put(store.put(s, len));
```

Template functions

Definition:

```
template <class X> void swapargs(X& a, X& b) {
  X temp;
  temp = a;
  a = b;
  b = temp;
}
```

Use:

```
  int i,j;
  double x,y;
  char a, b;
  swapargs(i,j);
  swapargs(x,y);
  swapargs(a,b);
```

Specialization

Definition:

```
template <> void swapargs(int& a, int& b) {
  // different code
}
```

This overrides the template body for `int` arguments.

Template classes

Like functions, classes can be made into templates.

```
template <class T>
class FlexArray { ...  };
```

makes `FlexArray` into a template class.

When instantiated, it can be used just like any other class.

For a flex array of ints, the name is `FlexArray<int>`.

No implicit instantiation, unlike functions.

Compilation issues

Remote (non-inline) template functions must be compiled and linked for each instantiation.

Two possible solutions:

1. Put all template function definitions in the `.hpp` file along with the class definition.
2. Put template function definitions in a `.cpp` file as usual but explicitly instantiate.
   E.g., `template class FlexArray<int>;` forces compilation of the `int` instantiation of `FlexArray`.

Template parameters

Templates can have multiple parameters.

Example:

`template<class T, int size>` declares a template with two parameters, a type parameter `T` and an int parameter `size`.

Template parameters can also have default values.
Used when parameter is omitted.

Example:

`template<class T=int, int size=100> class A { ... }`.

`A<double>` instantiates `A` to type `A<double, 100>`.
`A<50>` instantiates `A` to type `A<int, 50>`.

Templatizing a class

Demo `22a-BarGraph-template` results from templatizing `Row` and `Cell` classes in `13-BarGraph`.
Template parameter `T` replaces uses of `Item` within `Row`.

Here is what was necessary to carry this out:

1. Fold the code from `row.cpp` into `row.hpp`.
2. Precede each class and function declaration (outside of class) with `template<class T>`.
3. Follow occurrences of `Row` with template argument `<Item>` in `Graph.hpp` and `Graph.cpp`.
4. Follow each use of `Row` with template argument `<T>` in `row.hpp`.

Using template classes

Demo `22b-Evaluate` is a simple expression evaluator based on a precedence parser.

It uses templates and derivation together by deriving a template class `Stack<T>` from the template class `FlexArray<T>`, which is a simplified version of `vector<T>`.

The precedence parser makes uses of two instantiations of `Stack<T>`:

1. `Stack<double> Ands;`
2. `Stack<Operator> Ators;`