Generics: C++ Templates

9 Mar 2009
CMPT166
Dr. Sean Ho
Trinity Western University

Generic programming

- Writing code (functions, classes) that works with multiple data types
- Write algorithm once, but apply to many types
  - e.g., define max(a, b):

```
int max(int a, int b) {        // other types?
    if (b > a) {
        return b;
    } else {
        return a;
    }
}
```

max() with const references

- A shorter max():
  - int max(int a, int b) {
    - return (b > a) ? b : a; }
- Use const references to operate on objects:
  - const string& max(
    const string& a, const string& b) {
    - return (b > a) ? b : a; }
- This means max():
  - Takes two parameters (call-by-ref) but does not change them
  - Returns a ref to an object that can't be changed

Static typing w/o templates

- In a statically-typed language like C++, extending this to other types needs duplication:
    - const MyObj& max(
      const MyObj& a, const MyObj& b) {
        - return (b > a) ? b : a; }
- Or a preprocessor macro:
    - #define MAX(a,b) ((b > a) ? b : a)
    - cout << MAX(2, 3);
    - cout << MAX(“hello”, “world”);
- Or void pointers (void *) cast to correct type
- Kludgy!

C++ templates

- The proper C++ solution is to use templates
- Type is given as a "template parameter" in the function declaration
- Templates are instantiated when the calling code specifies the type to use
- Two uses of templates:
  - Function templates:
    - e.g., max(a,b) taking any comparable type
  - Class templates:
    - e.g., vector<> of any type

Function templates

- Our max(a,b) function only requires that a and b be comparable: have a '>' operator defined
- Keyword 'template' in function declaration indicates that we are using templates:

```
template <typename Comparable>
const Comparable& max(
    const Comparable& a,
    const Comparable& b) {
        return (b > a) ? b : a;
}
```

- Comparable is the template type parameter

Using templates

- When we invoke the function template, we instantiate it with a particular type:
  - cout << max(4, 5);   // Comparable = int
  - cout << max(“hi”, “ho”);   // string
  - cout << max(Jane, Bob);
    // error: no '>' operator for Student
- max() is not a function, but a template
  - max(int& a, int& b) is a function
- Template instantiation done at compile-time
  - Compiler produces all needed instances

Templates and .cpp/.h files

- Usually when we declare a new class, we put the class declaration (with declarations for member methods) in a *.h file
- Code (bodies of methods) goes in *.cpp file
- But because templates are instanatiated at compile-time, templated classes need to be declared and defined in same header file
  - This is how Python and Java usually do things

Using templates: arguments

```
template <typename Comparable>
const Comparable& max(
    const Comparable& a,
    const Comparable& b) { .... }
```

- Note that a and b are required to be same type

```
max(3, 5.5)       // compile-time error!
```

- Solutions:

```
max( (double) 3, 5.5 )    // static_cast<double>
max<double>(3, 5.5)       // instantiated
```

Multiple template parameters

- Template parameters need not be types:

```
template <typename Elt, unsigned N>
  class NDpt {
        public:
              Elt pt[N];
  }
```

- Instantiating with Elt=float and N=3:

```
NDpt<float, 3> pt3d;
pt3d.pt[0] = 17.0;
pt3d.pt[1] = -5.3;
pt3d.pt[2] = 0.5;
```

Methods in templated classes

- Return unit-vector copy of point:
  - **template <typename Elt, unsigned N> class NDpt {**
    - **NDpt<Elt,N> normalize() {**
    - ** NDpt<Elt,N> newpt;**
    - ** for (int i=0; i<N; i++) newpt.pt[i] = pt[i];**
    - ** return newpt;**
    - **}**

Templated classes vs. functions

- Classes are templated slightly differently from functions:
  - Class template params can't be deduced (no arguments): specify explicitly
    - NDpt<double, 3>
  - Class template params may have defaults:
    - template <typename T=int>
  - Class templates may be partially specialized:
    - template <typename T>
    - class NDpt<T, 3> { …. }