Lecture 13a
Templates!!!
April 18th, 2016

Announcements
• Reading Assignment
  – Chapter 7 for today
  – Chapter 10 for next Monday
• Recitations: Version Control (subversion)
• PA8 is due Wednesday
  – Any questions?

Where are we?
• 1st part: memory & memory management
• 2nd part: Objected oriented programming
• 3rd part: Templates!
  – But don't forget memory management
  – I still owe you two more memory management strategies...
• Let's say you want to build a vector class in
Java that can hold anything
  – Forget that Java already has one

Data Type Abstraction in Java
• Approach #1:
  – Polymorphism: build a vector of objects
  – Upsides
    • Simple
  – Downsides:
    • Can't store primitives (except in wrappers)
    • Data type errors don't appear until run-time
    • No locality of references
    • Lot's of down-casting in your code
      – With run-time type checking
    • No optimizations based on data type

Data Type Abstraction in Java
• Approach #2: Generics
  – Tells the compiler what types of objects will be stored
  – Upsides
    • No need for down-casting
    • Data-type errors move from run-time to compile-time
  – Downsides
    • Can't store primitives (except in wrappers)
    • No locality of references
    • No optimizations based on data type

Data Type Abstraction in C++
• Java data type abstraction is based on polymorphism
  – With or without generics
• C++ also provides polymorphism
  – But without a single inheritance hierarchy
  – Less useful for storing a vector of anything
• C++ provides a new mechanism: templates
  – Syntactically similar to generics
  – But meta-programming, not polymorphism
Template = Meta-code
(Example: Max)

template<
typename TYPE>
TYPE Max(TYPE arg1, TYPE arg2)
{
    if (arg1 > arg2) return arg1;
    else return arg2;
}

About Templates
• The template command applies to the next C++
  expression
  – In this case, the definition of Max
  – Use template once for every function/method being
  templated
• The template command is an instruction to the
  compiler
  – The Max example generates no code when it is read by
  the compiler

"If a program uses a function called Max on two
arguments of the same type, and you don't already have a
function that matches this specification, here is how you
should define it."

Using Templates
int a = foo();
int b = bar();
int c = Max(a,b);

-----------------------------------------------
int Max(int arg1, int arg2)
{
    if (arg1 > arg2) return arg1;
    else return arg2;
}

What the compiler does...
1. Check for a function called Max(int, int)
2. If none, check for a template that could
   match Max(int, int)
   – Yes!
     • TYPE -> int
     • Max(TYPE, TYPE) -> Max(int, int)
3. Generate code from template
4. Recursively call the compiler on the
   generated code

Recursive Compilation
• In the recursive compiler call...
  – The data types are known at compile-time
    • No polymorphism is used
  – The function/method is fully optimized for the
    data types provided
  – Example: it knows what version of '>' to statically
    dispatch in Max
  – Errors will be caught at compile-time
  – New templates can be recursively expanded, if
    needed

Using Templates (II)
Quagga a = foo();
Quagga b = bar();
Quagga c = Max(a,b);

-----------------------------------------------
Quagga Max(Quagga arg1, Quagga arg2)
{
    if (arg1 > arg2) return arg1;
    else return arg2;
}
What happens

- Compile-time error
  - ‘>’ not defined for Quaggas
  - But with operator overloading, it could be...
    - So define Quagga::operator >> and recompile
    - Warning: template error messages are ugly
- Now it works, but is it efficient?
  - Arguments are passed by value
    - ... and therefore copied
  - Return value is also copied
  - So write a better template

A better version...

```
template<typename TYPE>
TYPE& Max(TYPE& arg1, TYPE& arg2)
{
    if (arg1 > arg2) return arg1;
    else return arg2;
}
```

Notes

- When the arguments were primitives, we didn’t care how they were passed
  - But with templates, we don’t know
- Quagga is a data type; so is const Quagga
  - So this will work either way
  - Quagga can be implicitly cast to const Quagga, so this work for a Quagga & a const Quagga
    - Bit in this case, will return a const Quagga....