Announcements

- Midterm #2
  - I have them with me, if you didn’t pick them up
- Reading Assignment
  - Chapter 7 (with Quiz) for next Tuesday
- PA7 due Thursday
  - 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’ 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
  - Example : build a vector of objects
  - Upsides
    - Simple
  - Downsides:
    - Can’t store primitives (except in wrappers)
    - Lot’s of down-casting in your code
      - With run-time type checking
    - Data type errors don’t appear until run-time
    - 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)

```cpp
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

```cpp
int a = foo();
int b = bar();
int c = Max(a, b);
```

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
    - E.g.: 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)

```cpp
Quagga a = foo();
Quagga b = bar();
Quagga c = Max(a, b);
```

```cpp
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 and a const Quagga
    • But in this case, will return a const Quagga....

Finally... a real vector class!

template<typename ELEMENT>
class vector {
public:
  vector(unsigned int sz = 0);
  ~vector();
  void push_back(ELEMENT element);
  ELEMENT& operator [] (unsigned int index);
private:
  unsigned int size;
  unsigned int capacity;
  ELEMENT* data;
};

More notes...

• ‘template’ still applies to one C++ expression
  – This time, a class definition
• ELEMENT appears as
  – An argument (to push_back)
  – A return type (operator [])
  – A data field (data)
• ELEMENT* and ELEMENT& are OK
  – Because when you substitute ELEMENT, you get a
    legal data type
• Some methods/fields do not use ELEMENT
  – ~vector()
  – size, capacity

Implementing vector

Template<typename ELEMENT>
Vector<ELEMENT>::vector(unsigned int sz)
  : size(sz), capacity(2*sz)
{
  if (sz != 0) data = new ELEMENT[capacity];
  else data = NULL;
}
Notes

• Implemented like a method
• ELEMENT can be used as a data type
• Fully-qualified class name is vector<ELEMENT>
  – There is no class called vector
  – Only vector<TYPE> for TYPES

Push_back

template<typename ELEMENT>
void vector<ELEMENT>::push_back(ELEMENT element)
{
  if ((size+1) < capacity) {
    data[size] = element;
    size++;
  } else {
    capacity * = 2;
    ELEMENT* new_data = new ELEMENT[capacity];
    for(unsigned int i = 0; i < size; i++) {
      new_data[i] = data[i];
    }
    delete [] data;
    data = new_data;
    data[size] = element;
    size++;
  }
}

Where to put templates?

• Put the entire template class – definition and implementation – in one .tpp file.
  – This is called the Borland convention
  – If the .tpp file is included in multiple .cpp files, it may cause the same functions to be multiply defined
  – In the Borland convention, the linker sorts this out
    • The GNU linker supports the Borland convention
    • So does Apple
    • The STL uses it too (except the .tpp filename)
  – Downside: slows the compiler
    • As templates are compiled many times

Alternative Conventions

• The CFront Model
  – Templated class definition goes into .h file
  – Templated implementation goes into .cpp file
  – Programmers add explicit instantiations as needed at the end of .h
  – Downside: need to modify .h for every program
• A new model
  – Templates split into .h and .cpp (like CFront)
  – Applications use a #pragma to declare instantiations
  – A utility function called from makefile scans for pragmas and creates the necessary .h and .cpp files
  – Downside: tools not yet in circulation