

Prototypes & Reuse

April 4, 2005

I. Administration

Midterm: April 18, in class

II. Reuse via Templates

Templates (polymorphism) allow abstraction across types. Reuse occurs because the operation is not dependent on the type. Best example is container classes, such as Sets, Arrays, etc.

III. Reuse via Inheritance

Common base classes can also be used to provide reuse across types: just make a linked list of class Base and then you have a linked list for any object (that is derived from Base). This is the java approach.

```
class LL public Base {
  private:
    LL *Next;
    Base *Object;
  public:
    void insertFront(const Base& obj);
    Base& removeFront();
};
```

Main problem: when I get an object from removeFront, what is its type? We don't really know... If we think we know (statically) then we can just cast it, but if we're wrong...

Java improvement: instanceof operator:

```
if (obj instanceof Customer) { }
```

IV. Reuse via Interfaces

The callee typically isn't reused at all. Instead the caller is reused by many callees. Any client with a Sort-Interface can reuse the sorting code. Any object with a PrintInterface and reuse various print styles/formats.

Prototypes and Reuse

V. Comparison

Templates are better if the type really is abstract. The compiler will correctly enforce the types for you.

If the type matters, then you can't use a template as easily. Base classes can capture key properties of an object more easily (and much clearer typically).

Example: list of objects that are printable...

Templates: invoke `T.print()` -- if `T` has no print method it is a link time error (typically). Even if it has one, the caller (the template code) may or may not meet the contract for that particular "print" (since we don't know which one it is).

Base class: abstract method `print()`. Now we get compile time checking and we also have an explicit contract of what the `print()` method is supposed to do... (Template case there is no obvious contract to follow.)

Yes, they can work together well...

VI. Arguments and Returns Types

Pointers are easy. Reference types (in C++) are pointers internally but can't be NULL. Means "pass by reference" for arguments and return values.

Return "T" means allocate space for T on the caller stack and copy the result into it.

Return "T&" means allocate space for T on the caller stack and use it as the working copy for the callee (just like ref args, the caller overwrites the original).

Return T* means allocate space only for T* and copy the resulting pointer into it.

VII. Prototypes

Goal: lookup an item in a collection based on some key

Question: what should the prototype of the lookup method be?

1. `T lookup(Key k)`
First problem, key should be const pass by reference: `T lookup(Key &k const)`. Second problem, what if `k` is not in the set?
Java: `T` can be null, no need for ref types, since everything is pass by reference
2. `T* lookup(Key &k const)`
Returns NULL if no matches. Problem: where is `T*` allocated and deallocated? What can the object do about rearranging items in the set? (nothing, since others may have pointers to `T`'s, they can't be moved, ever). Not possible in Java, since there are no pointer types.
3. `T lookup(Key &k const, bool* b)`
`T` is undefined if `k` is not in the set. Doesn't work in java. Better variation: `T&` as return type avoids a copy; `T` as return type implies 2 copies, one during the return and one during the inevitable assignment operator.

Prototypes and Reuse

4. `bool lookup(Key &k const, T& value)`
What is value of T& if no match? Forces copy of T into caller's copy. (copy semantics) This solves allocation problem, but does require an extra copy. Works fine in java (without the "&"s).
5. `bool lookup(Key &k const, T* value)`
This is almost the same, but slightly more risky since T* must be a valid T pointer. This is not too bad in practice, since normally it would be called with "&t" as the arg. Not possible in java...
6. `bool lookup(Key &k const, T** value)`
T* can be NULL if no match and does not require copy. Reintroduces alloc concerns. Not possible in java.

VIII.Binary Ops

Difference between `a += b` and `a = a + b`. The former avoid temporary variables and copies. For example, operator+(T& a, T& b) *must* create a new object of type T, since the return type has type T and cannot be either a or b. The compound assignment operators reuse the storage of a for the result.