CSE 333 Systems Programming *Professor: John Zahorjan*

Quiz 2 Solution

Name: John Zahorjan *Date: Thursday, 11/21/2013*

We use the term **class** loosely in what follows. **Data type** would probably be more accurate, but it doesn't read as well.

1. One way to build generics is to identify some lowest-common-denominator data type and define method prototypes that rely only on that type. (For example, in C/C++C we use void*; in Java we use Object.) Circle the most important limitation of this approach that motivates doing something more sophisticated (like templates).

   (A) There may be some classes that aren't subclasses of the assumed lowest-common-denominator class, and we therefore can't use the generic implementations with them.

   (B) There's no way for the generic library to do anything more with the objects it given than store them.

   (C) There is a way to allow the library to do more than just store the objects, but it's clumsy to use.

   Solution: *(C)*

   Callbacks provide a method for the library routine to defer type specific operations to caller-supplied methods. Anything beyond very simple manipulations may require many, small, callback functions to be defined and passed, though.

2. Indicate whether each statement below is true or false for C++ templates.

   (A) Every templated method can operate on any class.

   (B) A library based on templates must be distributed as source code.

   (C) Templated methods are likely to be much slower than equivalent methods based on some common base class.

   (D) Template libraries can lead to much larger executables (in terms of the amount of main memory required to run efficiently) than equivalent libraries based on some common base class.

   (E) It's much easier to understand the conditions under which a non-templated library routine functions correctly than it is to understand when a templated method functions correctly.

   Solution:

(A) False. If the template performs any operation on the object (e.g., copy), the object must support it.

(B) True. The template code is compiled at the same time as the calling code.

(C) False. Templates specialize the implementation to a class, potentially allowing discovery of optimizations at compile time, whereas the base class method relies on virtual methods, which are slow to invoke and impede optimizations.

(D) True. With templates there is an instance of the library for each type it is applied to.

(E) True. The non-templated implementation declares a type for the arguments, so you can examine the interface for that type. The templated code gives no clue what the type might be, so you have to examine the full source of the implementation to determine what it relies on, and even then you may end up with only a vague idea.

3. Two versions of a library method have been written. One requires argument objects to be from subclasses of class Foo and the other uses templates. Both versions perform assignment on the argument objects, and in each case the author assumed that the assignment operator had its conventional meaning. Assignment has not been overloaded in class Foo or any of its superclasses. Unexpectedly, a client program manages to pass in an object of a class that has redefined assignment, and that does something totally unexpected for it. Which statement below is true?

   (A) Both implementations break.

   (B) The class based approach breaks but the template based one works.

   (C) The class based approach works but the templated based one breaks.

   (D) Both implementations work.

   Solution: *(C)*

   In the class based approach, the code was compiled long ago, and the generated code uses standard semantics for assignment. The template is compiled with the calling code, and uses the whatever assignment semantics it imposes.

4. Explain briefly but convincingly why it's not possible to compile the code below. (Note: this is a question about templates. Answers not having to do with non-trivial template concepts aren't answers to this question.)

```
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;
```

```
template <int N>
void filter(list<int> L) {
  for ( int i : L ) {
    if ( i < N ) cout << endl;
  }
}

int main(int argc, char *argv[]) {
  list<int> L = { 10, 20, 40, 80, 160, 320, 640, 1280};
  int N = atoi(argv[1]);
  filter<N>(L);
  return 0;
}
```

Solution: *To expand the template into a specific realization the compiler needs the value of N, but that value isn't known until run time.*