Review Answers

I promised answers to these questions, so here they are. I'll probably readdress the most important of these over and over again, so it's not terribly important that you memorize all of these.

Objects: Idioms and Best Practices

5. What is Resource-Acquisition-Is-Initialization (RAII)? Why should you use it?

   Ans: RAII describes a core concept of C++ programming. In brief, one should acquire resources (memory, file handles, threads) as soon as (but no sooner than) you are ready to use that resource. The destructor should release the resource, and you should have the object go out of scope (and thus call the destructor) as soon as you are ready.

6. Could you implement `operator=` for a container class?

   Ans: The most basic form looks something like:
   This version is not necessarily exception-safe. We'll discuss how to fix that in the first lecture or two.

```
const MyObj & MyObj::operator=(const MyObj & rhs)
{
   if(this != &rhs) // This isn't guaranteed to work...
   {
        // this.obj1_ = rhs.obj1_;
        // this.ptr1_ = new Ptr1Type(*rhs.ptr1_);
   }
   return *this;
}
```

7. When should a destructor be `virtual`?

   Ans: Whenever the class might be derived from and delete'd as a pointer-to-base. Otherwise, you might get behavior like:

```
Base * b = new Derived; // Derived has a new string resource.
delete b; // calls Base::~Base()! The string is leaked!
```

8. What's wrong with this class? (Hint: What's missing?)

```
class log_file
{
```

```
private:
   FILE * file_;
public:
   log_file() // Ignore fopen failures...
   : file_(fopen("path/to/my/log","W"))
   {
        fprintf(file_,"Log opened.\n\r");
   }
   ~log_file()
   {
        fprintf(file_,"Log closed.\n\r");
        fclose(file_);
   }
   void print_message(const char * mesg)
   {
        fprintf(file_,mesg);
   }
};
```

Ans: If someone creates a `log_file log1("my_path")` and then also a `log2(log1),` then the compiler will automatically synthesize a copy constructor that does a bitwise copy of the structures. However, we don't want to bitwise copy a `FILE *`, because we'd then try to `fclose` it twice. (Why?) In this case, we'd either have to provide a copy constructor that does the right thing, or declare it `private` and not implement it. We'd need to do the same thing with `operator=`. In general, if you declare (and possibly implement) either a destructor, a copy constructor, or a destructor, you probably need to properly declare (and possibly implement) all three.

Templates

9. Could you implement an STL-style iterator? (with, say, a standard library reference?)

   Ans: If you need help with this, let me know and I'll provide some references. This is mostly material I'm assuming you've covered at one point somewhere. It's non-essential to this class for the most part. (Unless you want to do something STL-y for your final project.)

10. What's `typename` for? Do you know when to use it?

    Ans: In a sense, when writing templates, whenever a type "depends" on a template parameter for the status of being a type, you have to write `typename`. This isn't a great description, but an example will help: (This is taken from Jerry's List iterator code, so it may look familiar.)

```
template <typename T>
typename mylist<T>::const_iterator mylist<T>::find(const T&
                          elem) const
{
  return find<typename mylist<T>::const_iterator, const
          mylist<T> *>(this, elem);
}
```

The idea is that the compiler has no idea what you're talking about when you're using a template. `mylist<T>::const_iterator` could be an integer constant or an `enum` for all it knows. You have to help the poor compiler out in these kinds of circumstances.

11. What kinds of things can be template parameters?

    Ans: In short, types, class templates, and everything that can be represented as an integer at compile time. This includes values of every integral type, enum constants, normal pointers, pointers to functions, pointers to member functions, even pointers to member variables. I'm sure I missed something.

12. Does Substitution-Failure-Is-Not-An-Error (SFINAE) mean anything to you?

    SFINAE is a principle that says that if a template cannot be instantiated on the given arguments (based on the signature), it is simply removed from the candidate template specializations. There are a number of good uses for this when combined with partial specialization. For instance, providing a version of `std::copy` that uses `memcpy` if all the template parameters are pointers and the type itself is a builtin and then providing the normal `for`-loop when those conditions aren't met. Again, we'll go into more detail later on.

13. What is a policy? What are traits?

    Policies are (normally) template classes passed as template parameters that describe how a class should behave. Policies might tell a class to use `memcpy`, or that we're in a multithreaded environment and locking is necessary. Traits are class templates that contain information about their type. For instance, std::char_traits<T> provides information about how to compare to characters of type T.

Exceptions

Ans: The Exceptions handout will be posted this weekend, with most of this and more.

Efficiency

14. What are the first two rules of optimization?

    Ans: 1) Don't, and 2) (for experts only) Don't yet.

15. What is the 80-20 rule?

    Ans: The 80-20 rule has a lot of applications, but the one I'm focused on is that 80% of the execution time of your program is spent in 20% of your code. This is important in that you should know what to optimize.

16. What is the compiler allowed to optimize away? (What is a "trivial" destructor?)

    Ans: The standard only mandates that the compiler mimic observable behavior, which is said to be reads and writes to `volatile` variables, and calls to IO facilities. Anything else can be optimized away.

17. Why are functions not `virtual` by default?

    Ans: Virtual functions require an additional pointer dereference. This violates C++'s principle of "you don't pay for what you don't use."

18. What's wrong with this code:

```
for(int i = 0; i < names.size(); ++i)
{
 std::string outstr = names[i] + " is taking CS93SI!";
 send_message(outstr);
}
```

    Ans: A number of things were mentioned, but no one caught what I was looking for. In short, outstr has a char * that is being allocated and freed under the hood every time this code is being called. This is completely unnecessary. Better would be (including the other suggestions)

```
std::string outstr;
const container::iterator end = names.end();
for(container::iterator i = 0; i != end ; ++i)
{
   outstr.assign(*i);
   outstr.append(" is taking CS93SI!");
   send_message(outstr);
}
```

Esoterica

19. Do you know what Argument-Dependent Lookup is? Two-Phase Name Lookup? Are you curious?

Argument-Dependent Lookup:

ADL, also called **Koenig Lookup**, essentially states that for non-qualified function calls, if any of the arguments' types reside in a different namespace, then you look first in those namespaces (as well the current one) when performing overload resolution. That's how this works. If you notice, I don't do using `std::cos`, but gcc automatically knows how to look it up, because it correctly performs ADL.

```
#include <iostream>
#include <complex>

int main()
{
std::cout << cos( (std::complex<float>(1)) ) << std::endl;
};
```

(Here, the compiler sees that std::cos is defined in complex, and since `complex<>` is defined in `std`, it knows to use that.

Two Phase Name Lookup has to do with how templates are instantiated, and is more complicated than it ought to be. Essentially, unqualified, non-dependent names (that is, names that do not depend on a type of a template parameter), are bound to the proper entity (class type, function, whatever) when the template is first seen, and dependent types are looked up when it's instantiated. (It makes sense: you want to parse templates early, but names that depend on the type of the object can't be parsed, so just lookup what you can.) This leads to rather odd behavior:

```
void f() {cout << "2-phase lookup!" << endl;}

template <class T>
struct Base
{
     void f() {cout << "Dependent phase lookup!" << endl;}
     void g() {/* do you know how to call global f here? */}
};

template <class T>
struct Child: public Base<T>
{
     // Note that if global f wasn't defined, we would have
     // a compile error.
     void CallGlobalF() { f();}

     void CallBaseF() {this->f();} // or Base<T>::f(i), or
                                   // whatever. The this
                                   // pointer depends on T.
};
```

```
int main()
{
     Child<int>().CallGlobalF();
     Child<int>().CallBaseF();
}
```

Some compilers don't support this correctly (VS 2003/7.1) but I think all of the current generation does.