CS 321 Lecture Notes

Orion Sky Lawlor
Department of Computer Science, University of Alaska at Fairbanks
http://lawlor.cs.uaf.edu/ olawlor olawlor@acm.org

1 2005/01/26 Lecture Notes

We discussed libraries, specifically the libraries you'll have to build for your semester projects. The most difficult part about writing libraries is designing a good set of interface calls (often called the "Application Programmer's Interface", or API).

The whole purpose of writing a library is to simplify some aspect of the user's life. Simplicity comes in a variety of forms, though:

- Small is good, big is bad. A good interface is small–it doesn't have many routines, and the routines don't have many parameters. This makes the interface easier to learn and remember, because there's just less of it.

- Clarity is good, confusion is bad. In a clear interface, it's obvious what each routine and parameter is supposed to do. For example, a parameter named "i" could mean anything—confusion reigns. A parameter named "nItems", though, clearly means the number of items.

- Generality is good. The interface aught to make it possible to add related functionality later, without significantly changing the interface. Too much generality can make an interface big and confusing, but too little makes it "brittle" (prone to stop working after minor changes) and the interface soon stops being useful (or stops being small).

Designing a good interface is something of an art.

1.1 Control Transfers: Callbacks

We also discussed several different ways control can be transferred between parts of a program—and especially in libraries. The simplest control transfer is the function call, which everybody should be familiar with.

1.1.1 C Function Pointer

Function calls are a great way for lots of different code to call a library. But what about when you need the opposite case—when your library has to call a bunch of different code?

The classic solution to this is for your library to use a "function pointer", which is just a way for your library to store a pointer to a user function. The libary can then call the user's function without actually knowing its name—this allows the same libary to be used from several different places, by just passing different functions in via the function pointer. In C/C++, function pointers are just ordinary variables, so you can pass them to subroutines, store them in arrays, etc.

An example use of function pointers is like this:

```
/* Function to call if something goes wrong */
typedef void (*errorFunction)(double what,int *details);

void myLibraryRoutine(errorFunction err) {
...
err(what,mydetails); /* ''err'' points to myErr */
}

void myErr(double w,int *d) /* this is the code to call via a function pointer */
{
...
}

int main() {
myLibraryRoutine(myErr);
}
```

It's considered good style in C that any time you take a function pointer, you should also take a "void *". The pointer is used by the user to store any related data needed by the function. Then when you call the function pointer, you pass the "void *" into the function.

1.1.2 C++ Superclass

Because the C syntax is so funny-looking for function pointers, some people (including me) prefer to use C++ subclasses to do the same thing. Here, the library defines a C++ class (often an "abstract" class that can't actually be created) that contains some virtual methods. The user then creates a subclass of the library's class, overrides the virtual methods to do whatever he wants, and then passes his class into the library. The library then calls the class's methods, which end up executing the user's code.

```
/* Function to call if something goes wrong */
class errorClass {
public:
virtual void hitError(double what,int *details) =0;
};

void myLibraryRoutine(errorClass *err) {
...
err->hitError(what,mydetails);
}

class myErrorClass : public errorClass {
public:
virtual void hitError(double w,int *d)
{
...
}
};

int main() {
myErrorClass *e=new myErrorClass;
myLibraryRoutine(e);
}
```

There are various problems you can encounter with superclasses, such as forgetting the "virtual" keyword (this makes all calls end up in your library, not the user code!), copying the superclass instead of passing a pointer or reference (this is slow, and can cause crashes by "slicing" off pieces of the subclass), or having users mis-type the virtual method's name. Subclasses are a common and useful idiom in C++, though, and they're much nicer looking than function pointers.