

Lecture Topics

- module design: C to C++ [continued]
 - how C++ addresses C's pitfalls
 - example: NetLink [NetLinkCC to avoid Windows mishaps]
- exception handling (part today, remainder on Thurs)

Administrivia

- EWS accounts are ready.
- Lectures 1-4 are available, and Lecture 1 is now typed instead of scanned.

How C++ Addresses C's Pitfalls

- review of C module design pitfalls [try to keep on board and relate to next list]
 - [from Lecture 2]
 - language unaware of association between structure and functions
 - forced to pick “unique” prefix in global namespace
 - forced to use one file to avoid global scope for internal data
 - hiding structure implies only allowing dynamic storage class
 - use of module initialization/teardown is difficult to enforce with low overhead

 - [from Lecture 3]
 - choose between
 - fast and convenient OR slow and opaque
 - easy to change OR easy to read
 - control file organization OR control visibility (as above)
 - structure sub-types
 - compiler can't enforce proper field organization
 - type conversion hides some kinds of errors
 - again, the file organization issue
 - function variations over sub-types
 - type conversion again hides some errors

- benefits of classes, named scopes, and access control
 - function names
 - no need for prefixes
 - class name is implicitly part of function name
 - file organization
 - orthogonal to module design
 - left to programmer style, as it should be
 - instance and module initialization/teardown
 - captured naturally and transparently
 - encourages centralized definitions
 - enables use with static and local variables!
 - helps to avoid most related errors
 - but be aware: functions are being called on your behalf!
 - performance vs. centralization for simple fields
 - defining functions within class definition allows inlining
 - change you mind at any time by changing only the class definition!
 - variations on a theme
 - classes can inherit fields and functions
 - compiler handles derived class organization, so programmer can't break it by accident
 - functions can be virtualized
 - same layout, same functionality, no function pointer syntax!
 - type casts are safe and checked
 - automatic success from derived pointer to base pointer
 - implicit cast from base pointer to derived pointer when derived class virtual function is called
 - no other casts allowed implicitly (and thus checked)
 - no explicit casts necessary (and thus, again, checked)

NetLink Module in C++

- namespace `network_link_module` (no prefixes)
- constants (error values and TCP port numbers) defined in namespace
- class hierarchy uses types to differentiate connections and servers
 - base `NetLinkEndpoint` for common data + functions
 - automatic module initialization [see below]
 - Use of base class means only one constructor in which to place this code.
 - Some runtimes print name of unhandled exception, so long names can be useful.
 - assertions changed to exceptions (more on this topic later)
 - automatic call to close (in destructor)
 - simple public functions all inlined

`private:`

```
static err_t initialize (); // module init function
static err_t init_success; // initialized to UNINITIALIZED
```

`protected:`

```
NetLinkEndpoint () {
    if (UNINITIALIZED == init_success) {
        init_success = initialize ();
    }
    if (INIT_FAILED == init_success) {
        throw NetLinkInitializationFailedException ();
    }
    ... // actual constructor
}
```

- class hierarchy [continued]
 - derived NetLink class for connections
 - three kinds of constructors
 - using IP address and port number
 - using host name and port number
 - using server (accept incoming connection)
 - retrieve IP address (port number call is in NetLinkEndpoint)
 - read/write calls
 - derived NetLinkServer class for server sockets
 - constructor uses port number
 - NetLink constructor that accepts connection is a friend of this class (needs to access base class variables)

- total code length
 - header is larger
 - implementation is smaller
 - overall slightly smaller

- relative clarity
 - name visibility is much cleaner
 - use model is somewhat cleaner
 - extra type checking
 - automatic module initialization
 - deadlock issue remains

- C++ warnings helped find several potential problems in C
 - unsigned vs. signed int in read/write (I knew already, but hadn't fixed)
 - use of special type for **accept** argument
 - using void* for pointer arithmetic

Exception Handling

- read Ch. 16 of Stroustrup
- meaning of exception
 - an exceptional/unexpected condition in software
 - occurs infrequently relative to function calls
 - language-level concept
 - can be named
 - handlers can be written and applied to specific instances
- in contrast
 - not in the hardware exception sense (e.g., DIV0)
 - although these could give rise to software exceptions
- subtopics
 - using goto
 - setjmp/longjmp
 - cleanups
 - C++

The One Use of Goto

- Tolerable failures usually imply releasing resources...
 - may need to close files
 - may need to free memory
 - may need to undo data structure changes
- Sometimes there are lots of ways to fail
 - ideally, but not always, block structured (block structure is relatively easy)
 - acquire resource 1
 - acquire resource 2
 - acquire resource 3
 - Success! Return...
 - release resource 3
 - release resource 2
 - release resource 1
 - when not block structured
 - each failure necessitates releasing some set of resources
 - probably have several failures with common sets
 - again ideally, order checks according to groups
 - avoid replicating failure handling code
- some cases have a “natural” ordering or constraints on ordering that preclude
 - block structure
 - grouping according to resources to be released
- Code replication is bad. Instead, use goto.

- example from the Tcl sources (tclCmdAH.c)
 - Tcl file object command: file <option> <args> ...
 - 565 lines of code, most of which is one big switch statement
 - 34 options, most of which take one argument
 - if someone passes something other than one, print exactly the same message...

```
switch ((enum options) index) {
    ...

    case FCMD_WRITABLE:
        if (objc != 3) {
            goto only3Args;
        }
        return CheckAccess (interp, objv[2], W_OK);
}
```

```
only3Args:
    Tcl_WrongNumArgs (interp, 2, objv, "name");
    return TCL_ERROR;
}
```