Lecture Topics

- module design: C to C++ [continued]
  - variations on a theme
  - example: netlink
  - C++ features

Administrivia

- Use EWS accounts for labs (should have enough space).

Variations on a Theme

- Sometimes useful to define several sub-types of a data structure
- In 391 layout tool, for example
  - drawing widgets include gates, clocks, and other components
  - some shared aspects
  - common interface (e.g., draw, undraw, update signals, etc.)

- What's the approach in C?
  - structure organization [draw a picture]
    - common fields organized into a structure (component_t)
    - each specific structure starts with a component_t structure
    - thus casting pointer to specific structure (e.g., AND_gate_t*) into a component_t* and using with component functions works fine

[ABOVE THIS LINE COVERED LAST THURSDAY]

  - pitfalls
    - compiler knows nothing about structure organization
      - adding field before the component_t field breaks the code
      - removing the component_t field breaks the code
      - compiler unlikely to notice either error
    - type conversion is not clean
      - Do component_t functions take void*?
      - Or do sub-types need to be explicitly cast?
      - Either answer can hide errors.
    - component_t structure definition
      - must be known to define AND_gate_t, OR_gate_t, etc.
      - One big file?
      - Globally visible?
      - Or preprocessor hacking?

- function organization

  - simple (and slow) option
    - use a “subtype” field in component_t
    - switch statement forms the core of each function
      - each case calls a separate function
      - or code coalesced into one big function (yikes!)

  - better option: use function pointers!
    - draw function is a function pointer field in component_t
      - for AND_gate_t instances, we point it to draw_AND_gate
      - for OR_gate_t instances, we point it to draw_OR_gate
    - invoke by simply using pointer

  - Have lots of sub-type-specific (“virtual”) functions?
    - build one table of function pointers per class
    - add a table pointer field to component_t
    - instead of a bunch of function pointer fields

- function organization pitfalls

  - type conversion not clean here, either
    - need void* for automatic conversion
    - compiler doesn't know about "hierarchy," so can't detect mis-use (sending int* instead of struct pointer by accident)

  - example: option 1

```
void draw_AND_gate (void* g, point_t* p);
void draw_OR_gate (void* g, point_t* p);
```

    - pointers to these two functions have a common type
    - compiler can type-check assignments to jump table fields
    - but g parameter
      - must be cast in each function before use
      - implicit is ok; compiler will agree to any pointer from void*

  - example: option 2

```
void draw_AND_gate (AND_gate_t* g, point_t* p);
void draw_OR_gate (OR_gate_t* g, point_t* p);
```

    - now function pointers do not match, and compiler cannot type-check assignment to jump table field (must either force using type cast or force arbitrary acceptance with void* fields)
    - but functions are "cleaner."

- somewhat cleaner
  - when the base structure has no data fields
  - simply consists of a “jump table” (an array of function pointers)

- some examples from 391/Linux
  - Programmable Interrupt Controller (PIC) interface
  - file operations structure

- C++ has language features that allow the compiler to support almost everything we’ve talked about in a more natural and less error-prone way
  - Stronger type checking, and little/no need for explicit casts
  - Introduction of access control in addition to visibility (scope): in C++, something in scope may still not be accessible
  - Makes file organization and scope organization orthogonal issues
  - Provides an arbitrary number of hierarchically nested named scopes, including some that are not block structured (i.e., you can add to them piecemeal)

Netlink Module in C

- BSD sockets interface in C
  - more or less unchanged since creation
  - plenty of artifacts from early Internet
    - Support a range of protocol families (e.g., PF_INET).
    - Use odd names for common protocols (e.g., SOCK_STREAM means TCP).
    - TCP blocks server port re-use for minutes by default
    - How long should the OS queue for incoming connections be?
    - Requires several calls to create a server or connection.
  - protocol details also exposed
    - host name to IP address mapping using DNS (old calls are non-reentrant, but new calls not always available)
    - port numbers for known services (see /etc/services)
    - which end is up in the network (network order is big-endian)
    - read/write call fragmentation (read/write can stop part-way for various reasons)
  - Domain Name Service API updated in 2004 (get/freeaddrinfo)

- many high-level languages provide simpler interfaces

- We'll write as a module in C (netlink)
  - support only TCP and IPv4
  - two kinds of net links: server and connections
  - can create a server and then accept connections
  - can connect to a server by host name and port strings or IP address and port numbers
  - can read/write over a connection (we'll fix the number of bytes for simplicity)

- visible aspects of the module
  - prefix “NL_” or “nl_”
  - enumeration of error values that can be returned
  - enumeration of TCP port numbers (for convenience)
  - declaration of netlink_t structure (opaque)

- functions for netlink servers

```
netlink_t* nl_create_server (uint16_t port);

netlink_t* nl_accept (netlink_t* server);
```

- functions for netlink connections

```
netlink_t* nl_connect_by_name (const char* host_name,
                               const char* tcp_port);

netlink_t* nl_connect_by_ip (uint32_t ip_addr,
                             uint16_t tcp_port);

nl_err_t nl_blocking_read (netlink_t* n, void* buf,
                           uint32_t n_bytes);

nl_err_t nl_blocking_write (netlink_t* n, void* buf,
                            uint32_t n_bytes);
```

- functions for destroying a netlink

```
nl_err_t nl_free (netlink_t* n);
```

- functions for getting information about a netlink

```
uint32_t nl_get_remote_ip_address (netlink_t* n);
uint16_t nl_get_tcp_port (netlink_t* n);
int32_t nl_is_connected (netlink_t* n);
int32_t nl_is_server (netlink_t* n);
```

- module initialization to handle SIGPIPE

```
nl_err_t nl_initialize ();
```

- that's it!
  - implementation sits in a single file (a little over 500 lines)
  - note that access functions aren't likely to be used often, so performance is not a huge concern
  - most issues are handled by module, not exposed to users, but…
  - there are some deadlock possibilities

- source code file includes
  - netlink_t structure definition
  - several internal functions
    - for closing, allocation, freeing, and connecting
    - mostly to avoid code replication

Module Design in C++

- What additional tools does C++ provide to help you develop modules?
  - a class is a module (as defined earlier); it defines
    - fields & static data
    - interface & internal functions
    - instance initalization/teardown functions (constructors/destructors)
  - module initialization/teardown
    - handled by generalizing static variable initialization
    - to allow non-constant initialization (e.g., function calls)
    - controlling inter-dependent module ordering is tricky (see Str Sec. 3.11.4)
    - usually sufficient to do a check in constructor
  - class also defines a named scope
    - symbols in the class' scope must be listed within a single block
    - but definitions can be spread out arbitrarily
  - distribution enabled by adding concept of access control for names
    - in C, scope = visibility and access rights
    - in C++, scope = visibility only, and does not imply access
    - protects against accidents, not fraud/malice
    - access granted by class to class/function (not individual objects); access cannot be taken from outside
    - access rights are per name (important for overloading)
  - keywords in a class
    - "private:" access allowed only within the class (field / impl. func.)
    - "protected:" access allowed within class and derived classes (as with private, field / impl. func.)
    - "public:" access allowed to anyone (interface func.)
    - "static" class function / variable
    - (not static) instance function with implicit pointer arg / field