

## **Lecture Topics**

- module design: C to C++ [continued]
  - basic design strategy in C
  - variations on a theme
  - example: netlink (partial today, finish Thursday)

## **Administrivia**

- Hand out homework #1 and discuss.
- Hand out netlink code.
- EWS accounts have been requested (100MB each).

- What if you
  - need to avoid call overhead on fields (e.g., for a 2D point, x and y)?
  - want to use a local variable?
  - Oh, heck, make the struct public!
- In other words, choose between
  - fast and convenient
  - slow and opaque
- What if you
  - need to know who's screwing up by putting points off the screen?
  - want a histogram of points by location to optimize your accesses?
  - Good luck editing 1000+ uses of "x" and "y"  
(ok, sure, rename the field and take your time)
  - Instead

```
#define GETX(p) ((p).x)
#define PUTX(p,v) ((p).x = (v))
```

The debugger won't translate, so pick a transparent convention.
- In other words, choose between
  - easy to change
  - easy to read

- What if you
  - want to use more than one source file?
  - want to include a structure as a sub-structure?
  - conditional exposure with the preprocessor!

```
typedef struct point_t point_t;
#if defined(POINT_PRIVILEGE)
struct point_t {
    int x;
    int y;
};
#endif
```
  - note
    - static data exposed in this way are now global variables (no other choice)
    - C does allow namespace conflicts
    - but debugger can't necessarily find the one you meant
      - step through "origin.x = 42;" and return to caller function
      - print origin.x
      - see "0"
- In other words, choose between
  - controlling file organization
  - controlling visibility (scope)

## Variations on a Theme

- Sometimes useful to define several sub-types of a data structure
- In 391 layout tool, for example [need to get part of code for this class]
  - 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
  - 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
  - [look up how actual layout code implementation is done!]

- 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)
- many high-level languages provide simpler interfaces
- We'll write as a module in C (netlink)
  - support only TCP
  - 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 or IP address and port
  - can read/write over a connection (we'll fix the number of bytes for simplicity)