Lecture Topics

• historical view of C++
• module design: C to C++ (today: start the C part)
  – C approach followed by example
  – C++ approach followed by example
  – (examples solve the same problem)

Administrivia

• web page
  – overview + lectures on web page
  – module code out
• first assignment out Tuesday

• [review] strategy adopted with C++
  – backwards compatible
  – new tools must be simple, portable, and leverage existing ones
  – no new features used implies zero overhead
  – new features have minimal overhead
• quote for external perspective (ISO TR18015 C++ Performance, p. 202)

• historical highlights of C++ (see notes/Str Ch. 0 for more detail
  – started in 1979 (7 years after C)
  – exponential growth (10x every 2 years) in late 80s
  – not standardized until 2003

• [notes only] brief history of C++
  – 1972: C language developed by Dennis Ritchie as tool to simplify task of writing Unix
  – May 1979: C with Classes begun as tool to simplify task of distributing Unix kernel over a LAN
  – October 1979
    – first implementation of Cpre preprocessor
    – language called “C with Classes”
  – December 1983: C++ selected as name
  – October 1985: Cfront release 1.0 (C++ to C compiler front-end)
  – November 1986: first commercial PC port (Glockenspiel)
  – 1986: > 1,000 users
  – 1988: > 10,000 users
  – 1989: ANSI C standard
  – June 1989: Cfront release 2.0
  – 1990: > 100,000 users (strategy enabled this rate of growth)
  – October 1991: Cfront release 3.0 (templates)
  – 2006: ISO TR18015 C++ Performance
• C++ strategy also enabled C to co-evolve with C++
  – ideas from C++ have been added back into C
  – supported by goal of compatibility to ease programmer transition
  – example
    • implicit types are deprecated
    • best practice now to be explicit
    • but old form still allowed
    • Example: what does the following mean?

        extern foo ();

        – there’s a global function foo that returns an int
        – but I don’t want to tell the compiler its arguments!

• finally
  – C++ strategy enables us to teach you C and have the knowledge have
direct value for C++
  – although by now you should understand programming as a process well
enough that you can learn any imperative language on your own
Module Design and Implementation Strategy

• Think about debugging a large program.

At some point, you’ve figured out why a problem occurs. What do you want from your code at that point?
  – easy to figure out what to change
  – easy to make the change correctly

• abstract properties you might want: we’ll come back to some later; for now, think warm and fuzzy thoughts

  – modularity (design part by part)
    • easier to design one part at a time
    • easier to think about in a focused way
    • simplifies your design (when used well; building a house from grains of sand is not fun)
    • enables broader applicability (and thus reuse)
    • rarely optimal, but even more rarely worth optimizing

  – extensibility
    • generally, you can accommodate what you plan to accommodate
    • other extensions usually hard
      (e.g., GDB hooks vs. parallel debugger with Mantis)

  – information/implementation hiding
    • avoid exposing/making/forgetting assumptions with clean abstraction boundaries
    • performance sometimes pokes through
    • synchronization often impossible to hide
• What tools does C provide to help you develop modules?

  – file organization (headers and sources)

  – scope
    • one global scope
    • one scope per file
    • one scope per block (function or compound statement)

  – storage class
    • static (global data area)
      – one permanent copy
      – initialized as data in executable image (to 0 unless overridden)
    • automatic (stack)
      – one copy per function invocation
      – allocated when stack frame set up
      – removed when stack frame torn down
      – not initialized
    • dynamic (heap)
      – allocated/removed using library calls
      – can be initialized to 0
In C, we organize modules around one or more data structures.

A data structure is:
- a structure with fields (or several)
- related static data
- interface functions on one or more instances
- internal/implementation functions
- instance initialization/teardown routines

module also includes: module initialization/teardown routines

C offers no language support for identifying related functions:
- try to pick a unique naming prefix for interface functions
- relative to all past and future C code
- usually 2-8 letters

hiding implementation means not using the global scope for:
- structure definitions
- related static data
- implementation functions

these things go into file scope, so
- we must use one file for them
- also:
  - no structure definition (outside of module file) means
  - structure size is unknown, so
  - no local/static variables, only dynamic
• guaranteeing proper use of C modules
  – assert at module boundaries (interface functions)
  – What about instance initialization/teardown?
    • embed functions with malloc/free
    • no one can use arrays or non-dynamic variables anyway
  – What about module initialization/teardown?
    • check init in all interface functions (init becomes internal)?
    • assert init in all interface functions?
    • cross fingers?
    • module teardown? Good luck!
• Write your own tool!
  – grep out __init and __teardown prefix functions
  – construct a function calling all of them
  – dump it into a new file
  – call that function
  – build it into make
• 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!

```c
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
  - 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 [next time]