Basic Project Hygiene

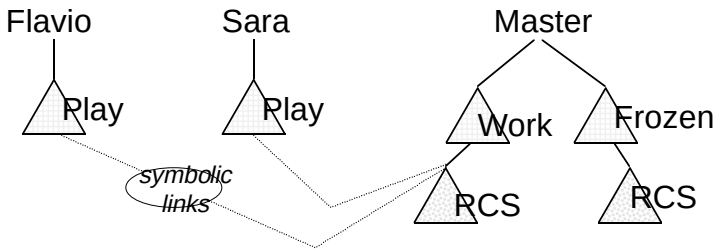
Change control, coding style, and other observations on avoiding messes

Change Control

- Typically three “builds” are current:
  - Frozen: The “demo” version (shared)
  - Work: The current integrated version (shared)
  - Play: Individual developer’s version
- Steps:
  - Programmer checks out module to “play”, makes changes and tests against “work” modules of others
  - Programmer checks in module when it has been tested against the “work” version (this may require coordination)
  - On a regular schedule, “Work” version is tested and moved to “Frozen” version

Version Management

- Use RCS, SCCS, or similar for version management and concurrency control (locking)
  - Have a policy on holding locks: e.g., 24 hours or less
    - Or use a "merging"-based revision control system like CVS
  - May need multiple RCS directories, or a protocol for indicating the components of "work" vs "frozen" versions



Distinguish "Derived" from "Source"

- All "ultimate source" should be under version/revision control
- All "derived" objects should be produced automatically (e.g., when you run "Make")
  - Never edit derived objects
    - Examples: Object code (obvious?), lex output
- When generating components, consider revision procedure
  - If post-generation changes are necessary, they should be saved and applied to revised version

Exploit High-Level Tools

- Use application generators, scripting languages, libraries, etc. when possible
  - Subject to constraints of portability, performance, etc.
- Consider generating parts of the application
  - Example: An Awk script can generate a large C struct initializer for error messages or help from a more easily editable text file
  - Message table in program and user manual could come from the same ultimate source
  - Remember then: generated code is “derived”, not “source”

Effective Unit Test

- A little is better than none
  - In a group project, the worst bugs are those from your teammates; yours are easier to find and fix
  - Never “leave it for integration”: Give your teammates clean, tested modules.
- Test drivers:
  - A fully automatic test driver should be part of delivered units, and should be re-run before turning over a change
    - After making a change, regression testing should be fully automatic (or it won’t be done)

Scaffolding

Build more than the application code

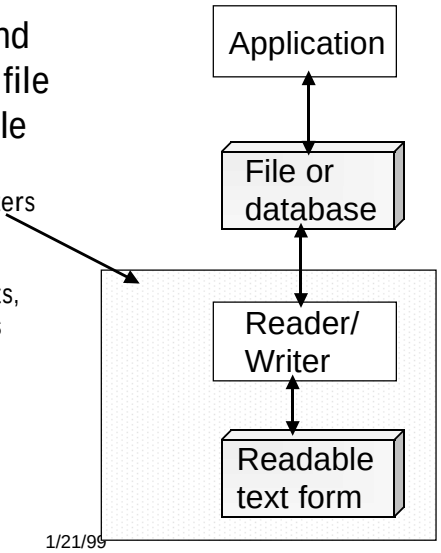
- Test drivers and cases
- Data structure viewers and validators
  - For any complex data structure, build tools to view it on demand, and to perform validity checks
- Instrumentation
  - Any performance-critical part of the program should be capable of measuring itself
- Stubs
  - Build "substitute" parts for testing and debugging

Avoid Inappropriate Optimization

- Consider efficiency only when/where needed
  - Efficiency is unimportant for many programs
    - Have a concrete performance goal, and a rationale for it.
    - KISS: If the goal can be met by simple algorithms and data structures, do not use complex algorithms and structures
  - 80/20 rule: Efficiency is unimportant for most parts of a program
    - Even if performance is a problem, it probably effects only a small part of the program. Identify them (by measuring), and put the effort where it matters.
  - Work at the highest possible level
    - Start with overall design, then algorithms and data structures; code-level optimizations should be used sparingly, if at all.

Prefer Readable, Editable Files

- Avoid binary files and other non-editable file structures if possible
  - If you must have them, provide readers & writers
- Why:
  - Debugging, experiments, prototypes, extensions
  - Breaking build-order dependencies



Compile-time Errors are Better than Run-time errors

- Principle: Whenever possible, help the compiler catch your errors
- Applications:
  - Strong typing (the stronger, the better)
    - Use explicit casts if necessary, rather than demoting types
  - Access functions rather than public data in module interfaces
    - Whenever you can classify "correct" and "wrong" ways to access the data
  - Volatility markers: const (C++), "in" mode (Ada), "final" (Java)

Suicide is Not a Sin (for Programs)

- "Defensive programming": check for errors and violated assumptions
- Better to quick death with a suicide note, than a lingering illness
  - In C, C++: the "assert" macro or Gnu *nana*
  - In Ada, Java, etc.: use exceptions
    - throw exception as close as possible to sign of trouble
  - Create "safe" versions of unsafe services
    - e.g., malloc/free with extra checking (for C/C++)

Programs are for Reading

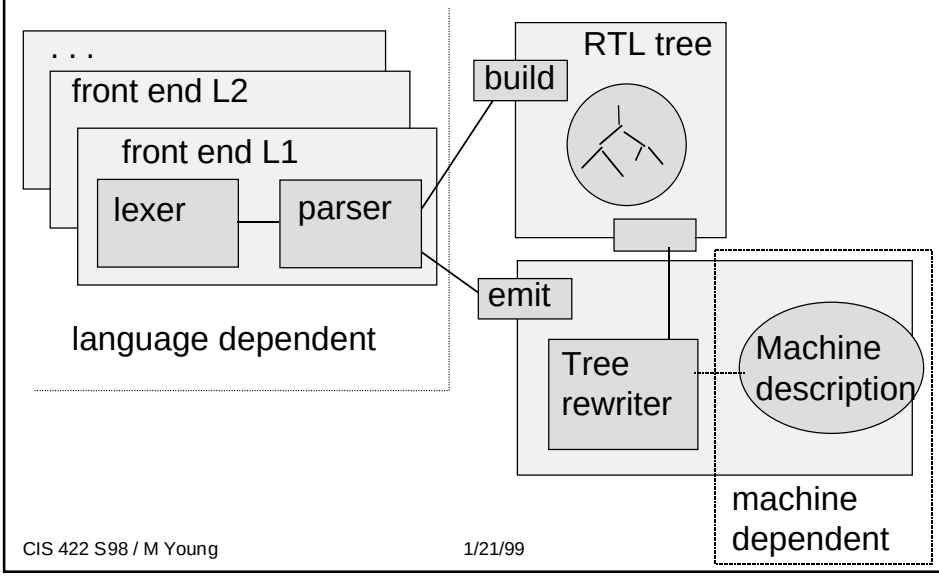
- Each line of code is written once, but read many times
  - Saving time in typing is a poor decision
- Code should be readable for the unfamiliar programmer (e.g., maintenance programmer)
  - Overall organization is more important than coding details; e.g., how can I separate the "front end" from "back end" files in the g++ compiler?
  - Within a file, most important is ability to scan for relevant parts

Architectural Overview

- "Orientation" documentation
  - What are the organizing principles for this system
  - What are the major pieces and their interfaces
  - Where are the parts making up those major pieces

An architectural diagram of GCC

(not entirely accurate)



Orientation to GCC …

- Front/back interface is (only)
  - construction of register-transfer-language tree
  - invoking code generator after each procedure
- Code generation for each machine is controlled by table (machdef.h)
- Should say where to look to answer questions:
  - How would I build a native code Java compiler?
  - How would I compile C to Java byte codes?

Comments

- Header comments: What I should know before reading the code
  - Consider extracting and indexing them, as in JavaDoc
- Code comments: What I might need to understand the code
  - Avoid restating the obvious
  - Help the reader "recover the design"
- More is not always better
  - But a ratio of 2-3 lines of comment for each line of code is often about right

Header Comments

- Interface comments: State the contract
  - What, not how
    - If an interface comment says "first this procedure does foo, then it does bar", then either the comment or the procedure is badly designed
- Design comments: Approach
  - In the implementation, not in the interface
  - Provide an overall view

Namespace is Precious

- The "name space" of a system is the set of available names (for programs, modules, files, variables, ...)
- In a large system, a "flat" namespace is quickly exhausted
  - "But there is already an object called Queue, so ..."
- Conserve namespace by ...
  - Partitioning (e.g., use local names in preference to global names, create hierarchy (structs, packages))
  - Using specific names with low likelihood of clash

Choosing Names

- The wider the scope, the longer the name
  - Global names (e.g., system constants, classes) should be very specific, even if the names are long and cumbersome
  - In very local scope, names may be shorter
    - i and j are perfectly good index variable names, for small loops
- Distinctness matters more than length
  - "Long_name_1" and "Long_name_2" are worse than "theta" and "gamma"
- Standards help
  - like_this or LikeThis or Like_This; GLOBAL or Global_

Pretty-Printing

- Pretty-printed code can be read more quickly
  - easier to "scan" for relevant parts
- Automatic pretty-printing exposes errors
  - choose a style in which common errors are obvious
  - color, fonts etc. help too: e.g., distinguishing comment from executable code.
- Consistency helps
  - Choose a team-wide standard, and stick to it