

Lecture Topics

- philosophical ruminations
- historical view of C++
- module design: C to C++ (today: most of the C part)

Administrivia

- will try to get first assignment out next week
- initial thoughts on midterm timing
 - EOH plans?
 - other exams on 3/5, 3/10, 3/11, 3/12?
- pass out note #1
- which form preferred for lecture notes
 - raw copies of my notes (faster)
 - extended version (clearer)
- 190 notes on C++ are available online

Philosophical Ruminations

- languages as engineering
 - (Str, p. 45) “...language design is not just design from first principles, but an art that requires experience, experiments, and sound engineering tradeoffs.”
 - (Str, p. 48) “...most people focus on syntax issues to the detriment of type issues. The critical issues in the design of C++ were always those of type, ambiguity, and access control, not those of syntax.”
 - people often like what they know
 - one reason that cycle between language research and widespread use is often ~20 years
 - anecdote about reviewed paper’s claim of aesthetic superiority
 - (Str, p. 38) “I always maintained a clear view of what an object looked like in memory and considered how language features affected operations on such objects.”
- one contrast: language as customer lock-in (Str, p. 37)
 - basic strategy is to create user dependence on revenue-generating activity
 - examples include
 - central support for tools
(development environment, compiler, debugger)
 - training
 - consultants

- philosophy and some potential drawbacks of C++
 - Java vs. C++
 - Java authors considered C++ to be the kitchen sink of languages, i.e., a language overflowing with constructs that at best any given programmer had used once, and most of which most programmers would never use at all.
 - Stroustrup believes in accommodating a wide range of programming styles, and C++ reflects that philosophy. (Str pp. 23-24)
 - exposing novice programmers to C++ can be risky
 - kid in a candy store phenomenon
 - chewing gum can be used as glue
 - guidance usually helpful or even necessary

Historical View of C++

- early goals (Str p. 21)
 - support for structure and program organization (classes, as in Simula)
 - enable fast programs and freedom to mix with other languages
 - BCPL used as example
 - historically of extreme importance in industry; also one of the reasons that IDEs (integrated development environments) have been slow to catch on
 - portability
 - use of C preprocessor and compiler
 - limited complexity (more detail on Str p. 37)
 - simple enough to attract users
 - simple enough to implement (~ one week to port compiler)
 - limited integration with OS

- attractive qualities of C (Str, p. 43)
 - flexible/expressive
 - efficient
 - available
 - portable (relatively)

- sources of C++ ideas (other than C)
 - Simula: classes, virtual functions
 - Algol: operator overloading, references, mixing statements+decls
 - BCPL: // comments
 - Ada: templates, exceptions, namespaces
 - Clu: exceptions
 - ML: exceptions
- (also considered Modula-2, Smalltalk, and Mesa)
- also see chart in chapter 0 (Str, p.6)

- strategy adopted
 - backwards compatible
 - new tools must be simple, portable, and leverage existing ones
 - no new features used implies zero overhead
 - new features have minimal overhead

- brief history of C++ (see Str Ch. 0 for details)
 - 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)
 - 1995, 1999: ANSI/ISO C standard revisions
 - 1998, 2003: ISO C++ standards

- 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

Topics for the First Five Weeks (from C to C++)

- module design and implementation strategy (review?)
- templates/type polymorphism
- container classes & iterators
- reference counting
- memory debug
- exception handling
- preprocessor use (maybe)

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