Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

Summary of “Modernizing Legacy C++ Code”

Marc Paterno

9 September 2020

Section 1

My summary

What is legacy code?

- Code that doesn't follow current "best practices"
  - so sometimes code is *legacy* the moment we write it
- Not necessarily old (but often is)
- What you are writing today is likely some day to be legacy
- *Should* we update legacy code? What about "If it ain't broke, don't fix it"?
  - Legacy code is hard to improve (add functionality, improve speed, etc.)
  - Updating it has a cost. Failure to update also has a continued cost.
- Updating legacy code can result in unexpected speed and memory use improvements.
  - I have found this to be a common effect.

Turn up the warning level

- Make your build *warning clear*.
- Better to have compiler find issues, rather than users — or paper readers.
- Treat warnings as errors.
  - Does your experiment already do this?
  - Can you convince them to do so, if they do not?

Q: What do you do if a header *you do not control* generates a compiler warning?

Q: What do you do if *your* legal and correct code generates a compiler warning?

Avoid conditional compilation

- Conditional compilation uses `#if defined` or similar preprocessor macros to include code.
- What should be used instead?
  - prefer function overload sets
  - prefer templates
- Kate and James suggest: *`#ifdef` entire functions*. I disagree.
- My favorite comment on this technique (I do not recall the author): “Congratulations, you have written platform multi-dependent code.”
- I prefer to leverage the build system:
  - Write functions (or classes, or templates) with the same name, and the same interface.
  - Have the build system choose (maybe with user guidance) which *one* gets compiled and linked (or just included, if all is in a header).

Avoid macros

- Why?
  - macros do not obey namespace rules
  - the preprocessor does not know about types
  - compiler error messages come from the generated code, not what the user sees
- What should be used instead?
  - prefer function overload sets
  - prefer templates
- Note that templates are not exactly a solution to the issue of poor error messages.
- Sometimes you need a macro Commonly appear as part of a plugin-handling system.

Q: What other good uses of macros have you encountered?

RAII and scope reduction

- “Housekeeping” boilerplate obscures logic (closing files, releasing DB connections, freeing memory, any other resource handling). Also, checking on “special” values of inputs.
- First suggestion: use `if (...)` and early return.
- But what about the structured programming rule of having a single point of exit for a function?

What was original idea behind single entry/single exit?

- Very old paper by Edsger Dijkstra introducing ideas of structured programming
- Promulgated in the era when subroutines were just being invented; really was talking about regions of code is a program without subroutines.
- Introduced the discipline necessary to make such code manageable.
  - especially for things like making sure all resources were correctly released
- In modern C++, *with ubiquitous use of RAII*, this is already handled.
- This is a rule for another era; it does not hold for modern C++.

Use exceptions

- This doesn't seem to need belaboring in our community.
- If anything, we need to remind people to limit the use of exceptions to code that can't handle a failure locally.
- If your function throws and exception and catches it *in the same function*, you may be doing it wrong. . .

const (almost) all the things

- In session 2, Dan Saks told us that `const`qualifying the arguments of a function *definition* (not declaration) was pointless.
- Kate and James tell us that doing this is useful, because it helps show the intention of the implementer of the function.

Q: With whom do you agree?

What about `const` data members?

- A `const` *object* can't be modified — but that is just that *object*, not the type.
- A type with `const` data members is *immutable* — no object of that type can be modified.
- But having a `const` data member suppresses compiler generation of assignment, and can make *move* inefficient.

Q: Have you used `const` data members successfully?

Get rid of C-style casts

- Just don't do that. It is evil.

Q: What are good ways to identify C-style casts, to help in removing them?

Transform loops

- Sean Parent's talk "C++ Seasoning" is all about this.
- Nested loops in code is one of the most prevalent cause of complexity in code, making code hard to understand.
- "typical 700 line loop" — loops that are this long are too difficult to understand, and nearly impossible to test. Are you really sure you have tested *all* the branches in such a function?
- Lambdas make a world of difference in the use of algorithms to replace loops.

Q: When is a for loop better than use of `std::for_each`?

Q: What breakthrough moments have you had with other algorithms?

Q: Have you ever used `std::rotate`

Sean Parent seems to be able to do almost everything with `std::rotate`.

Section 2

Discussion time!