Reflection, Symbols

Axel Naumann

Reflection - Issues

Context:

- Object serialization, signal/slot, interpreter need reflection
- Arbitrary, user defined classes; can evolve over time: need abstraction (framework), provided by ROOT
- From headers extract types, member layout, inheritance… to produce reflection data and stubs (conversion, new/delete, function call normalization)

Cost:

- @runtime: need to connect objects with reflection
- @types: adding reflection info to user types allows faster reflection access given an object
- @build time: need to generate reflection data
- @memory: many strings (type names, member names)

Reflection - Ideas

- Compilers have types, members – want access! Less reflection data
- Extended type_info:
  - one object per process, not one per shared library: use as key
  - allow user-attached data: easier reflection access from object
  - object creation, destruction: less reflection functions
  - conversion of void* given its type_info and a target type_info: less reflection functions
  - is_base_of(other), offset(other, obj): less reflection data
- Canonical demangling / format of type_info::name(): less reflection data
- API for call normalization: call any function given object + vector of arguments, less reflection functions.

Symbols - Issues

- Too many shared libraries, too many symbols: interpreter needs all possibly called template functions: begin(), end(), size(), … Actually used: O(0.001)!
- Not dynamic, must use plugins (via interpreter) to reduce link-time dependencies
- Versioning problems: do libraries “match”?
- No canonical type names cause ambiguities in type databases
- vtable for interpreted classes deriving from compiled

Symbols - Ideas

- Access to standard, canonical name, from type_info or pure string manipulation
- Means to strip template defaults
- Symbol lookup from canonical name, e.g. for plugins