Fermilab

Introducing Iterative Algorithm Development

Sessions 16, 17, 18

Coverage

- 16 Introducing iterative algorithm development.
  - Introduce the ideas behind iterative development of algorithms, and why it is interesting.
  - 30 minutes (20 talk, 10 questions)
- 17 Completing an algorithm and improving it.
  - Complete a module that is partially written and restructure it into an algorithm that can be tested outside the framework. Write and run unit tests for this algorithm.
  - Explore different techniques for approaching algorithm improvements through exercises.
  - Gain some experience using C++14 standard algorithms
  - 130 minutes (130 work, sync between phases @30-45-45)
- 18 Summary – conclusion and wrap-up
  - 20 minutes (10 talk, 10 discussion)

The story

- Often the case for development efforts, we start with an initial body of code that has been put together rather quickly, and does only some of the things we need to have done.
- It is difficult to understand and modify this code – even if we’ve written it. This is especially true if bandages have already been placed upon it.
- The code is still missing key functionality. In this case, it is not yet creating the data product we need.
- We're going to refine the code in several steps before extending it to complete the desired functionality.

Purpose

- The purpose of this session is to do some iterative algorithm development, to obtain practice in
  - (1) refining existing code to improve clarity and maintainability,
  - (2) making code testable to reduce the chance for bugs -- now, and after future changes from others,
  - (3) extending code for greater generality and flexibility,
  - (4) using different techniques for implementing algorithms

Motivation – why bother with all this?

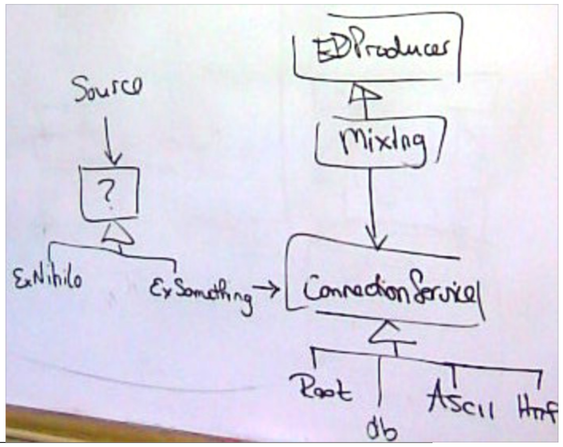
- C++ is now mature enough that with good names and good use of facilities, one should be able to read well-written code and know what it is doing.
- No one can remember all the code they have written, nor do they want to.
- There are better things to do than spend lots of hours trying to figure out what someone else’s code does.
- It is always nice to know if a change you made causes something else to fail. Many times untested changes that cause failures are not seen until a production release is built and used.
- If the code you add is well tested, blaming you for disasters will be very difficult. You can quickly move the finger from you to someone else. This puts you in a position of power.
- Less time debugging === more time for physics

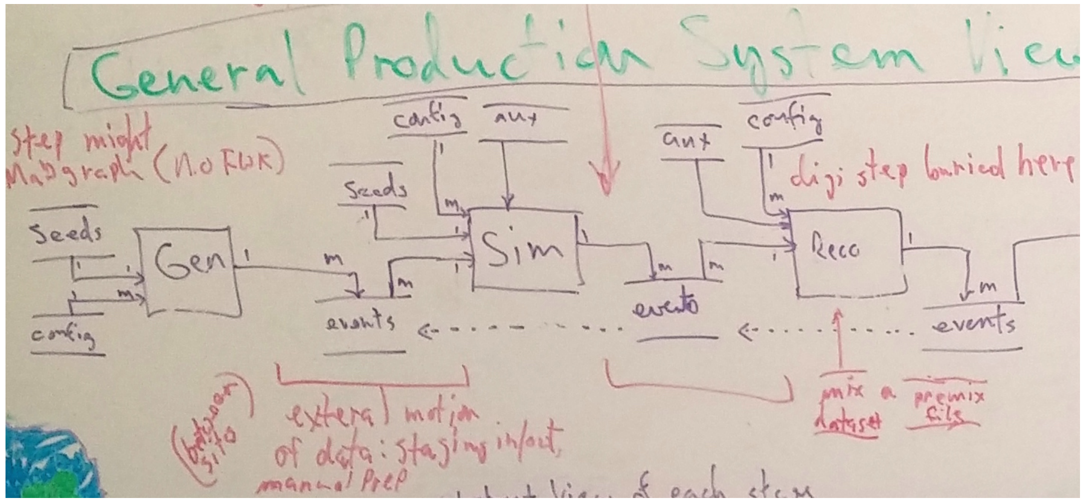
A strategy

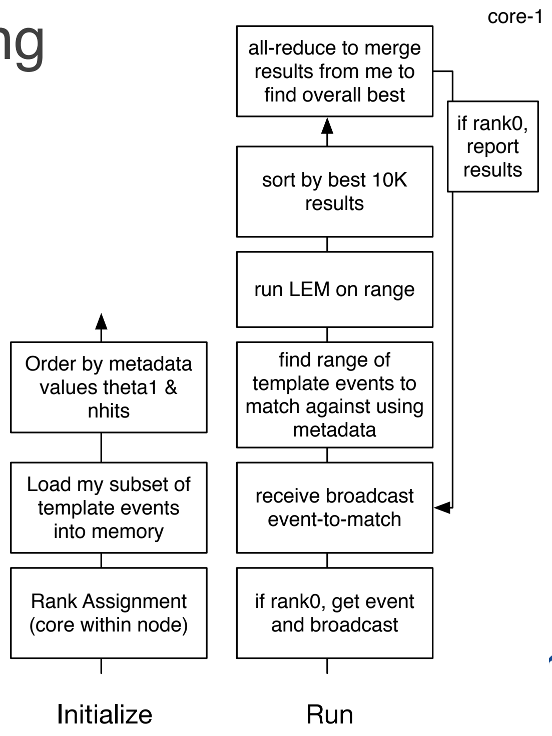
- The next set of slides list several guidelines that help improve algorithm and application code
- Applying any of these things is iterative – you should not expect to do everything all at once.
- Goal is to think about these guidelines as you work through the exercises. They have worked well for many of us.

1- A map of the code can be useful

- A block diagram on a whiteboard, a bullet list, or something similar
- Use names as you understand them and what the code is doing
- Capture only relevant features such as what functions are called and what are the major relationships in the data you are processing







core-1
all-reduce to merge results from me to find overall best
if rank0, report results
sort by best 10K results
run LEM on range
find range of template events to match against using metadata
receive broadcast event-to-match
if rank0, get event and broadcast
Order by metadata values theta1 & nhits
Load my subset of template events into memory
Rank Assignment (core within node)
Initialize
Run

2- Look for constants

- Symbolic names almost always help
- The experiment may already have a name for the constant that should be used
- The number may be used in more than one place (or been meant to be used in multiple places and not all have been edited)
- Make sure it is not really a configurable parameter (or needs to be cached)

3- Look for blocks that calculate something meaningful

- Sometimes these are prefaced by a comment explaining what the set of statements does.
- It is almost always better to have a well-named function replace the block
- The new function will read better and can be independently tested

4- Look for more than one level of nesting

- Think of **if** statements, **while** and **for** loops
- Working from the bottom up can be a useful way to tackle this one
  - inmost nesting body of code to outermost
- With **if/else** constructs, better to have positive statement in the **if** expression
- More than one **return** is okay in C++, along with **continue** if it is used well
- Use exceptions for failures requiring premature function exit - even within a loop
  - This does not imply using **try/catch** to handle a loop exit

```
if( !(flat.fire(1.0)<=fQE) ||
    !(fWavelen>fWavelenLow
&& fWavelen<fWavelenHigh) )
  ++fCountOpDetOther;
else {
  fThePhotonTreeDet->Fill();
  ++fCountOpDetDetected;
  }
```

```
void CandVertex::select( ClusterList &listU,
     ClusterList &listV,
     ClusterList &listW) {
  if (listU.uninteresting()) return;
  // rest of procesing
}
```

```
if((flat.fire(1.0)<=fQE) &&
   (fWavelen>fWavelenLow) &&
(fWavelen<fWavelenHigh) )
  { … } else { … }
```

5- Look for repeated blocks or lines of code

- That only differ in
  - starting or ending points
  - data being addressed or used
- Obvious candidates for new functions
- Don't forget about the function template and local functions here!
- Looking for code generating using copy-and-paste

6- Write a unit test that validates that the algorithm is working

- This will also test your knowledge of the class or function.
- If this is a difficult task, it might indicate that the functions is doing too much or requires to many facilities to be very useful.

7- Apply standard idioms and practices

- *Many of these covered on the first morning*
- RAII
- No bare pointers
- Prefer range-for to other for-loops
- Standard algorithms are also fun and easy to use now!
  - modern C++ makes this possible
- Arguments and return values
  - Pass big things by const-ref
  - return vectors by value (new with modern C++)

```
for(auto const& phot: theHit) {
  phot.process_me();
}
```

```
class Login {
public:
 Login(Database* db,
         std::string const& user):
  conn(db->connect(user))
 ~Login() { conn-
>disconnect(); }
private:
 DB::Connection* conn;
};
```

```
std::transform(x.begin(),x.end(), y.begin(),
   [&](double d) { return d + nd(eng); });
```

```
int* ip1 = new int(3); // bad
std::shared_ptr<int> ip2(new int(3)); // ok, but see below
std::unique_ptr<int> ip3(new int(3)); // good
auto ip5 = std::make_shared<int>(3); // preferred
```

8- Don’t hand-code things that the language will do for you automatically

- Do not write code for functions that the compiler will correctly generate for you
  - copy ctor, default ctor, destructor, etc.
- No manual memory management
  - should never see delete in the middle of a function
- Sorting, hash tables, set operations,
- random numbers <random>, time manipulations <chrono> , regular expressions <regex>

```
SymsVec out = find_syms();
sort(out.begin(),out.end());
```

```
// Engine likely comes from an art Service!
std::default_random_engine gen;
std::weibull_distribution<double> dist(1.2,300);
double n = dist(gen);
```

```
std::smatch m;
std::regex e ("(L_HitData_)([0-9]+)(.+)");
std::string hitname = find_hit_name(i);
myfiles_hit.push_back(hitname);
std::regex_search (hitname,m,e);
cout << "number = " << m[2] << "\n";
```

```
auto time_begin = std::chrono::system_clock::now();
double answer = calculate();
auto time_end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = time_end - time_begin;
std::time_t ending_time = std::chrono::system_clock::to_time_t(time_end);
```

9- The way you describe an algorithm or module to someone else might be the ideal way to express it in code.

- Do not need to have all the underlying functions in an algorithm written
  - Can just pretend they exist.
- Introduce a new class (datatype) if there is state to be maintained.
  - Think of int or double as a simple class that maintains one piece of data and has many operations defined on it.

Let’s get to work