Fermilab

Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

Iterative Algorithm Development Summary – conclusion and wrap-up

Sessions 18

What we’ve accomplished

- We’ve used a bunch of stuff from the strategy
  - Finding constants
  - Locating blocks of code to add functions and datatypes
  - Extracting algorithm code and testing it
  - Trying out different C++ facilities
- Demonstrated some utility in doing this
  - Evolved and explored changes to the algorithm outside of the framework
  - Changes the clarity of the module and algorithm

But how do I start from scratch?

- Remember the last point on the strategy slides
- Produce pseudo-code that describes the algorithm
- Write the code that invokes the fictitious function
- You will likely be able to move down a couple of layers into the function doing this.
- Eventually you will need to stop and implement pieces

Addressing bigger problems

- Starting from the top with the questions
  - "how does one obtain the results from this algorithm"
  - "what do I need to calculate the results"
- Always keep in mind the major general C++ design practices
  - Inheritance for interfaces, not implementation
    - public inheritance is not good for aggregating functionality
  - Datatypes (classes) should do one thing, not many
    - schizophrenia is not good
  - If there is no *state* to be maintained and changed, a function is certainly going to be better.
    - Data members listed in the class are *state*
  - Do not expose your guts (data members), unless the thing is a struct.

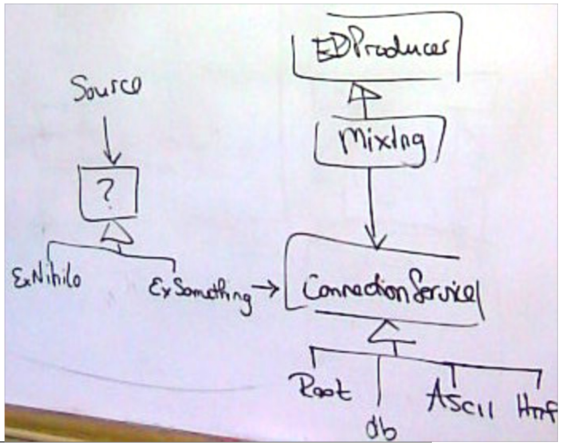
Generally good advice

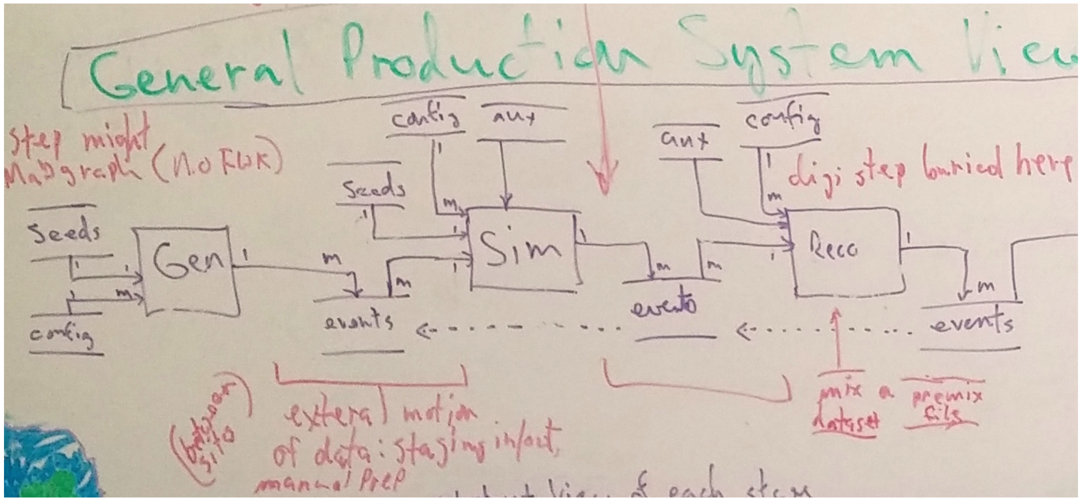
- Do not try to get it perfect the first time.
  - It is easier to complete something close and apply the techniques we used for the make combinations algorithm above
  - Doing several (four or more) quick versions or iterations is expected.
- Look for classes that already do what you want
  - 4-vector class is an example
- Invent the things you need
  - invent abstractions when you encounter the need
  - Not only datatypes or classes, but functions as well

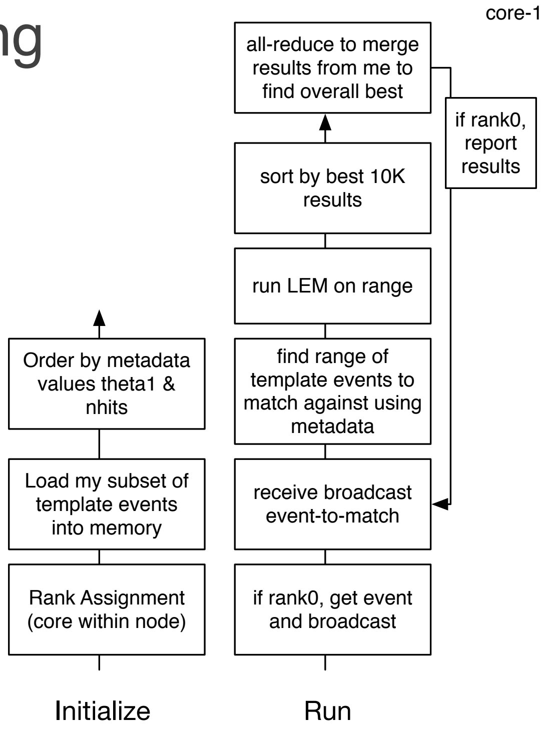
Backup slides – the strategy

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

Order by metadata values theta1 & nhits

find range of template events to match against using metadata

Load my subset of template events into memory

receive broadcast event-to-match

Rank Assignment (core within node)

if rank0, get event and broadcast

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!

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());
```

```
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";
```

```
std::chrono::time_point<std::chrono::system_clock> begin = std::chrono::system_clock::now();
 double answer = calculate();
 std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now();
 std::chrono::duration<double> elapsed_seconds = end-begin;
 std::time_t ending_time = std::chrono::system_clock::to_time_t(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.