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

Session 3
Basics of Data Structures

Chris Jones

art/LArSoft Course

03 August 2015

What We Will Cover

- C++ Standard Containers
- How to Choose Which Container to Use
- How to Use Containers Efficiently
- Gotcha

C++ Standard Containers

- `std::array<>`
- `std::vector<>`
- `std::unordered_map<>`
- `std::map<>`

std::array<>

- A fixed size container which holds a specific type

```
template< typename T, std::size_t N> struct array;
```

- Most memory efficient container
  - Basically just a wrapper around a C style array

```
std::array<int, 3> values = {1,2,3};

for( int v: values) {
   std::cout << v << std::endl;
}
```

std::array<> vs C Style Arrays

- std::array<> can be copied

```
std::array<int, 3> values = {1,2,3};
auto values2 = values;
```

- std::array<> has standard container interface

```
std::array<int, 3> avals = …; int cvals[3] = …;

assert(avals.size() == sizeof(cvals)/sizeof(int));

std::sort(avals.begin(),avals.end());
std::sort(cvals, cvals+sizeof(cvals)/sizeof(int));
```

std::vector<>

- A variable sized container which holds a specific type

```
template< typename T, …> class vector;
```

- The container you should use the most often
  - Can grow as needed
  - Excellent balance between memory use and speed

```
std::vector<int> values = {1,2,3};
values.emplace_back(4);

for( int v: values) {
   std::cout << v << std::endl;
}
```

std::unordered_map<>

- A container which associates a *value* to a *key*

```
template< typename Key, typename T,…> class
unordered_map;
```

- Used to quickly find a value when given a key
  - values are in an arbitrary order

```
std::unordered_map<int,std::string> valueToName =
  {{1,"one"},{2,"two"}, {3,"three"} };
valueToName[4] = "four";

for( auto const& v: valueToName) {
   std::cout << v.first <<" "<< v.second << std::endl;
}
```

std::map<>

- An ordered container which associates a *value* to a *key*

```
template< typename Key, typename T,…> class map;
```

- Used to quickly find a value when given a key
  - values are kept in order

```
std::map<int,std::string> valueToName =
  {{1,"one"},{2,"two"}, {3,"three"} };
valueToName[4] = "four";

for( auto const& v: valueToName) {
   std::cout << v.first <<" "<< v.second << std::endl;
}
```

Decision Tree for Picking a Container

- Are you adding to or removing from the container frequently?
  - Do you need to keep the items in the container ordered?
    - **`std::map<>`**
  - Do you need to quickly lookup items?
    - **`std::unordered_map<>`**
  - Default choice
    - **`std::vector<>`**
- Once filled, is the container unchanging?
  - Is the number of elements to store known at compile time?
    - **`std::array<>`**
  - Default choice
    - **`std::vector<>`**

Efficient: Pass Containers As Const References

- Passing containers by value requires copying the container
  - Very time and memory expensive
- Much better to pass as const reference

```
void slowArguments(std::vector<Foo> foos);

void fastArguments(std::vector<Foo> const& foos);
```

Efficient: Return Containers by Value

- C++ compiler can avoid copying container returned by a function
  - "Return Value Optimization"

```
std::vector<Foo> makeFoos(…);

void myfunction(…) {
…
    std::vector<Foo> foos;
    //compiler will transfer ownership from function
    // to local variable
    foos = makeFoos(…);
}
```

Efficient: Looping

- Use new “ranged for” or iterators when looping
  - only use integral index `for` loop if need index in the algorithm

```
std::vector<Foo> foos = …;
for(auto const& foo : foos ) { check(foo); }

for(auto it = foos.begin(), itEnd = foos.end();
    it != itEnd; ++it) {
   check_remaining(it, itEnd);
}

for(size_t i=0; i< foos.size(); ++i) {
   save_index(i, foos[i]); //less efficient
}
```

Gotcha: Modifying Container While Iterating

- Iterators can become invalid while you are modifying a container

```
std::vector<int> values = {…};

for(auto it = values.begin();
    it != values.end(); ++it) {
  if( *it % 2 == 0 ) {
     values.erase(it);
     //BAD, it is now invalid
  }
}
```