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

Session 2
Basics of Resource Management

Chris Jones

art/LArSoft Course

03 August 2015

What we will cover

- Stack, Heap and Global
- Smart Pointers
- RAII
- static Variables

Types of Memory

- Global
- Stack
- Heap

Global

- Memory for global objects is reserved at compile time
- Memory assigned when shared library is loaded into process

```
namespace bounds {
    constexpr int kMaxSize = 4;
    const std::vector<int> kDefaultSizes =…;
}
```

- Avoid use of non-const globals
  - Cause hidden dependencies which make debugging difficult
  - Cause thread safety problems

Stack

- Memory for local variables in a function
- Memory assigned when function is called
  - Memory is released when returned from function

```
void foo() {  int a = 4;  … }
```

- Never return a reference or pointer to a local variable
  - The memory for the variable may be used for another variable

```
const int& bar() {
  int a = 4;
  return a; //BAD
}
```

Heap

- Memory assigned during call to `new`
- Memory released when call `delete`

```
int* ptr =  new int{4};
…;
delete ptr;
```

- Safe to use heap memory for function return values
- Do not use raw pointers to manage memory
  - Use smart pointers

When to Use Each Type of Memory

- Global
  - constants known at compile time or at library load time
- Heap
  - use only when memory must persist after a function call
    - might even be avoidable if return by value and allow copy or move
- Stack
  - preferred memory for all other cases

Avoid Raw Pointers

- Managing memory by raw pointers is error prone
  - Failing to delete will cause program to crash when it runs out of memory

```
Foo * makeFoo() {
    return new Foo{…};
}

int bar() {
    auto f = makeFoo();
    return f->value();
    //Memory never released!
}
```

Smart Pointers

- Use smart pointers to safely handle heap memory

```
std::unique_ptr<Foo> makeFoo() {
    return std::make_unique<Foo>(…);
}

int bar() {
    auto f = makeFoo();
    return f->value();
    //Memory automatically released
}
```

Smart Pointer Types

- std::unique_ptr<>
- std::shared_ptr<>

std::unique_ptr<>

- Represents a single owner for the memory
  - use std::make_unique<> to create the object
- Can transfer ownership via **move**

```
auto f = std::make_unique<Foo>(…);
f->setValue(…);

//hand ownership to a function
save( std::move(f) );
//after std::move f will be null
```

std::shared_ptr<>

- Allows memory to be shared by multiple items
  - use std::make_shared<> to create the object
- Sharing is accomplished via copy

```
auto f = std::make_shared<Foo>(…);
f->setValue(…);

//share ownerships with a function
save( f );

//still safe to use
f->setValue(…);
```

RAII

- Smart pointers are an example of RAII

- RAII
  - Resource Acquisition Is Initialization
  - other half: Resource Release Is Destruction

- Meaning
  - Use objects to control lifetime of a resource
    - take control of resource in constructor
    - release resource in destructor

RAII Example

- Safely use C file handling API

```
class CFile {
   FILE * m_cfile;
public:
   CFile( const char* iName):
    m_cfile{fopen(iName, “w”)} {}

   ~CFile() { fclose(m_cfile); }

   void write(int i) {…}
};
```

RAII Example Continued

```
void writeValues(std::vector<int> const& values) {
   //file is opened
   CFile f{"test.log"};
   for( int v : values) {
     if (v > 10) {
        //file automatically closed
        throw ValueTooBigException();
     }
     f.write(v);
   }
   //file automatically closed
}
```

`static` Variables

- Using the keyword static with a variable defines a ‘global’
  - All users of that variable name address the same memory

- Only uses static to define const values

File Scope `static`

- Defines a 'global' that can only be accessed by code in the source file

```
static const unsigned int kMaxWidgets = 10;
```

- Better to use anonymous namespace
  - Only the source file can see items in the anonymous namespace

```
namespace {
    const unsigned int kMaxWidgets = 10;

    class Foo {…};
}
```

Class `static`

- Class static defines a global within a class' scope

```
class WidgetFactory {
    …
    static const unsigned int kMaxWidgets = 10;
};
```

Function Local `static`

- Defines a global variable that can only be used by that function

```
void foo(…) {
    …
    static const BigHelper helper;
    …
}
```