JETSCAPE Framework: Installation, Compilation And Testing

Prerequisites

- Compilation and building
  - C++ Compiler with C++11 support
  - CMake version 3 and above
  - zlib (should be already installed)
- Recommended Installation of the Prerequisites:
  - *Windows:* Windows is currently not supported. We recommend you install a virtual machine
  - https://www.storagecraft.com/blog/the-dead-simple-guide-to-installing-a-linux-virtual-machine-on-windows

  and a recent version of Scientific Linux. Then follow the instructions for Linux.

- Apple Mac:
  - Install a recent version of Xcode from the App Store; it should provide clang version 3.3 or above.
  - Install the command line tools: Open a Terminal and enter (without the "%")
    - `% xcode-select --install`
  - Install Homebrew.
  - Install cmake: Open a Terminal and enter (without the "%")
    - `% brew install cmake`
  - If necessary, install zlib: Open a Terminal and enter (without the "%")
    - `% brew install zlib`

Prerequisites - Linux

NOTE: It is assumed that you have administrator permissions or can obtain them using sudo. If that is not the case, please contact your System Adminstrator.

- Ensure gcc version 7 or above: In the command line, enter (without the "%")
  - `% gcc --version`
- If the version is too low, please update to the required version
- If necessary, install cmake using your distribution's package manager.

  In the command line, enter (without the "%")
  - `% sudo yum install cmake`
- If the version is too low, please update to the required version
- If necessary, install zlib: In the command line, enter (without the "%")
  - `% sudo yum install zlib`

Prerequisites - Optional

- Another session of the workshop will focus on an "analysis" of JETSCAPE output. For histogramming and visualization, we recommend ROOT v. 6, an existing version of which will be picked up during installation. It can be installed with the same packaged managers in the same manner as the required dependencies, i.e.

  ```
  % brew install root
  ```

  or

  ```
  % sudo yum install root
  ```

- **Note:** Please make us (mekhalaj@wayne.edu) aware of any other packages you would like to use instead.

JETSCAPE – Installation and Compilation

- Clone the repository from Github:
  - git clone https://github.com/JETSCAPE/JETSCAPE.git
- In the command line, navigate into the JETSCAPE directory and enter (without %):
  - `% cd JETSCAPE-`
  - `% mkdir build`
  - `% cd build`
  - `% cmake ..`
  - `% make`
- Test with the following commands which should produce an output file named test_out.dat:
  - `% ./brickTest`
  - `% ./readerTest`
- We encourage all attendees to familiarize themselves with the code and experiment with it!
- **NOTE**: you can find details installation instruction on JETSAPE GitHub page

JETSCAPE - Testing

- Inside build directory, you can run different tests
  - `./brickTest`
  - `./readerTest`
  - `./readerTestWithRoot`

Quick Introduction to C++11

Useful New Features

- Automatic Type Deduction
- Null Pointer
- Smart Pointer
- Range-based `for` Loops

Automatic Type Deduction : auto

- auto is a placeholder for a type
- Enables type inference

```
auto i = 42; // i is an int
auto l = 42LL; // l is an long long
auto p = new foo(); // p is a foo*
```

- can be used when declaring variables in different scopes
  - namespaces,
  - Blocks, or
  - initialization statement of for loops
- Using auto usually means less code

Null Pointer : `nullptr`

- Zero (0) used to be the value of null pointers
  - Problem: the implicit conversion to integral types

- The keyword `nullptr` denotes a value of type `std::nulptr_t`
  - Represents the null pointer literal

- No more implicit conversion to integral types.

Smart Pointers

- A wrapper class over a pointer with operators like * and -> overloaded

- The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction
  - we don’t have to explicitly use delete

- Destructor is automatically called when an object goes out of scope
  - The dynamically allocated memory is automatically deleted

Types of Smart Pointers

- `unique_ptr`
  - Should be used when ownership of a memory resource does not have to be shared
  - It doesn't have a copy constructor, but it can be transferred to another `unique_ptr`
- `shared_ptr`
  - Should be used when ownership of a memory resource should be shared
- `weak_ptr`
  - Holds a reference to an object managed by a `shared_ptr`, but does not contribute to the reference count;

Allocating Memory

- `make_shared<T>`
  - a non-member function for allocating memory for the shared object and the smart pointer

- Creating a smart pointer of type Foo

```
auto foo = std::make_shared<Foo>();
```

Example: (brickTest.cc)

```
// Main framework task
    int Nevents = 10;
    auto jetscape = make_shared<JetScape>("./jetscape_init.xml",Nevents);
    jetscape->SetId("primary");

    // Empty initial state
    auto ini = make_shared<InitialState>();
    ini->SetId("InitialState");

    // mono-energetic particle gun, parameters in XML file
    // 25% probability each for g, u, d, s
    auto pGun= make_shared<PGun> ();

    // Simple brick, parameters in XML file
    auto hydro = make_shared<Brick> ();

    // Energy loss manager, parameters in XML file
    auto jlossmanager = make_shared<JetEnergyLossManager> ();

    …

    //Remark: For now modules have to be added in proper "workflow" order
    jetscape->Add(ini);
    jetscape->Add(pGun);
    jetscape->Add(hydro);

    …

    // Intialize all modules tasks
    jetscape->Init();

    // Run JetScape with all task/modules as specified ...
    jetscape->Exec();

    jetscape->Finish();
```

Ranged-Based `for` Loops

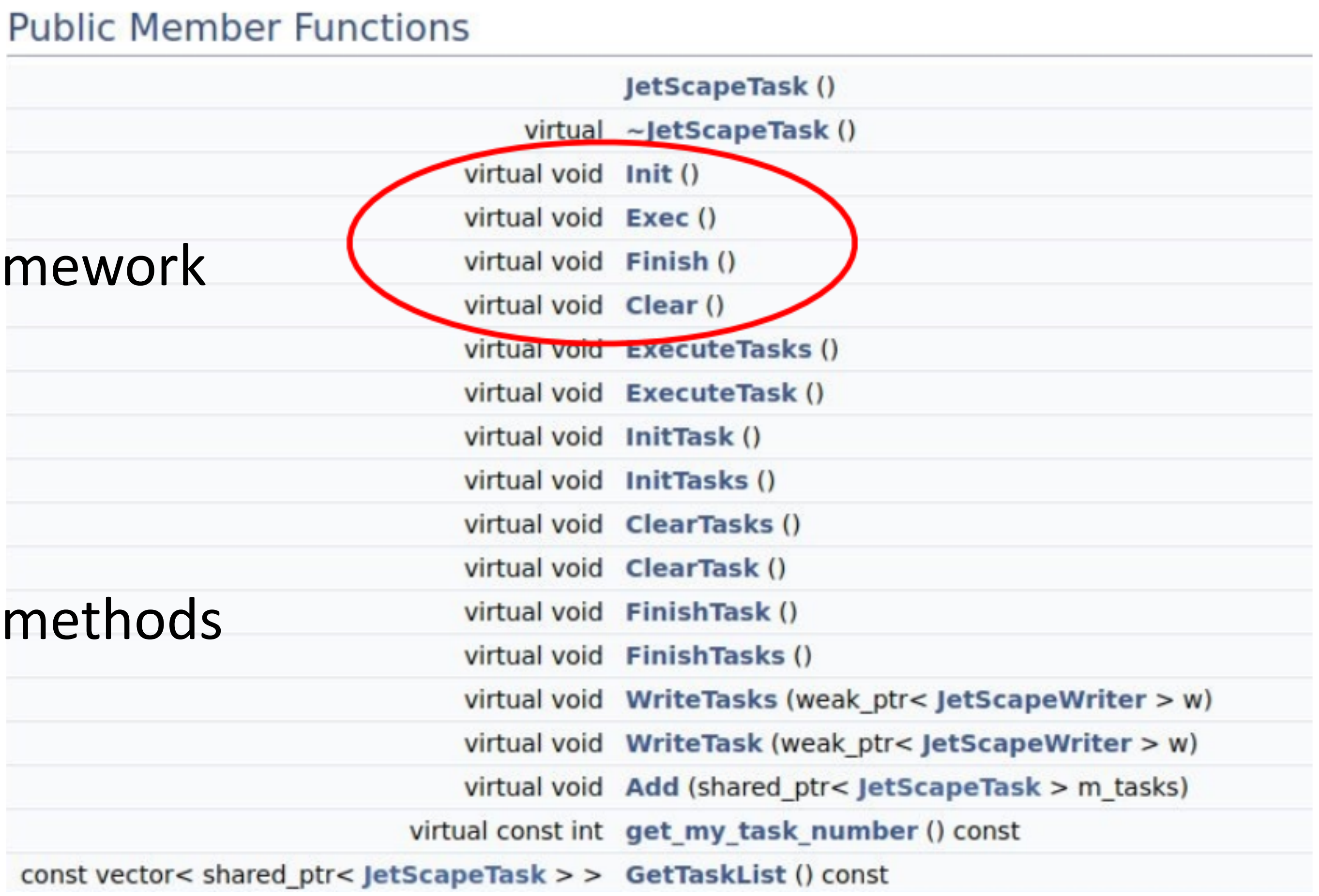
- Like the `foreach` paradigm of iterating over collections
- useful when you just want to get and do something with the elements of a collection/array
    - you don't care about indexes, iterators or number of elements

```
std::vector<int> v = {1,2,3,4,5};
for(auto i : v){
    std::cout << i << ‘ ‘;
}
```

Task-Based Framework And Adding a Hello World Task/ Module

Task-Based Framework

- JETSCAPE Framework provides base classes
  - JetScapeTask
    - All the tasks in the framework are subclasses of this class
  - Methods called recursively by framework
    - Init()
    - Exec()
    - Finish()
    - Clear()
  - Each task must implement these methods



Public Member Functions

	JetScapeTask ()
virtual	~JetScapeTask ()
virtual void	Init ()
virtual void	Exec ()
virtual void	Finish ()
virtual void	Clear ()
virtual void	ExecuteTasks ()
virtual void	ExecuteTask ()
virtual void	InitTask ()
virtual void	InitTasks ()
virtual void	ClearTasks ()
virtual void	ClearTask ()
virtual void	FinishTask ()
virtual void	FinishTasks ()
virtual void	WriteTasks (weak_ptr< JetScapeWriter > w)
virtual void	WriteTask (weak_ptr< JetScapeWriter > w)
virtual void	Add (shared_ptr< JetScapeTask > m_tasks)
virtual const int	get_my_task_number () const
const vector< shared_ptr< JetScapeTask > >	GetTaskList () const

Add Your Module: Hello World!

- Add the header file
- Overrides JetScapeTask
  - Methods are virtual
  - Will be implemented in source file

To add you own task or module:
- Create a header file *.h
- Create a source file *.cc
You can add these test files in the framework folder. If your code ends with .cc then cmake will automatically find and compile!

```
#ifndef HELLOWORLDMODULETEST_H
#define HELLOWORLDMODULETEST_H
#include "JetEnergyLossModule.h"
using namespace Jetscape;
class HelloWorld : public JetScapeTask
{
public:
HelloWorld();
virtual ~HelloWorld();
virtual void
virtual void Init();
virtual void Exec();
virtual void Clear();
virtual void Finish();
};
#endif
```

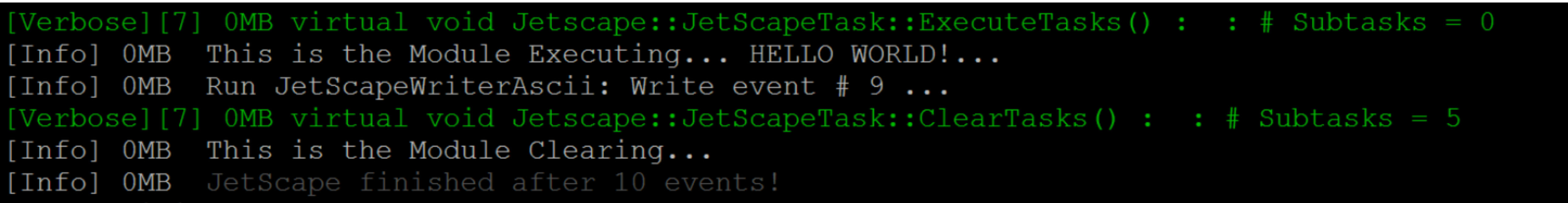
Add Your Module: Hello World!

- Add the source file
- Implements JetScapeTask methods

```
#include "HelloWorldModuleTest.h"
#include<iostream>

using namespace Jetscape;

HelloWorld::HelloWorld()
{
   SetId("HellowWorld");
   VERBOSE(8);
}

HelloWorld::~HelloWorld()
{
   VERBOSE(8);
}

void HelloWorld::Init()
{
   INFO<<"Initialize HelloWorld Module ...";
}

void HelloWorld::Exec()
{
    INFO<<"This is the Module Executing... HELLO WORLD!...";
}

void HelloWorld::Clear()
{
    INFO<<"This is the Module Clearing...";
}

void HelloWorld::Finish()
{
    INFO<<"This is the Module Finishing...";
}
```

Add Your Hello World Module in Brick test

- Include the header file
  - `#include "HellowWorldModuleTest.h“`
- Create a pointer
  - `auto helloWorld = make_shared<HelloWorld> ();`
- Attach the pointer to the main task
  - `jetscape->Add(helloWorld);`
- Framework will run your module



```
[Verbose][7] 0MB virtual void Jetscape::JetScapeTask::ExecuteTasks() :   : # Subtasks = 0
[Info] 0MB  This is the Module Executing... HELLO WORLD!...
[Info] 0MB  Run JetScapeWriterAscii: Write event # 9 ...
[Verbose][7] 0MB virtual void Jetscape::JetScapeTask::ClearTasks() :   : # Subtasks = 5
[Info] 0MB  This is the Module Clearing...
[Info] 0MB  JetScape finished after 10 events!
```