Course Title: Tour of C++11 & 14

Faculty: Rajesh Sola

L&T Technology Services

LTTS
GLOBAL
ENGINEERING
ACADEMY

Agenda

- Basics
- Classes & Objects
- Move Semantics
- Lambdas & Callbacks
- Templates & Exception Handling
- STL Improvements
- Smart Pointer

Introduction

Agenda

- What's new in C++11 & C++14
- Language Improvements
- Move Semantics
- Lambdas & Callable elements
- STL Improvements
- Smart Pointers
- Concurrency & IPC
- Some core insights

Standards & Compiler Support

- ISO C++ standards
  - C++98, C++03, C++11, C++14
  - C++17 , C++20
- GNU Dialects
  - gnu++98, gnu++03,gnu++11, gnu++14, gnu++17
- Intermediate standards
  - C++0x, C++1x, C++1y, C++1z, C++2a
  - gnu++0x, gnu++1x, gnu++1y, gnu++1z
- Compiler support
  - http://en.cppreference.com/w/cpp/compiler_support
  - https://gcc.gnu.org/projects/cxx-status.html
  - https://clang.llvm.org/cxx_status.html

Language Basics - Improvements

- General Features
  - constexpr
  - auto type
  - decltype
  - range based for loops
  - static_assert
  - Nullptr
  - Scoped/Strongly typed enums
  - strict initializers with {}
  - using keyword for aliasing
  - user defined literals
  - binary literals
  - digit separators
  - User defined Literals
  - Raw string literals

Classes & Objects- Improvements

- Classes & Objects
  - Default controls
  - Constructor delegating
  - In class initializers
  - Uniform initializers
  - Initializer list
  - Explicit conversion operators
  - Read only objects
  - Explicit Type Conversions
  - Type Traits
- Inheritance
  - final, override keywords
  - explicit inheritance of base class members

Object Model & Move Semantics

- Types
  - Trivial
  - POD
  - Standard layout
- Move operations
  - Move constructor
  - Move operator=
- R-value references, compatibility, casting (std::move)
- Rule of three/five/zero
- Universal references, Perfect forwarding (std::forward)

Lambdas & Callbacks

- Lambda expressions, usage
- Capture Syntax
- Generic Lambdas
- std::function
- std::bind

Templates, Exception Handling

- Templates
  - Right angled parenthesis
  - Aliasing Templates
  - Extern Templates
  - Variadic Templates
- Exception handling
  - noexcept keyword

STL Improvements

- New Containers
  - std::array, std::forward_list, unordered maps & sets
- New Operations
  - emplace_back, shrink_to_fit, data
- New Algorithms
  - std::for_each
- Tuples
- Regular Expressions

Smart Pointers

- Challenges with Raw pointers
- Memory leaks, Heap Issues
- std::unique_ptr
- std::shared_ptr
- std::make_unique
- std::make_shared
- std::weak_ptr
- Cyclic Redundancy problem & solution
- dynamic_pointer_cast, static_pointer_cast

Templates, Exception Handling

Outline

- Templates
  - Right angled parenthesis
  - Aliasing Templates
  - Extern Templates
  - Variadic Templates
- Exception handling
  - noexcept keyword

Right Angled Parenthesis & Aliasing

```
std::vector<std::complex<double>   > v1;
//space required prior to C++11

std::vector<std::complex<double>> v1;
//no space required from C++11
```

```
using ivector=std::vector<int>;
template<typename T1, typename T2,typename T3=float>
class Sample { };
template<typename T>
using isample=Sample<T,int,int>;      //typedef won't work here
template<typename T1, typename T2>
using dsample=Sample<T1, T2, double>; //typedef won't work here
using IPoint=Point<int>
using IPointVector=std::vector<Point<int>>; //std::vector<IPoint>
```

Variadic Templates

```
void myprint(int n) {  //required on reaching end of argument list
std::cout << "processed all arguments\n";
}
template<typename T,typename... Args>
void myprint(int n,T val,Args... args) {
  std::cout << val << "\n";
  myprint(n-1,args...);
}
int main() {
  myprint(3,10,2.3f,"hello");
  myprint(5,"hello",2.3f,15,46,54);
}
```

- variadic templates can take variable no.of arguments
- A better replacement for va_list ,va_arg,va_start,va_end from stdarg.h
- Use cases:- emplace operations, make_xxxx wrappers
- std::forward usage with variadic args

Need for Extern Templates

```
template<typename T>
void myswap(T& r1,T& r2) {
  T temp = r1;
  r1 = r2;
  r2 = temp;
}
```

myswap.h

- myswap definition is instantiated in each source file
- myswap is a weak symbol in each translation unit
- check symbol state of myswap using nm/objdump

```
#include"fun.h"
void f1() {
  int a=10,b=20;
  myswap<int>(a,b);
}
```

f1.c

```
#include"fun.h"
void f2() {
  int a=10,b=20;
  myswap<int>(a,b);
}
```

f2.c

Extern Templates

```
template<typename T>
void myswap(T& r1,T& r2) {
  T temp = r1;
  r1 = r2;
  r2 = temp;
}
extern template void myswap(int&,int&);
```

- Now myswap is a undefinedsymbol in each translation
- Ensure that myswap Is defined exactly in one place, typically in main file

```
template void myswap(int&,int&);
```

```
#include"fun.h"
void f1() {
  int a=10,b=20;
  myswap<int>(a,b);
}
```

```
#include"fun.h"
void f2() {
  int a=10,b=20;
  myswap<int>(a,b);
}
```

Templates in STL

```
//In multiple files if std::vector<int> is used
extern template std::vector<int>;

//In one translation unit say main.cpp
template std::vector<int>
```

```
//In multiple files if std::list<Point> is used
extern template std::list<Point>;

//In one translation unit say main.cpp
template std::list<std::Point>
```

STL Additions

Outline

- New Containers
  - std::array, std::forward_list, unordered maps & sets
- Added Operations
  - emplace_back, shrink_to_fit, data
- New Algorithms
  - std::for_each
- Tuples
- Regular Expressions

New Containers

- std::array
- std::unordered_map
- std::unordered_set
- std::unordered_multimap
- std::unordered_multiset
- std::forward_list

- std::vector vs std::array
- Dynamic vs Fixed size, Non-Type Template parameter for creation of array
- Unordered containers are based on hashing techniques (hash tables) whereas ordered containers (std::map, std::set etc.) are tree based

Additional Container Operations

- std::vector, std::list ==> push_back(T&&)
- std::vector ==> data, shrink_to_fit
- most containers ==> T&&, std::initializer_list with insert
- most containers ==> emplace, emplace_back
- std::list ==> emplace_front
- std::map, std::set ==> emplace_hint, try_emplace
- std::map ==> at

New Algorithms

- std::for_each
- std::copy_if, std::copy_n
- std::std::move (algo)
- all_of, any_of, none_of
- minmax, minmax_element
- std::iota, std::shuffle
- std::is_sorted, std::is_sorted_until
- std::is_heap, std::is_heap_until
- std::is_partitioned
- std::partition_copy, std::parition_point
- std::minmax, std::minmax_element

Tuples

```
std::tuple<int,int,std::string> t1(10,20,"abcd");
int val1=std::get<0>(t1);
int val2=std::get<1>(t2);
std::string val3=std::get<2>(t1);
auto val=std::get<4>(t1); //error
val=std::get<x>(t1);      //error
std::tie(val1,val2,val3)=t1;
std:: tie(val1,ignore,val3)=t1;
std::tuple<int,int,int> t2 =
      std::make_tuple(10,20,30);
auto t3 = make_tuple(10,5.6,"abcd");
```

Tuples

```
std::tuple<int,std::string> t4(10,"abc");
std::tuple<double,bool> t5(2.3,true);
auto t6=std::tuple_cat(t4,t5);
//std::tuple_size<decltype(t4)>::value
//std::tuple_size<decltype(t5)>::value
//std::tuple_size<decltype(t6)>::value
std::tuple_element<0,decltype(t6)>::type first = std::get<0>(t6);
using type3 = std::tuple_element<3,decltype(t6)>::type;
std::tuple<int,char,int,char> somefun() {
  //some code
  return std::make_tuple(10, 'A', 20, 'B');
}
```

std::regex usage

- Classes
  - std::basic_regex
  - std::sub_match
  - std::match_results
- Algorithms
  - std::regex_match
  - std::regex_search
  - std::regex_replace
- std::regex_error
- std::regex_iterator
- std::regex_token_iterator

std::regex_match

```
std::string msg = "GNU Linux";
std::regex p1("(GNU)(.*)");
if(std::regex_match(msg, p1))
std::cout << "Match OK";
std::regex p2("(.*)(Linux)");
if(std::regex_match(msg, p2))
std::cout << "Match OK";

std::string str = "My phone number is +91-98450-12345, yes;
std::regex pm("(.*)\\+([0-9]{2})\\-([0-9]{5})\\-([0-9]{5})(.*)");
std::cout << std::regex_match(str,pm) << "\n";
```

std::regex_search

```
std::string str = "My phone number is +91-98450-12345, yes;
std::regex p1("\\+([0-9]{2})\\-([0-9]{5})\\-([0-9]{5})");
std::regex pm("(.*)\\+([0-9]{2})\\-([0-9]{5})\\-([0-9]{5})(.*)");
std::cout << std::regex_search(str,p1) << "\n";
std::smatch sm; std::regex_search(str,sm,p1);
for(auto ch:sm)
  std::cout << ch << "\n";
```

std::regex_replace

```
std::string s1 = "This is a book and is heavy";
std::regex p1("is");
std::cout << std::regex_replace(s1, p1, "was");

std::string s2 = "Hello, How Are You\n";
std::regex p2("a|e|i|o|u");

std::string s3;
std::regex_replace(std::back_inserter(s3), s2.begin(), s2.end(), p2, "@");
std::regex_replace(std::ostreambuf_iterator<char>(std::cout),
s2.begin(), s2.end(), p2, "*");
```

Reference Wrapper

```
std::vector<int> v1{10,20,30,40,50};
std::vector<int&> v2(v1.begin(), v1.end()); //error, Not allowed
std::vector<std::reference_wrapper<int>> v2(v1.begin(), v1.end()); //ok
++v1.at(0);
std::cout << v2.at(0) << "\n";
int a=10,b=20,c=30,d=40,e=50;
std::reference_wrapper<int> arr[]={a, b, c, d, e};
++c;
std::cout << arr[2] << "\n";
arr[3]=45;
std::cout << d << "\n";
```

std::ref, std::cref

```
template<typename T>
void test(T& x,T& y) {
  ++x;
  ++y;
}
template<typename T>
void test(const T& x,const T& y) {
  //some code
}

int a=10,b=20;
test(std::ref(a), std::ref(b)); //matches test(T&,T&)
test(std::cref(a), std::cref(b)); //matches test(const T&,const T&)
```

std::ref, std::cref

```
void test(int& a,int& b) {
  ++a; ++b;
}

//auto fbound = std::bind(test, x , y );
auto fbound = std::bind(test, std::ref(x) , std::ref(y) );
x=10;
y=20;
std::cout << x << "," << y << "\n";
fbound();
std::cout << x << "," << y << "\n";
```

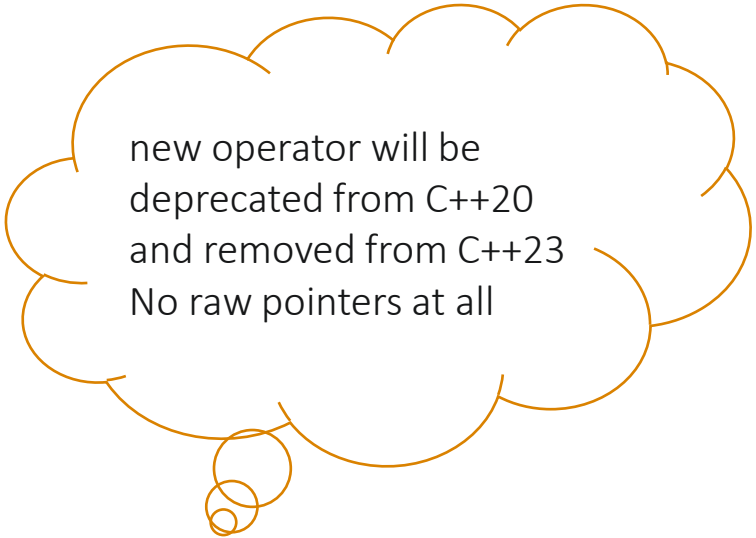
Smart Pointers

Outline

- Challenges with Raw pointers – Memory leaks, Heap Issues
- std::unique_ptr
- std::shared_ptr
- std::make_unique, std::make_shared
- std::weak_ptr
- Cyclic Redundancy problem & solution
- dynamic_pointer_cast, static_pointer_cast

Challenges with Raw pointers

- Memory Leaks
- Ownership – single/ multiple, transfer
- Lifetime control of heap objects
- Thread Safe Pointers
- Copy on Write



Memory Leaks

```
int *parr;
parr=new int[max_len];  //Simple Array
//delete[] parr         // Omission of delete
```

```
Sample *ptr=new Sample(x,y);
ptr->printxy();           //User defined objects
//delete ptr;             //Omission of delete
```

Omission of destructor in a non-trivial class like MyString??

Smart Pointers - Overview

- std::unique_ptr
- std::shared_ptr
- std::weak_ptr
- Wrappers for creation
  - std::make_unique
  - std::make_shared
- Type Casting
  - std::dynamic_pointer_cast
  - std::static_pointer_cast
  - std::const_pointer_cast
  - std::reinterpret_pointer_cast

std::unique_ptr

- std::unique_ptr creates a smart pointer with exclusive ownership to managed object
- Managed object will be released when unique pointer goes out of scope (with a call to delete
- on managed object as default deleter, custom deleter can alter this behavior)
- Managed object is accessible through operator*, operator->
- Copy semantics are restricted on unique pointer, justified for exclusive ownership
- When move semantics are applied ownership will be transferred from one object to other

std::unique_ptr

```
std::unique_ptr<Sample> p1(new Sample(x,y));
p1->printxy();
std:: unique_ptr<Sample> p2(p1); //error, deleted copy ctor
std:: unique_ptr<Sample> p2=p1; //error, deleted operator=
std:: unique_ptr<Sample> p2(std::move(p1)); //move constructor
std::unique_ptr<Sample> p3=std::move(p2); //move operator=
Sample &ref=*p3; //ok
Sample s2(*p3); //ok, s2 is clone of *p3
Sample s3(std::move(*p3)); //move ctor
```

- only p3 is valid & pointing to object now
- ownership transferred to p2 p1 won't point to object now

std::make_unique

```
std::unique_ptr<Sample> p1;
p1 = make_unique<Sample>(10,20);
std::unique_ptr<MyString>
s1=make_unique<MyString>("abcdxyz");
```

- Constructs the object with necessary constructor arguments
- returns the address in the form of unique_ptr

Polymorphic Pointers

```
std::unique_ptr<Account> acc1(new SavingsAccount(id,name,bal));
std::unique_ptr<Account> acc2;
acc2=std::unique_ptr<SavingsAccount>(new SavingsAccount(id,name,bal));
std::unique_ptr<Account> acc3;
acc3=std::make_unique<SavingsAccount>(id,name,bal);
```

```
std::unique_ptr<IMUSensor> reading1(new Accelerometer(x,y,z));
std::unique_ptr<IMUSensor> reading2;
reading2 = std::unique_ptr<Accelerometer>(new Accelerometer(x,y,z));
std::unique_ptr<IMUSensor>
reading3=std::make_unique<Accelerometer>(x,y,z);
```

Example – STL Containers

```
std::vector<Sample*> mylist;
mylist.push_back(new Sample(10, 20));
mylist.push_back(new Sample(11, 22));
//omission of delete on each object leads to memory leaks
```

```
std::vector<std::unique_ptr<Sample>> slist;
slist.push_back(std::unique_ptr<Sample>(new Sample(10, 20)));
slist.push_back(std::make_unique<Sample>(10,20));
//no need to release the vector elements explicitly
```

Custom Deleters

```
class MyDeleter {
  public:
  void operator() (Box* rawptr) {   //Single Object
    delete rawptr;
  }
};
std::unique<Box, MyDeleter> sp(new Box(10,12,5));

template<typename T>
class MyDeleter {
  public:
  void operator() (T* rawptr) {    //Array of objects
    delete[] rawptr;
  }
};
std::unique_ptr<Box, MyDeleter> sparr(new Box[10]);
```

Polymorphic Pointers

```
std::vector<std::unique_ptr<Account>> accounts;
accounts.push_back(std::unique_ptr<Account>(new
SavingsAccount(id,name,bal));
accounts.push_back(std::unique_ptr<SavingsAccount>(new
SavingsAccount(id,name,bal));
accounts.push_back(std::make_unique<SavingsAccount>(id,name,bal));
```

```
std::array<std::unique_ptr<IMUSensor>,maxsize> readings;
readings.push_back(std::unique_ptr<IMUSensor>(new Accelerometer(x,y,z));
readings.push_back(std::unique_ptr<Accelerometer>(new
Accelerometer(x,y,z));
readings.push_back(std::make_unique<Acceleroeter>(x,y,z));
```

std::shared_ptr

- Unlike unique pointers, shared pointer allows shared ownership to managed objects, i.e.access to the managed object using multiple shared pointers.
- Managed object will be destructed (delete the raw pointer) when all the shared pointers goes out of scope.
- operator[] is not available with shared pointer to manage array of objects (unlike unique_ptr), i.e. std::shared_ptr<T[]> is anot available, but std::shared_ptr<T> can point to array of objects
- These points are same as unique_ptr
  - get member provides access to raw pointer for the managed object
  - Managed object is accessible through operator*, operator->
  - operator++, operator-- are not available

std::shared_ptr

```
std::shared_ptr<MyString> ps1(new MyString("abcdxyz"));
cout<< ps1.use_count();
std::shared_ptr<MyString> ps2=ps1;
cout<< ps2.use_count();
test(ps1);

void test(std::shared_ptr<MyString> ps3) {
  cout<< ps3.use_count() << "\n";
}
```

- Shared ownership among multiple instances of shared_ptr
- Underlying object will be destroyed only when use count drops to zero

std::make_shared

```
std::shared_ptr<Sample> sp=std::make_shared<Sample>(10,20);
std::shared_ptr<Account>
sp=std::make_shared<SavingsAccount>(id,name,balance);
std::shared_ptr<Sensor> sp=std::make_shared<Accelerometer>(x,y,z);
```

Significance of std::make_unique, std::make_shared instead of calling constructors of std::unique_ptr, std::shared_ptr

Default and Custom Deleters

```
void mydeleter(Box *pbase) {
  delete[] base;
}
std::shared_ptr<Box> sp(new Box[len],mydeleter);
```

```
template<typename T> //Using Function Objects
class ArrayDeleter {
 public:
  void operator() (T* rawptr) {
    delete[] rawptr;
  }
};
std::shared_ptr<Box> p1(new Box[10], ArrayDeleter<Box>());
std::shared_ptr<int> p2(new int[len],ArrayDeleter<int>());
```

Default and Custom Deleters

```
std::shared_ptr<Sample> p1(new Sample[10],[](auto *pbase) {
  delete[] pbase;
});
std::shared_ptr<int> p1(new int[len],[](auto *pbase) {
  delete[] pbase;
});
//Lambdas as custom deleters
```

LinkedList Example

```
class Node {
  int m_value;
  std::shared_ptr<Node> m_next;
  public:
  Node(int val,std::shared_ptr<Node>ptr):m_value(val),m_next(ptr) { }
  friend class LinkedList;
};
class LinkedList {
  std::shared_ptr<Node> pstart=nullptr;
 public:
  void insbeg(int val) {
    std::shared_ptr<Node> ptemp=std::make_shared<Node>(val,nullptr);
    ptemp->m_next=pstart;
    pstart=ptemp;
  }
```

LinkedList Example

```
  void delbeg() {
    std::cout << "going to delete:" << pstart->m_value;
    pstart=pstart->m_next;
  }
  void display() {
    std::shared_ptr<Node> pcur=pstart;
    for(; pcur!=nullptr; pcur=pcur->m_next) {
      std::cout << pcur->m_value << "\n";
      std::cout << pcur.use_count() << "\n";
    }
   }
};
```

std::weak_ptr

- Weak pointers holds a non owning reference to an object managed by another shared pointer
- Weak pointers doesn't participate in reference counting
- Weak pointers need to be converted into shared pointer to access underlying objects.
- Weak pointer may live beyond the life time of managed object ,but is considered as expired and can't access underlying object

std::weak_ptr

```
void test_weak_ptr(std::weak_ptr<Sample>& wp) {
  std::cout << "test--wp.expired?: " << wp.expired() << std::endl;
  if( std::shared_ptr<Sample> stemp = wp.lock() ) {
    std::cout << "Test--weak pointer is alive\n";
    std::cout << "stemp.use_count(): " << stemp.use_count() << std::endl;
    stemp->printxy();
  }
  else {
    std::cout << "Test -- Weak pointer got expired\n";
    //stemp->printxy(); //run time exception
  }
}
```

std::weak_ptr

```
int main() {
  //std::shared_ptr<Sample> sp(new Sample(10,20));
  std::shared_ptr<Sample> sp=std::make_shared<Sample>(10,20);
  std::weak_ptr<Sample> wp=sp;
  std::cout << "weak ptr use_count: " << wp.use_count() << std::endl;
  std::cout << "shared ptr use_count: " << sp.use_count() << std::endl;
  test_weak_ptr(wp);
  sp.reset(); //managed object is released
  test_weak_ptr(wp);
  return 0;
}
```

std::weak_ptr

```
std::weak_ptr<Sample> wp;

{
  std::shared_ptr<Sample> sp=std::make_shared<Sample>(10,20);
  wp=sp;
  test_weak_ptr(wp);
}

//managed object is released as sp is out of scope
//But wp still exists in expired state, can't access object

test_weak_ptr(wp);
```

std::weak_ptr

```
std::weak_ptr<Sample> wp;
void bar(std::shared_ptr<Sample>& sptr) {
  std::shared_ptr<Sample> stemp(sptr);
  test_weak_ptr(wp); //3
}
void foo() {
  std::shared_ptr<Sample> sp=std::make_shared<Sample>(10,20);
  test_weak_ptr(wp); //2
  bar(sp);
  test_weak_ptr(wp); //4
}
int main() {
  test_weak_ptr(wp); //1
  foo();
  test_weak_ptr(wp); //5
}
```

Cyclic Redundancy Problem

- Usecase Scenario – Widget Example
- Solution with Shared Pointers – One Way
- Solution with Shared Pointers – Two Way
- Cyclic Redudancy with above approach
- Solution with Weak Pointers

Pointer Cast with Shared Ptr

- dynamic_pointer_cast
- static_pointer_cast
- const_pointer_cast
- reinterpreter_pointer_cast

Thank You !

L&T Technology Services