Course Title: Tour of C++11 & 14

Faculty: Rajesh Sola

L&T Technology Services

LTTS GLOBAL ENGINEERING ACADEMY

Agenda

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

Chrono Library

- std::chrono library
- Predefined literals
- System clock

Threads & Async Tasks

- std::thread
- Callbacks
- std::async
- std::future

Locking & IPC Techniques

- Atomic Variables (std::atomic)
- std::mutex
- std::unique_lock
- std::lock_guard
- std::condition_variable
- std::future
- std::promise

Language Basics

Outline

- 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

constexpr

- How do you define constants prior to C++11?
- constexpr variables are evaluated at compile time (truly), must be initialized with compile time known values
- constexpr functions are evaluated at compile time, if all arguments are compile time known.
- The return values can be assigned to constexpr variables•
- constexpr improves ROMability of the code
- constexpr variables are eligible as
  - size of global arrays, a better choice than const variable
  - non type templates parameters
- Relaxed constexpr constraints in C++14
- Read only objects can be modeled using constexpr (Will discuss this later)

constexpr - syntax & examples

```
constexpr int maxval=100;
constexpr double pi=22.0/7.0;
```

Global Variables

```
constexpr double pi=22.0/7.0
constexpr double area(int radius) {
  return pi*radius*radius;
}
```

Global Functions, compile time evaluation if all arguments are compile-time known (literals or constexpr variables)

auto keyword

- Gained a lot of popularity from C++11
- Type inference based on assignments / return values
- Generic parameters with auto keyword, an alternative to templates
- Syntactical changes in return type between C++11 & C++14
- Some other usage
  - Loop variables in range based for loops
  - Declaring iterators
  - Declaring parameters of lambda expressions
  - Declaring callback types

Auto keyword

```
auto max = 100;
auto res = foo(x, y);
auto iter = mylist.begin();

int sum(auto x, auto y) {
  return x + y;
}

auto sum(auto x, auto y) {
  return x + y;
}
```

- Declaring Variables
- Function Parameters
- Function return type (from C++14)
  - In C++11 trailing return syntax required

decltype

- Another great keyword for type inference
- Static type inference with decltype
- Optional initialization
- decltype for Template parameters
- Additional
  - decltype for trailing return syntax
  - Handling reference types with decltype (std::remove_reference)
  - delctype(auto) to preserve reference types

decltype – syntax and examples

```
int x=10;double y;
decltype(x+y) z;
auto res=testcall();
decltype(res) val;
decltype(ptr->x) temp;
decltype(ptr->getsum()) temp;
```

```
int arr[] = { 10,20,30,40,50 };
std::vector<decltype(arr[0])> v1;

//int sum(int,int);
decltype(sum(a,b)) c;
```

```
std::vector<float> v1;
decltype(v1.at(0)) sum;
std::list<std::remove_reference<decltype(v1.at(0))>::type> l1;
```

range based for loops

```
int arr[]={10,20,30,40,50};
for (auto &r : arr) {
  r=rand()%100;
}
for (auto x : arr) {
  sum+=x;
}

std::vector<int> v1;
//add some data,push_back
for (int &x : v1) {
  x=rand()%100;
}
for (int x : v1) {
  sum+=x;
}
```

- Applicable for iterator-based data sets (containers)
- Declare loop variables : auto vs specific types
- By value, By reference
- const keyword prefix with reference

Strict initializers

```
int x(10);       //ok
float y(2.3f);   //ok
int z(2.3f);     //conversion
int w{2.3f};     //error
int b{20};       //ok


Sample s1(10,20);
//Sample::Sample(int,int);
Sample s2(2.3,5.6};  //conversion
Sample s3{2.3,5.6};  //error
Sample s4{12,18};    //ok
```

Preventing Narrowing conversions with { }

- Simple variables
- Constructor arguments

nullptr

- A better replacement for NULL macro
- Keyword + special literal , of std::nullptr_t type
- Implicitly compatible with any other pointer type
- Not compatible with scalar types
- bool convertible
- std::is_null_pointer from type_traits (C++14)

Enum class

- Also known as scope enum, strongly typed enumeration types
- Must be scoped with enum type identifier
- Useful to avoid conflict between members for various enum types
- Not compatible with scalar types
- Variables can be declared as same name as enum members in case of scoped enums (Which is not allowed in case of normal enums)

Scoped enum

```
enum FanSpeed {Low, Medium, High};
enum Temperature {Low, Medium, High};
Temperature mode = High;
if (mode == Medium)//some code

enum class FanSpeed {Low, Medium, High};
enum class Temperature {Low, Medium, High};
Temperature mode = High; //error, not scoped
Temperature mode = Temperature::High;
FanSpeed fmode;
if(fmode == FanSpeed::Low)
//some code
```

static_assert

```
static_assert( sizeof(int*) == 4, "Pointer size is not 4 bytes");
static_assert( std::is_trivial<Sample>::value,
      "Sample type is not a trivial type");

template<size_t sz>
class MyBitset {
  static_assert(sz>=2, "Bitset size must be minimum 2");
};

template<typename T,size_t sz>
class MyBuffer {
  static_assert(sz % 1024 ==- , “Buffer size must be multiple of 1024");
};
```

- Performs compile time check , Causes compile time error if given condition is false
- Contrast to assert, which does run time check and causes runtime error

Raw string literals

```
“id":"sample",
"temperature":24,
"humidity":72
```

```
A Better way in C++11
const char* payload=R"({
    "id":"sample",
    "temperature":24,
    "humidity":72
}");
```

```
//How do you encode above as
//a string prior to C++11?
 const char *payload = "{    \
 \"id\":\"sample\",     \
 \"temperature\":24,    \
 \"humidity\":72        \
}";
```

Unicode literals

- UTF-8, UTF-16, UTF-32 representations
- Raw string literals in UTF form.

Binary Literals & Digit Separators

```
int bval=0b00100101; //0x45
uint8_t mask=0B00100000; //0x20
auto val=0B01000110; //0x46
```

Binary Literals

```
int x=1'235'659;
auto bval=0b100'1010;
auto oval=0234'724'451;
```

Digit Seperators

User defined literals

```
long double operator"" _kg(long double n) {
  return n*1000;
}
long double operator"" _mg(long double n) {
  return n/1000;
}
long double operator"" _km(long double n) {
  return n*1000;
}
long double operator"" _deg(long double n) {
  return n*pi/180.0;
}
```

```
long double distance=5.6_km;   //5600 meters
long double weight=2.34_kg+5623_mg;
                        //2340+5.623 grams
long double nradians=45_deg;
```

- ❑ User defined literals allow, custom suffix for better meaningful literals.
- ❑ UDLs can be implemented through operator overloading syntax (non member functions)
- ❑ Use _(underscope) prefix for custom literals, literals wthout prefix is meant for built-in literals

Some built-in literals

std::complex	==> operator""i, operator""if, operator""il
std::chrono::duration	==> operator""h, operator""min, operator""s
	operator""ms, operator""us, operator""ns
std::string	==> operator""s

Inline Namespaces

```
namespace org {
  namespace ltts {
    inline namespace examples {
      class Sensor { };
    }
  }
}
```

Here class HVAC is accessible without the need for scoping with nested namespace examples
i.e. org::ltts::Sensor is allowed instead of org::ltts::examples::Sensor

Classes & Objects

Outline

- 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

Default Controls

```
class Sample{
  int x;
  inty;
 public:
  Sample()=default;
  //Sample()=delete;
  Sample(const Sample&)=delete;
  //Sample(const Sample&)=default;
  ~Sample()=default;
};
```

Default controls
    =default
    =delete
Applicable for

- ❑ Default constructor
- ❑ Copy constructor
- ❑ Overloaded operator=
- ❑ Destructor (??)
- ❑ Parameterized constructor (??)

Significance/Impact of =default, =delete on each !!

Constructor Delegation

```
Sample::Sample(int p,int q):x(p),y(q) { }
Sample::Sample(int p): Sample(p,20) { }
Sample():Sample(10,20) { }
Sample(const Sample& rs):Sample(rs.x, rs.y) { }
```

```
Box(int l,int b,int h):m_l(l),m_b(b),m_h(h) { }
Box(int s):Box(s,s,s) { }
Box():Box(10,12,5) { }
Box(const Box& rb):Box(rb.m_l,rb.m_b,rb.m_h) { }
```

- ❑ Reuse common business logic among constructors
- ❑ Eliminates the need for helper functions

In Class Initializers

```
class Box{
  int l=10;   // int l{10};
  int b=12;   // int b{12};
  int h=5;    // int h{5};
  public:
};
```

```
class Sample {
  string str="abcd";
  const int cval=100;
  const int dval=200;
  //Box b1 { 10, 12, 5 };
  public:
};
```

- ❑ In class initializers for const variables
- ❑ Can we use runtime expressions (value not known at compile time)
- ❑ What if some member is initialized in class as well as by constructor?
- ❑ In class initializers for user defined types? Hint – uniform initializers

Uniform Initializers

```
class Box {
  int l;
  int b;
  int h;
  public:
  Box(int x, int y, int z):l(x),b(y),c(z) { }
};
int findVolume(Box b) { }
Box getInstance() { return {x,y,z}; }
  // return Box{10,12,5};
  //unnamed object, compatible with T&&
Derived::Derived(int p,int q)
  : Base{p}, y(q) { }
```

```
Box b1{10,12,5};
Box b2={10,12,5};
Box *pb=new Box{10,12,5};
findVolume( {10,12,5} );

Box b1{10.2,12,5};
//error, due to narrowing
conversion
```

Initializer List

```
std::vector<int> v1 { 10,20,30,40,50 };
std::list<Point> points { {1,2} , {2,3}, {3,4}, {5,6}, {7,8} };
//std::list<Point> points {Point{1,2} , Point{2,3}, Point{3,4} };
std::map<int,std::string> days { {1,“Sun"}, {2,“Mon"}, {3,“Tue"} };

void append(const std::set<std::string>& addSkills) {
  skillset.insert(addSkills.begin(), addSkills.end());
}
append( { “c++”, “Linux” } ); //no need to create set object explicitly
```

- ❑ Helpful to initialize data sets like STL containers
- ❑ Similar style of Uniform initialization, received thru instance of std::initializer_list at backend
- ❑ Works well with collection of objects, through uniform initializers or unnamed objects.

Initializer List

```
//Example – Own Implementation
class MyArray {
  int m_arr[max_sz];
  int m_len;
  public:
  MyArray(const std::initializer_list<int>& list):m_len(list.size()) {
     for(auto val:list) { m_arr[index++]=val; }
  }
  void fill(std::initializer_list<int>& list);
  void append(std::initializer_list<int>& list);
};

MyArray a1 {10,20,30,40,50 };
a1.append( {60,70,80} );
```

Explicit inheritance of constructors

```
class A {
  int x;
  public:A(int p):x(p) { }
};
class B : public A
{
  int y;
  public:
  using A::A;
  B(int p,int q):A(p),y(q) { }
};
class C : public B {
  public:
  using B::B;
};
```

```
B b1(10,20); //actual ctor
B b2(15);    //inherited ctor
C c1(11,12); //inherited ctor
C c2(18);    //inherited ctor
```

- ❑ Recall explicit inheritance of normal functions (allowed even in C++98)
- ❑ Explicit inheritance of constructors is meaningful when no specialization is required in Derived class

Overriding Example

```
class A {
  public:
  virtual void f1(const Sample&);
  virtual void f2();
  virtual void f3(const Sample&);
  virtual void f4() const;
};
class B : public A {
  public:
  void f1(Sample&); //mismatch
  void f2() const; //mismatch
  void f3(const Sample&);
  void f4() const;
};
```

```
Sample s1;
A *ptr = new B();
ptr->f1(s1); //??
ptr->f2(); //??
```

Override keyword

```
class A {
  public:
  virtual void f1(const Sample&);
  virtual void f2();
  virtual void f3(const Sample&);
  virtual void f4() const;
};
class B : public A {
  public:
  void f1(Sample&) override; //error
  void f2() const override;  //error
  void f3(const Sample&) override;
  void f4() const override;
};
```

- ❑ Override keyword detects if function is properly overriding
- ❑ Gives error in case of any mismatch, i.e. not overriding truly

Final keyword

```
class A {
  public:
  virtual void f1(const Sample&) final;
  virtual void f2() final;
  virtual void f3(const Sample&);
  virtual void f4() const;
};
class B : public A {
  public:
  void f1(Sample&);
  void f2() const;
  void f3(const Sample&) override final;
  void f4() const override final;
};
```

```
class FreezedType final  {

}; //Can’t inherit above class
```

- ❑ f1, f2 can’t be overridden beyond class A
- ❑ f3, f4 can’t be overridden beyond class B

Read only Objects

```
class Sample {
  int x;
  int y;
  public:
  constexpr Sample(int p,int q):x(p),y(q) { }
  constexpr int getx() const { return x; }
  constexpr int gety() const { return y; }
  constexpr int getsum() const { return x+y;
}
  int getdiff() const { return x – y; }
};

int main() {
  constexpr Sample s1(10,20);
  constexpr cx = s1.getx();
  constexpr cs = s1.getsum();
}
```

```
constexpr Sample
      s1(10,20);
constexpr cx =
      s1.getx();
constexpr cs =
      s1.getsum();
constexpr int d =
  s1.getdiff(); //error
Sample s2(11,12); //ok
int x = s2.getx();
constexpr int cy =
  s2.gety(); //error
```

Explicit Type Conversions

```
operator int() { //1
  return hh*3600+mm*60+ss;
}
explicit operator int() { //2
  return hh*3600+mm*60+ss;
}
operator int()=delete; //3
```

- ❑ If operator T() is explicit, user defined to scalar conversion allowed only with the syntax T(object)
- ❑ if operator T() is deleted, no conversion from user defined to scalar is allowed

```
val=t1;       //error in case of 2,allowed in case of 1
val=int(t1); //allowed in case of 1 & 2
val=(int)t1; //allowed in case of 1 & 2
val=t1.operator int(); //also works in case of 1 & 2,but not required
val=int{b1}; //error in case of 2
```

Explicit Type Conversions

```
MyTime(int);            //1
explicit MyTime(int); //2
MyTime(int)=delete;   //3
```

- ❑ If single argument constructor is deleted, object creation with single argument & scalar to user defined conversions are not allowed
- ❑ If single argument constructor is explicit, it can't be used for scalar to user defined conversion and usage of = is not allowed during object creation, usage of parenthesis is must for object creation.

```
MyTime t1(3720);  //error in case of 3,allowed in case of 1 or 2
MyTime t2=4000; //error in any case of 2 or 3, allowed with 1
//in case of 3 no object creation allowed with single argument
```

Explicit Type Conversions

```
MyTime& operator=(int); //4
MyTime& operator=(int)=delete;

t1=4500; //error in any case of 5,
         //allowed with 4
```

If overloaded assignment operator is deleted scalar to user defined conversion is not allowed.

Type Traits

- std::is_trivial
- std::is_pod
- std::is_reference
- std::is_null_pointer
- std::is_pointer
- std::is_enum
- std::is_class
- std::is_function
- std::is_array
- std::is_void
- std::is_abstract

- std::remove_reference
- std::add_lvalue_reference
- std::add_rvalue_reference,
- std::remove_pointer
- std::add_pointer
- std::remove_const
- std::add_const
- std::remove_volatile
- std::add_volatile
- std::remove_cv
- std::add_cv

- ❑ Header file : <type_traits>
- ❑ For more detailed listing please refer

  https://en.cppreference.com/w/cpp/header/type_traits

  http://www.cplusplus.com/reference/type_traits/

Move Semantics

Outline

- 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)

Trivial Types

- A class is said to be trivial type under following conditions
  - No virtual functions or virtual base class
  - Trivial or deleted default constructor
  - TriviallyCopyable type
    - Trivial or deleted copy constructor
    - Trivial or deleted assignment operator
    - Trivial non deleted destructor
    - Trivial or delete move constructor, move assignment operator (From C++11)
  - All data members & base classes are of Trivial types

Type Traits Check –

- std::is_trivial,
- std::is_trivially_copyable

Ref:- https://en.cppreference.com/w/cpp/named_req/TrivialType

Standard Layout Type

- A class is said to be of Standard Layout Type, if
  - No virtual functions, virtual base class
  - All non static data members under same access control
  - No non static data member of reference type
  - All non static data members & base classes are of standard layout type

Type Traits Check

- std::is_standard_layout

Ref:- https://en.cppreference.com/w/cpp/named_req/StandardLayoutType

POD Type

- A class is said to be of Plain Old Data(POD) Type, if
  - Trivial type
  - Standard Layout type
  - All non static data members are of POD type

Type Traits Check

- std::is_pod

Ref:- https://en.cppreference.com/w/cpp/named_req/PODType

Usage of this Type category is getting deprecated

MyString class

```
class MyString {
  char* m_buf;
  int m_len;
  public:
  MyString():m_buf(nullptr),
                  m_len(0) { }
  MyString(const char* pbuf) {
    m_len=strlen(pbuf);
    m_buf=new char[m_len+1];
    strcpy(m_buf,pbuf);
  }
  MyString(const MyString& ref):

      m_len(ref.m_len) {
    m_buf=new char[m_len+1];
    strcpy(m_buf,ref.m_buf);
  }
```

```
  ~MyString() {
    if(m_len>0)delete[] m_buf;
  }
  MyString& operator=(
            const MyString& ref) {
    if(m_len>0) delete[] m_buf;
    m_len=ref.m_len;
    m_buf=new char[m_len+1];
    strcpy(m_buf,ref.m_buf);
    return *this;
  }
  //prints -const
  //length -const
};
```

Move Semantics

- Move operations prevent deep copy/duplication of underlying resources if one or more members are of non-trivial type (Contrast to cloning, efficient resource mapping)
- Prevention of such deep copy (shallow copy) is manageable/applicable certain times,
  - when source object has little lifetime (unnamed/temporary)
  - unique ownership is expected on underlying resources
- Please note that move operation doesn’t move object’s own memory (Of course not possible). Underlying resources are detached from source and attached to destination, i.e. transfer of resources from source object to destination
- Some move operations in standard library
  - Adding objects to STL containers (push_back)
  - Unique pointers, single ownership on managed objects
- Move operations are distinguished from copy operations through r-value references (Syntax wise)

R-value references : Primitive Types

```
int x=10;
int &r1 = x;       //ok
int &r2 = 10;      //error
const int& rc=10; //ok
int &&r3 = 10;     //ok
int &&r4 = x+5;    //ok
int &&r5 = x;      //error
int &&r6 = std::move(x); //ok,
but not meaningful
//for scalar/trivial types
```

R-value references and move operations are not meaningful for scalar/trivial types

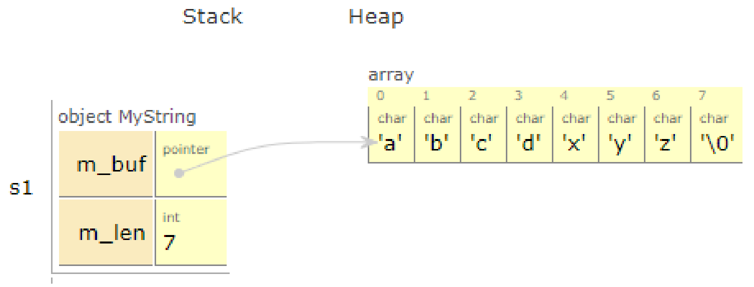
R-value references : user defined types

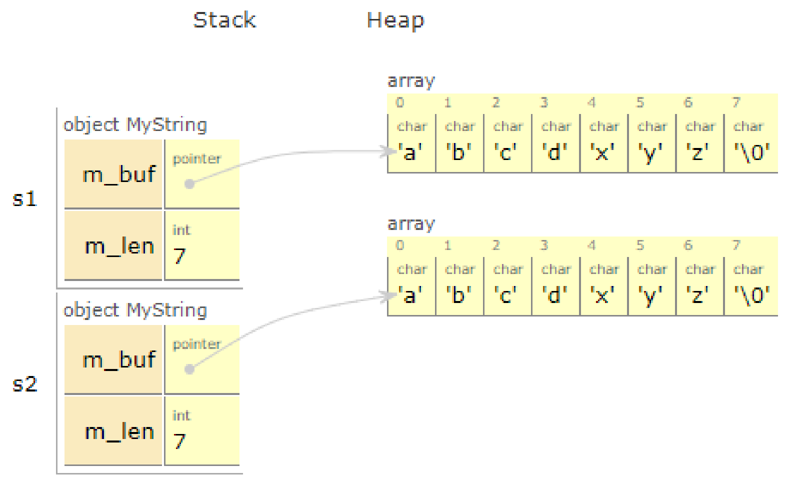
```
Box b1{10,12,5};
Box &r1=b1; //ok
Box &&r2 = b1; //error
Box &&r3 = std::move(b1); //ok

Box &ref=Box(10,12,5); //error
const Box &cref=Box(10,12,5);//ok
Box &r2 = {10,12,5}; //error
const Box &rc = {10,12,5}; //ok
Box&& rr = Box(10,12,5); //ok
Box&& rr = {10,12,5}; //ok
```

- ❑ A named object must be casted by std::move to be compatible with r-value references
- ❑ Unnamed objects are compatible with r-value references.
- ❑ These are not compatible with non const l-value references

Copy Operations

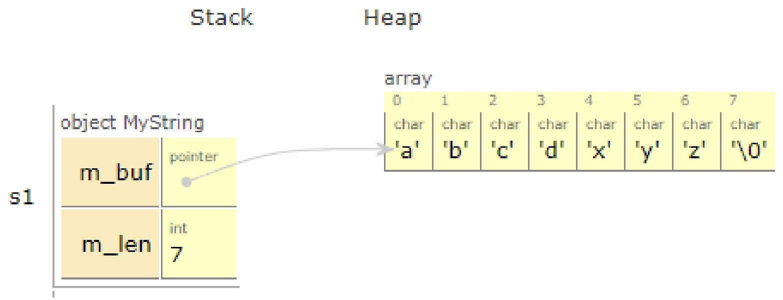
```
MyString::MyString(const MyString& rs)
                    :m_len(rs.m_len) {
  m_buf=new char[m_len+1];
  strcpy(m_buf,rs.m_buf);
}

MyString& operator=(const MyString&);
MyString s1("abcdxyz");
MyString s2(s1);
MyString s3;
s3 = s1;
```

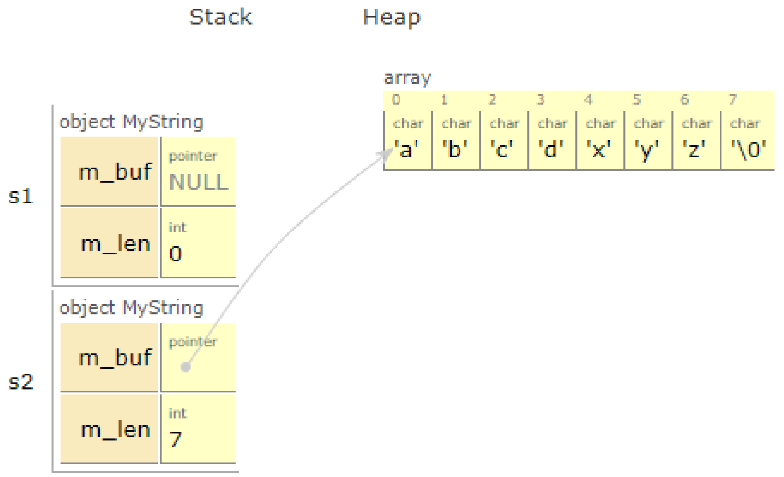

Stack
Heap
array
0 1 2 3 4 5 6 7
char 'a' char 'b' char 'c' char 'd' char 'x' char 'y' char 'z' char '\0'
object MyString
s1
m_buf
pointer
m_len
int
7


Stack
Heap
array
0 1 2 3 4 5 6 7
char 'a' char 'b' char 'c' char 'd' char 'x' char 'y' char 'z' char '\0'
array
0 1 2 3 4 5 6 7
char 'a' char 'b' char 'c' char 'd' char 'x' char 'y' char 'z' char '\0'
object MyString
s1
m_buf
pointer
m_len
int
7
object MyString
s2
m_buf
pointer
m_len
int
7

Move Constructor

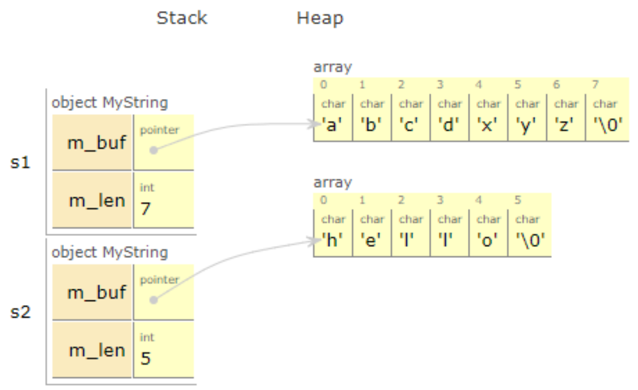
```
MyString::MyString(MyString&& rr) {
  m_len=rr.m_len;
  m_buf=rr.m_buf;
  rr.m_len=0;
  rr.m_buf=nullptr;
}
MyString s1("abcdxyz");
MyString s2(std::move(s1));
```

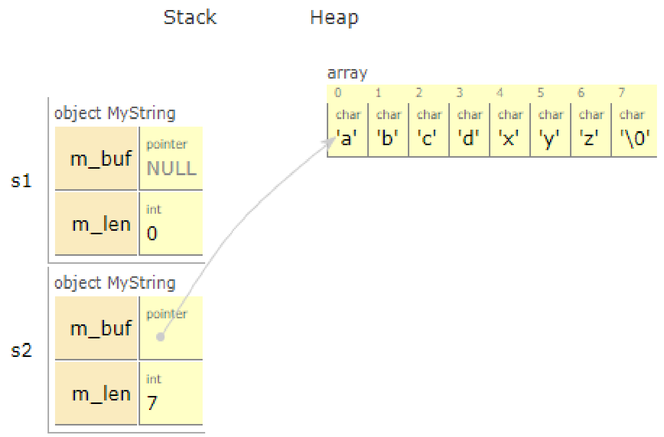

Stack
Heap
array
0 1 2 3 4 5 6 7
char 'a' char 'b' char 'c' char 'd' char 'x' char 'y' char 'z' char '\0'
object MyString
s1
m_buf pointer
m_len int 7



Move Operator=

```
MyString& MyString::operator=
                  (MyString&& rr) {
  assert(this != rr);
  if(m_buf!=nullptr) delete[] m_buf;
  m_len=rr.m_len;
  m_buf=rr.m_buf;
  rr.m_len=0;
  rr.m_buf=nullptr;
  return *this;
}
MyString s1("abcdxyz");
MyString s2("hello");
s2 = std::move(s1);
```





Rule of three/five/zero

- Trivial vs Non Trivial classes
  - Copy Constructor, T(const T&)
  - Move Constructor, T(T&&)
  - Assignment operator , operator=(T&)
  - Move assignment operator, operator=(T&&)
  - Destructor
- For trivial classes all of the above five operations can be provided by compiler implicitly.
- For non trivial classes all of the above five operations should be provided by programmer, implicitly defined operations will cause side effects, e.g. shallow copy
- No class will require few of the above five, either zero or all five will be required.

Implicitly defined members

- The following operations will be provided by compiler implicitly in absence of user defined member (With few exemptions)
  - Default constructor
  - Copy Constructor
  - Move Constructor
  - Assignment operator
  - Move assignment operator
  - Destructor
- The generation of implicitly defined copy constructor is/getting deprecated (implicitly deleted) if
  - User defined move constructor or move assignment operator
  - User defined destructor (Not yet)
  - User defined assignment operator (Not yet)
- =default can be used to request compiler to enable implicit version (Prevent deletion)

Move Operations in STL Containers

```
std::vector<MyString> sarr(5);
MyString s1("abcdxyz");
sarr.push_back(std::move(s1));
```

- ❑ std::vector provides overloaded push_back operations which takes r-value references of instantiated type
- ❑ this prevents additional copy of resources in source object

s1 need not maintain the buffer any more once a copy of it(which is moved) is pushed on to vector

Universal References

```
void print(const Sample& r) {
  std::cout << "l-value ref\n";
}
void print(Sample&& r) {
  std::cout << "r-value ref\n";
}
template<typename T>
void test(T&& rr) {
  print(rr);
}
void check(auto&& rr) {
  print(rr);
}
```

```
int main() {
  Sample s1;
  test(s1); //check(s1)
  test(std::move(s1)); //check
  test(Sample(11,12)); //check
  return 0;
}
```

T&& doesn't mean r-value ref all the times. This is known as reference collapsing and such references are known as Universal References.

Same scenario with auto&&

Universal References

```
void print(const Sample& r) {
  std::cout << "l-value ref\n";
}
void print(Sample&& r) {
  std::cout << "r-value ref\n";
}
template<typename T>
void test(T&& rr) {
  print(std::move(rr));
}
void check(auto&& rr) {
  print(std::move(rr));
}
```

Both l-value, re-value references are forwarded as l-value references in this case due to std::move

How to achieve perfect forwarding??

Perfect Forwarding

```
void print(const Sample& r) {
  std::cout << "l-value ref\n";
}
void print(Sample&& r) {
  std::cout << "r-value ref\n";
}
template<typename T>
void test(T&& rr) {
  print(std::forward<T>(rr));
}
void check(auto&& rr) {
 print(std::forward<decltype>(rr));
 //delctype(std::remove_reference(rr));
}
```

std::forward is also helpful in handling variadic arguments, we'll discuss this during templates

This is known as perfect forwarding,

l-value ref will be forwarded as l-values and r-value references as r-values only

Value Categories

Lvalue	• Have identity but can't be moved, e.g. normal variables, named objects, functions returning l-value, ++var, *ptr etc.
Xvalue	• Have identity but can be moved, e.g. casting with std::move, functions returning rvalue ref, anonymous/temporary objects
Prvalue	• No identity, but can be moved, e.g. literals, expressions like var++, a + b , &var, functions not returning any ref etc.
Glvalue	• either lvalue or xvalue [Have identity]
Rvalue	• either prvalue or xvalue [Can be moved]

Ref:- https://en.cppreference.com/w/cpp/language/value_category

Lambdas and Callbacks

Outline

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

Need for Lambda expressions

```
void test(int x, int y,
        int (*fcomp)(int,int)) {
  int z;
  z=fcomp(x,y);
}

int sum(int x, int y) {
  return x*x + y*y;
}

int sumsqr(int x, int y) {
  return x*x + y*y;
};

test(10,20,sum);
test(10,20,sumsqr);
```

```
bool mycompare(int x) {
  return x % 3 == 0; // x > 30
}

std::vector<int> v1{12, 37, 25, 56, 48};
std::count_if(v1.begin(), v1.end(),
                             mycompare);
```

Solution with lambda expressions

```
void test(int x, int y,
        int (*fcomp)(int,int)) {
  int z;
  z=fcomp(x,y);
}

test(10,20,[](int x,int y) {
  return x + y;
});

test(10,20,[] (int x, int y) {
  return x*x + y*y;
});
```

```
std::vector<int> v1{12, 37, 25, 56, 48};
std::count_if(v1.begin(), v1.end(),
                           [](int x) {
   return x % 3 == 0; // x > 30
});
```

Lambdas for effective callback mechanism, without the need for defining named functions in advance

Another Example

```
std::array<int,10> data; //std::vector<int> data(10);
std::for_each(data.begin(), data.end(), [](int& r) {
  r=rand()%100;
});
std::for_each(data.begin(), data.end(), [](int x) {
  sum+=x;
});
```

Pass by reference

```
//Naming Lambdas
auto sum = [] (int x,inty) {
  return x+y;
};
```

Capture Syntax

```
std::vector<int> tarr{18,25, 16, 26, 22};
constexpr int tmax=30;
std::for_each(tarr.begin(), tarr.end(), [=] (int tval) {
  if(tval > tmax)
  //raise error / take some action
});
```

Capture all vars by value, tmax in this example

```
std::vector<Account> accounts;
constexpr double minbal = 1000;
std::count_if(accounts.begin(), accounts.end(), [minbal](Account & ref) {
  return ref.getBalance() > minbal;
}
```

Capture Modes

[=]	capture all variables by value
[ref]	capture all variables by reference
[val]	capture single variable by value
[&val]	capture single variable by reference
[=, &x]	capture all by value, specific variable by reference
[x, y]	capture both x, y by value
[&x, &y]	capture both x, y by reference
[&x, y]	capture x by reference, y by value
[this]	capture current object(*this) by reference
[std::move(expr)]	Move capture
[]	no capture

Generic Lambdas & Move Capture

```
auto f1= [](auto x, auto y) {
  return x + y;
}
```

```
[tmax=30] (int x) {
if(x > tmax) …
}

[ rr= std::move(s1)] (int x) {
  //captured move instance of s1
}
```

Lambda parameters can be specified with auto keyword

No need of trailing return syntax from C++14

tmax captured directly in lambda

move capture

std::function

```
#include<functional>
std::function<int(int,int)> fsum=sum;
std::function<int(int,int)> f1= [] (int x,inty) {
  return x+y;
};
```

```
//void test(int x,int y,auto cb); //c++14
void test(int x,inty,
         std::function<int(int,int)> cb) {
  int z;
  z=cb(x,y);
  //print z
}
test(10,20,fsum);
test(10,20,f1);
```

Alternative way of declaring callbacks, instead of conventional function pointers / named functions

More useful to create array of callbacks, collections using STL containers (Dispatch Table)

Table of Functions

```
std::function<int(int,int)> ftable[4];
int sum(int x,int y) { return x + y; }
int div(int x,int y) { return x / y; }
auto pro = [] (int x, int y) {
  return x * y;
};
std::function<int(int,int)> fdiv = div;
ftable[0] = sum;
ftable[1] = [] (int x, int y) {
  return x - y;
};
ftable[3] = pro;
ftable[4] = fdiv; //div
res = ftable[index](a, b);
test( a, b, ftable[index] );
std::function<int(int,int)> fcomp = ftable[index];
```

Table of Functions – STL Containers

```
std::vector < std::function<int(int,int)> > fvector
std::list < std::function<int(int,int)> > flist;
std::map < int, std::function<int(int,int)> > fmap;
using fcomp = std::function<int(int,int)>;
```

```
using fcomp_t = std::function<int(int,int)>;
std::vector<fcomp_t> fvector
std::list<fcomp_t> flist;
std::map <int, fcomp_t> fmap;
```

std::bind simple usage

```
bool bcompare(int x , int y ) {
  return x > y;
}

using std::placeholders::_1;
using std::placeholders::_2;
//using namespace std::placeholders;
std::function<bool(int)> ucomp;
ucomp=std::bind(compare, ::_1, maxval);
//auto ucomp = std::bind(compare, ::_1,
maxval);
//ucomp(x)==> compare(x, maxval);
std::function<bool(int)> fcomp;
fcomp=std::bind(compare, maxval, ::_1);
//fcomp(y) ==> compare(maxval, y)
```

- ❑ std::bind is useful to minimize no. of arguments to a function by fixing some arguments with known values and other parameters with placeholders (forward parameters passed to resultant function)
- ❑ The resultant function can be passed to another function as callback, stored by name in an instance of std::function or using auto keyword.

std::bind for callbacks

```
bool bcompare(int x , int y ) {
return x > y;
}
std::vector<int> v1{12, 37, 25, 56, 48};
int tmax = 30;
std::count_if(v1.begin(), v1.end(), std::bind(bcompare, ::_1, tmax) );
std::function< bool(int) > ucompare = std::bind(bcompare, ::_1, tmax);
//auto ucompare = std::bind(bcompare, ::_1, tmax);
std::count_if(v1.begin(), v1.end(), ucompare);
std::count_if(v1.begin(), v1.end(),
      std::bind(std::greater<int>(), ::_1, tmax) );
std::count_if(v1.begin(), v1.end(),
      std::bind(std::less<int>(), tval, ::_1) );
```

// Conversion from comparators as predicates in STL algorithms,
// i.e. Binary functions as unary functions

Another example

```
bool checkTemperature(Weather& wref, tint tmin, int tmax) { //3 args
  return wref.getTemperature() > tmin && wref.getTemperature() < tmax;
};
auto isValidTemperature = std::bind(checkTemperature, _1, 18, 30 );//1 arg
auto minTemperature = std::bind(checkTemperature, _1, 18, ::_2 ); //2 args
auto maxTemperature = std::bind(checkTemperature, _1, _2, 30 );  // 2 args
//auto vs std::function usage
std::list<Weather> allRecords; //Assume some data
std::list<Weather> validRecords;
std::count_if(allRecords.begin(), allRecords.end(),
std::bind(checkTemperature, ::_1, 18, 30));
std::copy_if(allRecords.begin(), allRecords.end(),
std::back_inserter(validRecords), isValidTemperature);
```

Binding objects

```
class Box {
  int l;
  int b;
  int h;
public:
Box(int x, int q, int r):l(x),b(y),h(r) { }
int volume() { return l * b * h; }
void zoom(double scale) { }
Void update(int p,int q, int r);
};
Box b1(10,12,5);
auto fvolume = std::bind(&Box::volume, b1);
auto fzoom = std::bind(&Box::zoom, b1, ::1);
Auto fupdate = std::bind(&Box::update, b1 , ::1, ::2, ::3);
//fvolume(15) ==> b1.volume();
//fzoom(1.5)  ==> b1.zoom(1.5);
```

Converting member functions as global functions, this is useful when some API can accept global functions only

std::bind vs lambdas

```
std::count_if(v1.begin(), v1.end(), std::bind(bcompare, ::_1, tmax) );
==> std::count_if(v1.begin(), v1.end(), [](int val) {
  return bcompare(val, tmax); //return val > tmax;
});
std::count_if(v1.begin(), v1.end(),
                         std::bind(std::greater<int>(), ::_1, tmax) );
==> std::count_if(v1.begin(), v1.end(),[](int val) {
  return std::greater<int>(val, tmax);
});
std::count_if(allRecords.begin(), allRecords.end(),
                                std::bind(checkTemperature, ::_1, 18, 3));
==> std::count_if(allRecords.begin(), allRecords.end(),
                                          [](const Weather& wref) {
  return wref.getTemperature() > 18 && wref.getTemperature() < 30;
});
auto fsample = std::bind(&Sample::test, s1, ::_1);
==> auto fsample = [](int val) { return s1.test(val); };
```

Thank You !

L&T Technology Services