Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

This Lecture

Write even less code

- **Callable Object**
  - Passing a function Ch. 10.3
    - Review: Function pointers, Ch. 6.7
    - Functors, Ch. 14.8
    - C++11: bind, Ch. 10.3.4
    - C++11: Lambdas, Ch 10.3.2-10.3.3
  - Aside
    - for_each and C++11: range for loop

Function Pointers

- **STL, GUIs, etc. expect to pass a callback function.**
  – We can use objects with operator overloading (functors).
  – But in simple cases a function is enough
  – Example: lessThan

```
// Function declaration
bool lessThan(const Point2D&, const Point2D& );
// Function accepting a function pointer
const Point2D& compare( const Point2D&, const Point2D&,
                 bool (*) (const Point2D&, const Point2D&) );
// Function definition
bool lessThan(const Point2D& ptA, const Point2D& ptB ) {
  return ptA.d_components[1] < ptB.d_components[1];
}
```

Review: Using Function Pointers

- **Function pointer types have to match**
  - arguments and *return type (unlike function overloading)*
- **Function pointers have awkward syntax**
  - We can use simplified notations, or, we can use typedefs
  - Example: lessThan (continued)

```
// Still the same function accepting a function pointer
// but no (*)
const Point2D& compare( const Point2D&, const Point2D&,
                 bool (const Point2&, const Point2D&) );
// using a typedef - pre C++11 style
typedef bool (*pt_compare)(const Point2D&, const Point2D&);
const Point2D& compare( const Point2D&, const Point2D&,
                        pt_compare );
```

Review: Calling Function through Pointers

optional

- explicitly dereferenced
- implicitly dereferenced

```
// Function declaration
bool lessThan(const Point2D&, const Point2D& );
// no typedef
Point2D ptA, ptB;
lessThan( ptA, ptB ); // direct call of function, no ptr
bool (*ptr) ( const Point2D&, const Point2D& ) = &lessThan;
(*ptr)(ptA,ptB);
ptr(ptA,ptB);
// using a typedef C++11 notation
using pt_compare=bool (*)(const Point2D&, const Point2D&);
pt_compare c = &lessThan;
(*c)(ptA,ptB);
c(ptA,ptB);
```

Function Objects

- **Sometimes functions need to keep track of a state, e.g.:**
  - Maintain a count, unique resource etc. for a specific function
  - Function needs to be initialized before it is passed on to another method
- **Solution**
  - Create a class
  - Overload `operator()`

Example: Initialization Functor

– Used in example `function.cpp` to assign values to an array.
  - State variable `d_count`

```
struct intFunctor {
  int d_count;
  intFunctor(int _count) : d_count(_count){};
  void operator()(int& e) {
    e=++d_count;
    return;
  }
};
```

std::bind

- **Adjusting parameter lists for functions**
  - fixing values of parameters
  - reordering parameters
  - parameters for the new callable are named with placeholders
  - new callable is of template type `std::function`
  - `bind` is in `std` but placeholders `_1`, `_2`, … are in `std::placeholders`
  - default behaviour is to do call by value but can specify `ref` for reference or `cref` for constant reference

Bind Example

```
using std::bind; using std::placeholders::_1; using std::cref;

template <class T, const int NUM>
bool lessThan(const Point<T,NUM>& ptA, const Point<T,NUM>& ptB ) {
  return ptA.d_components[0] < ptB.d_components[0] || (
    !(ptB.d_components[0] < ptA.d_components[0]) &&
    ptA.d_components[1] < ptB.d_components[1]);
}

int main() {
  …
  Point<int,2> iPt1(initA), iPt2(initB), iPt0(zeroVal);
  std::function<bool( const Point<int,2>& )> lessThanZero =
    bind(lessThan<int,2>,_1,cref(iPt0));
  auto greaterThan = bind(lessThan<int,2>,_2,_1);
  if (lessThan(iPt2,iPt1)) { … }
  if (lessThanZero(iPt1)) { … }
  if (greaterThan(iPt1,iPt2)) { … }
}
```

Better loops

- **Better loops**
  - C++11: Range for loops
    - similar to Java
    - makes looping through a container cleaner
    - use with auto for maximum gain
- **std::for_each**
  - ensures each element in a container is processed exactly once (no breaks or continues)
  - uses iterators, i.e., more flexibility than range loops (start not at the beginning, or end not at the end).

Loop Example

```
template <class T>
void printElements( const T& _container ) {
  // C++11 loop and print using auto and for range
  for ( auto &element : _container ) {
    cout << element;
  }
  cout << endl;
  return;
}

// Using a function ptr and std::for_each
template<class T, const int NUM>
void print( const Point<T,NUM>& _pt ) {
  cout << _pt;
}
for_each( pVec.begin(), pVec.end(), print<int,2> );
```

Lambdas

  - Generic programming requires function as arguments (e.g., as predicates in a sort) but only once
- **Java solution**
  - Anonymous inner classes
  - And with JDK 8 lambdas = anonymous functions
- **Lambdas in C++**
  - Anonymous inline functions

C++ Lambdas

- Lambdas have arguments and return types
  - Return type can be inferred automatically if lambdas have a simple return statement at the end
  - Return type can also be explicitly defined
- Lambdas also have a capture list
  - Capturing variable and references from the calling context
  - Captured variables provide direct access
  - Either by value or reference

Lambdas Syntax

```
lambda:
    capture parameters_opt return-type_opt {function body};
capture:
    to be defined later
parameters:
    (parameter-list)
parameter-list:
    type-specifier parameter-name {, parameter-list}_opt
return-type:
    -> type-specifier
function-body:
     omitted here
```

Use of Lambda

- **Using std::function template makes sure that a function can be called with a functor, function pointer or lambda**
  - i.e., any callable

```
#include <functional>
template<class T,const int N>
class Store {
  std::array<T,N> data;
public:
  void apply_to_all(std::function<void(T&)> f) {
    std::for_each( std::begin(data), std::end(data), f );
  }
};
…
 Store<int,10> st;
 st.apply_to_all([](int &e) { std::cout << e << " ";});
```

Lambda Captures

- Value captures
  - take the current value of a variable in the calling context where the lambda is **defined**
  - can be defined mutable if value is to be updated by the lambda
- Reference captures
  - “pass by reference” from the calling context
  - variable will be current during execution of the lambda and can also be updated
- For implicit captures the compiler attempts to deduce what variable to use in the lambda
- Explicit captures if automatic capture is not desired

Lambdas Capture Syntax

```
capture:
    []
    capture-list
capture-list:
        [mixed-list] mutable_opt
    [= {, reference-list}] mutable_opt
    [& {, value-list}_opt] mutable_opt
    [&]
    [reference-list]
mixed-list:
    reference-name, mixed-list
    value-name {, mixed-list} opt
value-list:
    value-name {, value-list} opt
reference-list:
    reference-name {, reference-list}_opt
```

Example

```
int a=0, b=1, c=1; string article = "A ", noun = "string";

auto val_cap = [=] { return a;};
++a; // will not affect value capture!
cout << "val_cap(): " << val_cap() << endl;

auto ref_cap = [&] { return ++b;};
++b; // will affect reference capture
cout << "ref_cap(): " << ref_cap() << endl;

auto val_cap_mutable = [=] () mutable  { return ++c;};
// c changed but no effect on calling context
cout << "val_cap_mutable(): " << val_cap_mutable() << endl;

auto mixed = [=,&article] { article="The ";
  return article+noun; };
```

Next

No reinventing the wheel

- **Standard Template Library**
  - Textbook (Lippman): Chapters 10.1-10.2, 10.4, 11.1-11.4
  - Review: Java Collections Framework
  - Sequential STL containers and container adaptors
  - STL iterators
  - Associative STL containers
  - Generic algorithms