C++ Refresher Tutorial

April 5th, 2019

Outline

- C++ Core Concepts with C++11 Emphasis
- C++ Standard Library
  - Containers
  - Algorithm & Functional
- C++ Code Compilation, Toolchain, and Workflow
  - Compiling on the Command Line
  - Makefiles

C++ Core Language Constructs

Basic Data Types

- Integer & Floating Point Types
  - [`unsigned`] `char`, `short`, `int`, `long`
  - `float`, `double`
- Pointers
  - Use `nullptr`, not the macro `NULL`
- Arrays
  - Statically allocate by `Type foo[n]`
  - Decay into pointers, e.g. `int[]` → `int *`
- Strings
  - Can be `char *` or `std::string`
- References
  - Denoted with ampersand: `Type&`
  - "Safe" pointers: can't be null

C++11 & Memory Management

- Any memory allocated dynamically must be freed.

- C: malloc & free

```
int *arr = (int *) malloc(n * sizeof(int));
free(arr);
```

- C++: new & delete, with [] for arrays

```
Foo *f = new Foo();
delete f;

int *arr = new int[4];
delete[] arr;
```

C++11 & Memory Management

- C++11 introduces *smart pointers*, which automatically manages dynamic memory for you.

- `std::unique_ptr` lets only one variable reference the given memory, and releases when out of scope.

- `std::shared_ptr` lets multiple variables reference the memory, only releasing when no one references it.

```
#include <memory>

std::shared_ptr<Foo> p_foo = std::make_shared<Foo>(...);
```

Memory Management Example

- Open `memorymgmt.cpp`
- Three examples to implement
  - `malloc` & `free`
  - `new` & `delete`
  - `std::shared_ptr<T>`

- What do you notice?

Structs & Classes

- Both very similar in C++
  - Structs have default public members
  - Classes have default private members
- Use `this` keyword to refer to class members or functions
  - Can be omitted if clear
- Constructors typically initialize & allocate resources
- Destructors typically release resources

```
class Foo {
int a_; // private
public:
  Foo(int f);
  ~Foo();
   void bar();
private:
  void bar1();
};

Foo f(1);
f.bar();
f.bar1(); // compile error

Foo *f1 = new Foo(1);
f1->bar();
```

Inheritance & Polymorphism in C++

- To allow subclasses to provide custom implementations, declare base function `virtual`
- Subclass must have same method signature to override
  - Optionally put `override` to let compiler verify
- For “pure” base classes, provide no implementation by setting = 0.

```
class A {
public:
  virtual void foo() = 0;
};

class B : public A {
public:
  void foo() override {...}
};

B b(); // normal
A *b1 = &b;
b1->foo(); // calls B::foo
```

Inheritance & Polymorphism Example

- Open `inherit.cpp`

- Key Takeaway
  - Even if you have a pointer to a superclass, C++ will call the derived function unless you explicitly say not to

Operator Overloading

- It’s useful to define custom operations on our objects.
- C++ allows you to override most operators like
  - Math: +, -, *, /, &, |, ~, ^, ++, etc.
  - Comparison: &&, ||, !, !=, ==, etc.
  - Array [ ] and function call ( )
  - Assignment =
  - Stream operators << and >>
- Stream operators **cannot** be defined as a member function

```
struct Foo {
  int bar;
};

Foo Foo::operator +(const Foo& b) {
  return Foo(this->bar + b.bar);
}
```

Templates

- Some algorithms and data types are data-agnostic
- Use templates to specify placeholder types!
- Add `template <typename T>` before your function or class definition
  - Does not have to be `T`, anything is fine

```
template <typename T>
struct Foo {
  T data;
};
Foo<int> f(); // holds ints

template <typename U>
U foobar(const U& input);
U u1(...);
U u2 = foobar(u1); // type inferred
```

Exceptions

- If you encounter something that breaks pre- or post-conditions, throw an exception
- Similar in idea to assertions but exceptions can be handled
- Useful when testing edge cases in code

```
#include <stdexcept>

void foo() {
  ...
  if (something bad) {
    throw std::exception(“yikes”);
  }
  ...
}

try {
  foo();
} catch (const std::exception& e){
  cerr << “caught” << endl;
}
```

DenseMatrix Example

- Open `densematrix.cpp`

- Key takeaways:
  - We can overload the () operator with two versions: a getter and setter
  - Stream operators are not class functions. Require separate template parameter and friend keyword to access private functions.

Lambdas

- C++11 introduces *lambdas,* which are like mini functions
- Also known as predicates or anonymous functions
- General form:

```
[capture group](parameters) { return ... }
```

- Capture group: allows variables from outer scopes to be used inside
  - Pass by value: `[variable]`
  - Pass by reference: `[&variable]`
  - Class Member variables: `[this]`
  - Pass everything by value: `[=]`
  - Pass everything by reference: `[&]`

- Parameter list usually defined by function taking lambda.
- Lambdas do not have to be simple one line statements!

C++ Standard Library

Containers

- `std::vector<T>`: resizeable array
  - `std::vector<T>(n)` – set size
  - `::resize(n)` – expands/shrinks vector
  - `[index]` – get/set element
  - `::push_back(T)` – insert at end of vector

- `std::list<T>`: doubly linked lists
  - Most operations are the same
  - Some special operations unique to lists, like `::sort`

- `std::queue<T>`: standard FIFO
  - Given some other container, only allow pop/enqueue operations

Iterators

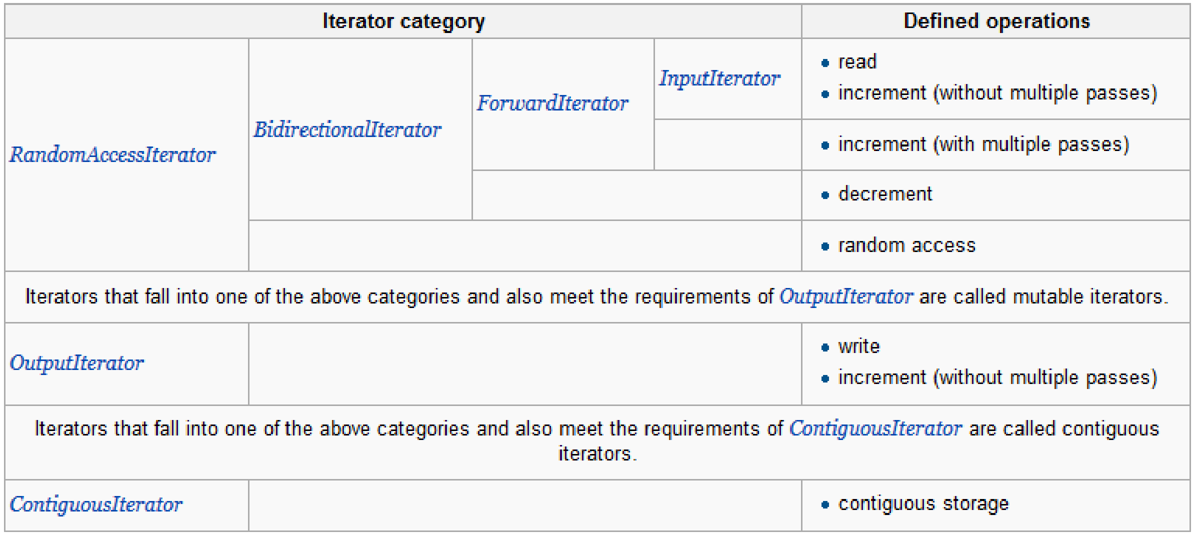
- Containers have `begin()` and `end()` functions for easy iteration

```
std::vector<T> foo = ...
auto& itr = foo.begin();
while (itr != foo.end()) {
  ...
  itr++;
}
```

- C++11 introduced ranged for loop

```
for (auto& i : foo) { ... }
```

- Not all iterators created equal:



Iterator category				Defined operations
RandomAccessIterator	*BidirectionalIterator*	*ForwardIterator*	*InputIterator*	• read • increment (without multiple passes)
				• increment (with multiple passes)
				• decrement
				• random access
Iterators that fall into one of the above categories and also meet the requirements of *OutputIterator* are called mutable iterators.				
OutputIterator				• write • increment (without multiple passes)
Iterators that fall into one of the above categories and also meet the requirements of *ContiguousIterator* are called contiguous iterators.				
ContiguousIterator				• contiguous storage

The `<algorithm>` Header

- `std::for_each(InputIt first, InputIt last, <lambda>)`
  - Lambda: `[](T& item) { … }`
  - Apply a lambda to each element
- `std::transform(InputIt first, InputIt last, InputIt dst, <lambda>)`
  - Lambda: `[](T& item) { return … }`
  - Apply a lambda to each element and put it in another place
- `std::sort(InputIt first, InputIt last, <lambda>)`
  - Lambda: `[](const T& a, const T& b) { return true }`
  - Sorts elements according to given lambda or default comparison

The `<numeric>` Header

- `std::accumulate(InputIt first, InputIt last, T init, <lambda>)`
  - Lambda: `[](T& sum, U& val) { return new_sum }`
  - Add all elements according to given lambda

- `std::iota(ForwardIt first, ForwardIt last, T val)`
  - Same idea as `range_iterator` from Lecture 1
  - Start at `val` and increment until done

```
std::vector<int> foo(10);
std::iota(foo.begin(), foo.end(), 0);
// foo = [0, 1, 2,..., 9]

int sq_sum = std::accumulate(foo.begin(), foo.end(),
  [](int& sum, int& val) { return sum + (val * val); }
);
```

Numeric Practice

- Open `numeric.cpp`
- Goal: summing every other element in a vector

C++ Compilation & Tools

Compiling Code on the Command Line

- Most code in CME 213 will be compiled via command line

- General order of flags for gcc/g++

```
g++ -I{include} -l{linking} {C/CXXFLAGS} <file>
```

- Example

```
g++ -o main -std=c++11 -Wall -g main.cpp
```

- `-std=c++11` enforces the C++11 standard
- `-Wall` turns on all warnings
- `-g` compiles in debug info

- I like to use `-pedantic` (no extensions) and `-Wextra` sometimes

Compiling via Makefiles

- Annoying to manually specify flags and file every time
- Makefiles makes this easier!
- Run on command line: `make <target>`

```
CXXFLAGS=-g -std=c++11 -Wall
INCLUDE=include/

default: main

main: main.cpp
  g++ $(CXXFLAGS) -I$(INCLUDE) $< -o $@

clean:
  rm -f *.o main
```

Wrapping up…

- Should know basics of:
  - Smart pointers
  - Operator overloading
  - Inheritance and polymorphism
  - Templates, Exceptions, Lambdas
  - Standard Library Headers

- Mastery not necessary!
- Ability to google these features is good enough
- HW1 is the most C++ feature-heavy!

Any Questions?